UNIVERSITÀ DEGLI STUDI "ROMA TRE"

DIPARTIMENTO DI MATEMATICA E FISICA

SCUOLA DOTTORALE IN SCIENZE MATEMATICHE E FISICHE

SEZIONE DI MATEMATICA
XXX CICLO

PH.D. THESIS

Exploiting GPUs to speed up cryptanalysis and machine learning.

Candidate:
Marco CIANFRIGLIA

Supervisor:
Dr. Massimo BERNASCHI

Coordinator:
Prof. Angelo Felice LOPEZ

A.A. 2017-2018

Abstract

In the past cryptography was mainly used by military organizations and governments for secure communication. However today almost everyone uses it everywhere, even if most people do not know it. Cryptography is used in many different scenarios; we use it when we login to a secure website or to our laptops, or when we digitally sign a document; cryptocurrencies like Bitcoin rely on it [1]; we use cryptography even when we chat with friends. As different situations may have different requirements and constraints, there exist various kind of ciphers to answer different needs. As an example, where resources are limited and speed is critical, we usually rely on Stream Ciphers. They are symmetric key ciphers that try to mimic the properties of the perfectly secure cipher One Time Pad (OTP) [2], by building a mechanism that is able to produce a very long pseudorandom keystream that should look like a truly random sequence of bits. This keystream is then xored with the message we want to encrypt. The keystream is serially generated starting from an initial random seed (the key and the IV) that is used to initialize one or more Feedback Shift Registers (FSR) and sometimes some Finite State Machines (FSM) [3, 4]. A stream cipher is said to be secure if any passive attacker can not learn anything about the stream cipher internal state and the key by looking at the keystream bits. Furthermore, the period[1] of the cipher should be long enough to avoid reusing the same keystream for more than one encryption. In literature [5] there exist many cryptanalysis techniques that focus on testing the security of Stream ciphers, like Correlation attacks [6], Fast Correlation attacks [7], Algebraic attacks[8] and, the Cube attack [9]. The latter, presented in 2009 by Dinur and Shamir, is a very fascinating technique that can be applied virtually to any cipher since its unique requirement is black-box access to the target cipher. The idea is to combine offline exhaustive searches over selected tweakable public/IV bits (the sides of the "cube") with an online key-recovery phase. Cubes computation grows exponentially with the dimension of the cubes, so many approaches have been proposed in the literature to optimize this process [9–12]; although, none of them fully investigates the potentiality of Graphics Processing Units (GPUs). Since the early times of their usage as general purpose computing devices, GPUs have been considered ideal candidates for cryptanalysis tasks as the intensive computations usually required for cryptanalysis can benefit

[1] In mathematics, a periodic function is a function that repeats its values in regular intervals or periods.

of the high level of parallelism and the computational power available on GPUs. Indeed, many works exist in the literature as evidence that they can be successfully used [13–15]. In order to leverage GPUs, it is mandatory to carefully design and implement the applications by taking into account their peculiar characteristics.

In the first part of the present thesis work, the first Cube Attack Framework tailored on GPUs [16] is presented. It has been designed and implemented as an optimized solution that is able to perform all the steps needed by the attack; all the design choices are discussed, detailing their respective advantages and difficulties. The framework can virtually support any cipher as the only requirement is development of a GPU version of the target cipher. It has been successfully validated on two different stream ciphers, Trivium [17] and Grain-128 [18]. Even though the attack against Trivium has been ran with only a few preliminarily sets of Initialization Vector (IV) bits (i.e. the cubes) - specifically selected to both validate the code and compare the obtained results with the literature- the findings improve the state-of-the-art for attacks against reduced-round version of Trivium; here there are presented the first full-key recovery for Trivium up to 781 initialization rounds without brute-force, and the first ever linear equation binding only key bits yields after 800 initialization rounds. Moreover, thanks to the framework, few new candidate linear equations in key bits for both Trivium and Grain-128, respectively after 800 initialization rounds and after 160 initialization rounds, have been discovered; the detailed description and analysis of these results are going to released soon in a work that is currently being finalized. Furthermore, the presented implementation allows for exhaustively assigning values to (subsets of) public variables with negligible additional costs. This approach allows to potentially weakening the assumption of a completely tweakable IV that has been done in previous works [9]. Eventually, the evaluations of the framework on the computational speedup with respect to a CPU-parallel benchmark, the performance dependence on system parameters and GPU architectures (Nvidia Kepler vs Nvidia Pascal), and the scalability of the proposed solution on Multi-GPU systems [19] have been reported. By exhibiting the benefits of a complete GPU-tailored implementations of the cube attack, this thesis work provides novel and strong elements in support of the general feasibility of the attack, thus paving the way for future work in the area.

In the second part of this dissertation, an automatic optimization solution for data-driven applications is presented. This solution has been developed by the canditate while he worked at Dividiti L.t.d., as research intern. A new framework for data-driven adaptive libraries has been designed and developed. It generates a pluggable runtime system based on Machine Learning that is able to choose for the current input the 'optimal' choice, according to a selected metrics, among a set known best choices for other inputs. In order to build the prediction model, the framework relies on the scikit-learn library [20] to generate several Decision Tree Classifiers [21]. Three datasets have been collected on two different architectures, Nvidia Pascal and Arm Mali Midgard and they have been used as training data to build the prediction models. The proposed framework automatically extracts the decision rules from the models generating the corresponding source code for the runtime system. This auto-generated code is then responsible to pick up the configuration according to the model rules. Moreover, a proof of concept based on the open-source OpenCL library CLBlast [22] has been proposed. The proof of concept shows how the proposed framework can be used to optimize matrix multiplication routines for unpredictable matrix sizes. Matrix multiplication is a core routine for many problems ranging from Machine Learning [23], simulation and cryptography. The runtime system generated by the proposed framework provides a speed up respect to the standard CLBlast up to 3× on Nvidia Pascal (Tesla P100) and 2.5× on ARM Midgard (Mali-T860). Moreover the presented framework can generate an optimized runtime system for any target library with a limited effort.

Acknowledgements

My PhD adventure is now approaching to the end and it is now the time for me to say thank to all people that encourage, help and support me along these three years.

First of all, I would like to say thank to my supervisor Massimo Bernaschi; he is a great mentor and I am glad for the opportunity to work with him. His advises, suggestions and his continued support have been precious for me.

I also want to thank my PhD tutor, Prof. Marco Pedicini; his passion for cryptography push me to work on cryptanalysis and his advices have been fundamental for the success of the framework. I would to thank Flavio Lombardi and Stefano Guarino for all their work and their precious support and for helping me in testing the cryptanalysis framework.

I would like also to say thank to Anton Lokhmotov and Grigori Fursin; the internship at Dividiti L.t.d was an invaluable experience where I had the opportunity to work on a very interesting and challenging project.

A special thank to Flavio Vella for supporting and helping me for all the internship and after, and to his family Luisa and Sofia, my "Cambridgeshire family".

I would like also to thank all my colleagues for their support, all the co-authors of my published and ongoing work, the director of the Ph.D programme Prof. Lopez and the anonymous reviewers for their valuable insights and suggestions. In particular, I would like to thank Alessandro Celestini, Alessandro Sabellico, Antonio Di Marco, Cedric Nugteren, Enrico Mastrostefano, Giancarlo Carbone, Giuseppe Totaro, Matteo Lulli, Mauro Bisson, and Vinca Prana.

Last but not least, a special thank to my wife and to my family. To my lovely wife Arianna for always supporting me for all these years, especially in the hardest moments and for always believes in me, to my mother and my father for their unconditionally support along all these years, to my sister, her life-mate Marco and my little wonderful nephew Mattia, and to Carla, Costantino, Francesca, and Gian Michele that always encourage me.

To my lovely grandma,
who always believed in me.

Contents

List of Figures

List of Tables

Acronyms

Acronym	Meaning
ALU	Arithmetic Logic Unit
ANF	Algebraic Normal Form
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
DTPR	Decision Tree Peak Ratio
DTTR	Decision Tree Tune Ratio
FLOPS	Floating Point Operation per Second
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSR	Feedback Shift Register
GB	Giga Byte
GEMM	Generic Matrix Multiplication
GPU	Graphics Processing Unit
HPC	High Performance Computing
IAC	Istituto per le Applicazioni del Calcolo
ISODAC	Indexing and Searching Of Data Against Crime
IV	Initialization Vector
LFSR	Linear Feedback Shift Register
OpenCL	Open Compute Library
OpenMP	Open MultiProcessing
OTP	One Time Pad
RAM	Random Access Memory
SM	Streaming Multiprocessor
TMDTO	Time-Memory-Data-Trade-off

1 Road Map and Contributions

Since the early times of usage as general purpose computing devices, GPUs have been considered ideal candidates for intensive computation tasks such as cryptanalysis, simulations [26] and, more generally, for all tasks that can benefit of the high level of parallelism and computational power provided by GPUs. In the last years, the rise of Machine Learning and Deep Learning frameworks [27–29] and libraries [30, 31] able to transparently use GPUs, breathes new life into the research community as GPUs can offer an appreciable speedup to the algorithms commonly used in Machine Learning and Deep Learning fields. Most of the aforementioned frameworks and libraries provide an abstraction layer through simple APIs that allows users to run their applications on GPUs without requiring any knowledge about their characteristics. However, in order to fully exploit the potentiality of GPUs, an in depth knowledge of them architecture is actually needed. This is especially true for cryptanalysis and data-driven applications; the former require to carefully design and implement the application because the computational power and the amount of memory required usually grow exponentially; the latter may require the implementation of new heuristics able to automatically tune the applications according to the characteristics of the input, that are known only at runtime.

In the present thesis work are reported the two main topics the candidate worked on during his PhD. The first part will focus on his research on cryptography and cryptanalysis. The first Cube Attack Framework on GPU is presented as well as the findings that improve the state-of-the-art cryptanalysis results on round-reduced version of Trivium. Moreover, all design choices are carefully described, and their respective advantages and drawbacks with respect to GPU architecture are discussed.

The second part of the dissertation is focused on data-driven applications. It presents a new framework based on Machine Learning that aims at optimising libraries for data-driven applications. The framework generates a pluggable runtime system that infers the optimal choice, according to the selected metrics, for a given input. A proof of concept focused on optimizing matrix multiplication for unpredictable matrix sizes is presented along with the experiments results used for its evaluation. The work presented in the second part of the present thesis was done by the candidate while he was research intern at *Dividiti L.t.d.*, as part of the *HiPEAC Industrial PhD Mobility Programme*.

Since the two parts of the dissertation address different topics, a dedicated introduction at the beginning of each part is provided. Therefore, this chapter mainly describes the road map of this thesis and it summaries the main contributions the candidate provides to both topics during his PhD. Furthermore, the candidate's contributions to other topics not covered by the present thesis work, are hereunder briefly described.

1.1 Road Map

- **Chapter 2** introduces the cryptanalysis Cube Attack framework and describes all the background needed to understand the attack and its components.
- **Chapter 3** provides a detailed analysis of the cube attack from a practical point of view, remarking the advantages and the drawbacks of GPUs for this kind of attack. Furthermore, it thoroughly describes the design and implementation choices made and it reports and discusses the results of the performance analysis experiments.
- **Chapter 4** concludes the first part on cryptanalysis; it reports and examines the findings that extend the state-of-the-art against round-reduced Trivium. Moreover, it describes some preliminarily results against round-reduced version of Grain-128.
- **Chapter 5** introduces the problem of optimizing data-driven applications and it provides the background needed in the next chapters.
- The design and implementation choices of the proposed framework are analyzed and discussed in **Chapter 6**.
- **Chapter 7** describes the proof of concept focus on Matrix Multiplication for unpredictable matrix sizes. Furthermore, it describes and analyzes the experiments results on two different architectures, Nvidia Pascal and ARM Mali Midgard for all the models generated by the proposed framework.
- Finally **Chapter 8** summarizes and discusses the main contributions as well as the novelties and the future works for both the topics.

1.2 Contributions

Most of the content of this dissertation is extracted from some of published (or submitted) works of the candidate; this thesis mainly collects and harmonises them. In Section 1.4, it is reported a brief description about the topics the candidate worked on during his first year of PhD that are not strictly related to the topics covered in this dissertation.

Part I

In Chapter 3, the candidate thoroughly analyzes and discusses the cube attack advantages and drawbacks from a GPU perspective. Furthermore, he detailing describes the design and implementation choices aimed at fully exploiting the high level of parallelism and the computational power of GPUs. The proposed implementation, that is to the best of his knowledge the first completely tailored on GPUs, allows to improve the state-of-the-art against reduced-round versions of Trivium, yielding full key recovery up to 781 initialization rounds without brute-force and the first ever maxterm after 800 initialization rounds [16]. Moreover the propose implementation allows for exhaustively assigning values to (subsets of) controlled public variables with negligible additional costs. A detailed discussion about the cryptanalysis results is provided in Chapter 4.

The candidate proposes a flexible multi-GPU framework, validated by using previous results in literature, that virtually supports any cipher and he provides the first GPU tailored implementation of the two target cipher, Trivium and Grain-128. Furthermore, the candidate provides also a thoroughly performance analysis in terms of speedup and scalability [19].

By exhibiting the benefits of a complete GPU-tailored implementation of the cube attack, the candidate provides novel and strong elements in support of the general feasibility of the attack, thus paving the way for future work in the area.

Part II

In Chapter 6 the candidate describes the requirements and architectural design of a new framework for generation of data-driven runtime. He proposes a new framework based on predictive model to select the optimal algorithm and the related tuned parameters in order to optimize data-driven applications. The framework is able to generate a runtime representing the predictive model that can be plugged on top of existing libraries.

In Chapter 7, the candidate presents a proof of concept for matrix multiplication based on generated runtime. Moreover, in the proof of concept, the candidate extends some functionalities of CLBlast, an OpenCL BLAS library. The proof of concept has been tested on two different architectures, Nvidia Pascal and ARM Mali Midgard.

Finally, the candidates describes and analyzes all the experiments on both the architectures. The experiments aim at evaluating both the theoretical and experimentally verified quality of several models as well as the overhead of the runtime generated by the proposed framework.

The results of the experiments shows a performance speedup up to 3× on Nvidia Pascal and up to 2.5× on ARM Mali Midgard, while the overhead in the worst case impacts less than 2% on performance.

1.3 List of contributions

These are the works accepted or submitted that are relevant to the topics covered in this thesis:

1. Marco Cianfriglia, Stefano Guarino, Massimo Bernaschi, Flavio Lombardi and Marco Pedicini. *A Novel GPU-Based Implementation of the Cube Attack. Preliminary Results Against Trivium.* In International Conference on Applied Cryptography and Network Security (ACNS 2017) [16];
2. M. Cianfriglia and S. Guarino. *Cryptanalysis on GPUs with the Cube Attack: Design, optimization and performance gains*. In 2017 International Conference on High Performance Computing Simulation (HPCS 2017) [19].

In the following the list of my other works that are currently being finalized:

- M. Cianfriglia, S. Guarino, M. Bernaschi, F. Lombardi, M. Pedicini - *A Cube Attack Framework on GPUs.* To be submitted at *Journal of Cryptographic Engineering*;
- M. Cianfriglia, F. Vella, C. Nugteren, M. Bernaschi, G. Fursin, A. Lokhmotov - *Next generation adaptive libraries for emerging data-driven applications.* To be submitted at *The International Conference for High Performance Computing, Networking, Storage, and Analysis - Supercomputing 2018*;

1.4 Other contributions

As part of his collaboration as research associate with the *Istituto per le Applicazioni del Calcolo (IAC) Mauro Picone - CNR*, the candidate contributed to the European project ISODAC (Indexing and Search Of Data Against Crimes). The goal of the project was the design and the develop an high performance solution for indexing and searching heterogeneous data. The last IBM estimation on Big Data says that every day 2.5 quintillion bytes of data are created [32], and this trend seems it is not going to stop. In this scenario, it is fundamental to have solutions that provide an efficient way to search on huge amount of unstructured data. Searching for words or sentences within large sets of textual documents can be very challenging unless an index of the data has been created in advance. However, indexing can be very time consuming especially if the text is not readily available and has to be extracted from files stored in different formats. Several solution based on the MapReduce paradigm, have been proposed to accelerate the process of index creation. These solutions perform well when data are already distributed across the hosts involved in the elaboration. On the other hand, the cost of distributing data can introduce noticeable overhead. In order to provide a solution for this problem, a new approach aimed at improving efficiency without sacrificing reliability has been proposed, ISODAC [33]. The proposed solution reduces to the bare minimum the number of I/O operations by using a stream of in-memory operations to extract and index text. ISODAC relies on GPUs to further improve the performance for the most computationally intensive tasks of the indexing procedure. It indexes heterogeneous documents up to 10.6× faster than other widely adopted solutions, such as Apache Spark [34]. As proof of concept, a tool to index forensic disk images has been developed. It provides a web interface that allows investigators to easily start indexing process and to submit queries the indexes that are already available.

- G. Totaro, M. Bernaschi, G. Carbone, M. Cianfriglia and A. Di Marco *ISODAC: A high performance solution for indexing and searching heterogeneous data.* In Journal of Systems and Software [33].

2 A new Cryptanalysis Framework on GPUs

2.1 Introduction

The security of a stream cipher relies on its ability to mimic the properties of the perfectly secure One Time Pad (OTP): predicting future keystream bits (*e.g.*, by recovering its inner state) must be computationally infeasible. As a matter of fact, as highlighted by *algebraic* [8] and *correlation* attacks [6], any statistical correlation between output bits and linear combinations of input bits is a potential security breach for the cipher. Cryptographers are therefore caught in between implementation requirements, which suggest the use of efficient primitives such as *Feedback Shift Registers* (FSRs) or *Finite State Machines* (FSMs), and security requirements, which demand for solutions able to disguise the dependence of keystream-bits on the inner state of the registers. Many recent stream ciphers therefore rely upon irregular clocks, mutual clock control, non-linear and/or mutual feedback among different registers, or combinations of these solutions. The cube attack, proposed by Dinur and Shamir [9], can be classified as an algebraic known-plaintext attack in which linear equations binding key bits are extracted through exhaustive searches over selected public/*IV* bits – the edges of the *cubes* the attack is named after. The success of the attack depends on its ability to detect imbalances into the distribution of monomials in the polynomial representation of the target cipher. Since these statistical flaws are generally unknown beforehand, the attack is often run without a clear prior insight into a convenient strategy for selecting the cubes – an approach made possible by the fact that the attack only requires black-box access to the attacked cipher. Possible practical strategies include exploring cubes of different (possibly large) size, trying many different sets of indices, and varying the binary assignment of the public bits not belonging to the tested cube, all solutions that come at an exponential cost. As in *Time-Memory-Data Trade-Off* (TMDTO) attacks [35], the extensiveness of the pre-computation stage, which ultimately determines the attack's success rate, must be carefully tuned on the available computing power, memory, and data.

While previous CPU-based approaches seem to pursuit a balanced use of time and memory, we develop an implementation of the cube attack that fully leverages the potential of Graphics Processing Units (GPUs) to boost the parallel search for suitable cubes.

The contributions of our approach are briefly summarized hereunder :

- We show how to tune the design and implementation of the cube attack to the characteristics of GPUs, in order to fully exploit parallelization while coping with limited memory [16, 19].
- We present a flexible framework to mount cube attacks against any cipher, under the sole condition that the cipher is also implemented in GPU. The tool is independent of the GPU architecture, and it supports extension to multi-GPU systems.
- We improve the state-of-the-art against reduced-round versions of Trivium, yielding full key recovery up to 781 initialization rounds without bruteforce and the first ever maxterm after 800 initialization rounds [16]. Moreover, we perform some other experiments to further explore Trivium after 800 initialization rounds and we discover few maxterm candidates up to 830 rounds. We are currently verifying these new results, and we are going to release them in a work being now finalized.
- Our implementation allows for exhaustively assigning values to (subsets of) public variables with negligible additional costs. This means extending the quest for superpolys to a dimension never explored in previous works, and, by not being tied to a very small set of IV combinations, potentially weakening one of the basic requirements of the cube attack, that is, the assumption of a completely tweakable IV.
- We carefully analyse the performance of our implementation, in terms of: (i) speedup with respect to a CPU implementation, (ii) dependence on system parameters, (iii) comparison among different architectures (including latest generation GPU cards), and (iv) impact of a multi-GPU distributed approach [19].
- We provide the first GPU tailored implementation, to the best of our knowledge, for Trivium and Grain-128. We validated our framework on both the ciphers by extracting the symbolical representation of the polynomial corresponding on round-reduced versions of the ciphers. We then ran the attacks on some selected cubes, specifically selected from the symbolical representations and we verified the consistency of the results.

2.2 Background

2.2.1 The Cube Attack

The cube attack is a widely applicable method of cryptanalysis introduced by Dinur and Shamir [9], based on a construction similar to Vielhaber's AIDA [36]. The underlying idea, object of extensions (*dynamic* cube attacks [37, 38], *cube testers* [39, 40]) and generalizations [41, 42], is to extract from the unknown polynomial representation of the target cipher a set of linear (or low-degree) equations binding key variables, replacing a symbolic factorization with an exhaustive evaluation over selected public variables.

Let $E(\mathbf{x}, \mathbf{y})$ denote the target cipher, as a function of two vectors: the n public variables $\mathbf{x}$ (the *IV*) and the k private variables $\mathbf{y}$ (the key K). A generic bit keystream z can be expressed as $z = p(\mathbf{x}, \mathbf{y})$, where p is the polynomial representation of E, and all variables appear in p with degree 1, at most. The idea of the attack is to choose a subset of m public variables $\mathbf{x}_I \subset \mathbf{x}$ indexed by $I \subset \{1, \ldots, n\}$ and focus on the quotient $p_{S(I)}(\mathbf{x}_{\bar{I}}, \mathbf{y})$ of the division of $p(\mathbf{x}, \mathbf{y})$ by the monomial $t_I = \prod_{x \in \mathbf{x}_I} x$. By definition, $p_{S(I)}(\mathbf{x}_{\bar{I}}, \mathbf{y})$, called the *superpoly* of I in p, only depends on public variables $\mathbf{x}_{\bar{I}}$ indexed by I's complement $\bar{I}$ (other than $\mathbf{y}$). If we find $p_{S(I)}(\mathbf{x}_{\bar{I}}, \mathbf{y})$ and assign any value $\mathbf{v}_{\bar{I}}$ to $\mathbf{x}_{\bar{I}}$, we obtain a polynomial $p_{S(I)}(\mathbf{v}_{\bar{I}}, \mathbf{y}) = p_{S(I)}(\mathbf{y})$ binding only key variables[1]. If I is such that $p_{S(I)}(\mathbf{y})$ is *linear*, the monomial t_I is called a *maxterm for p with the assignment* $\mathbf{v}_{\bar{I}}$. If we can identify maxterms and find the symbolic expression of their superpolys, we obtain a system of linear equations that can be used to recover the secret key.

Unfortunately, we can not find $p_{S(I)}(\mathbf{x}_{\bar{I}}, \mathbf{y})$ symbolically because we do not know $p(\mathbf{x}, \mathbf{y})$ in the first place. To make up for it, we observe that all monomials in $p_{S(I)}(\mathbf{x}_{\bar{I}}, \mathbf{y})$ do not contain any of the variables $\mathbf{x}_I$, whereas all monomials in the remainder $q(\mathbf{x}, \mathbf{y})$ of the division of $p(\mathbf{x}, \mathbf{y})$ by t_I do not contain *at least* one of the variables in $\mathbf{x}_I$. For this reason, if $C_I(\mathbf{v}_{\bar{I}})$ denotes the *cube* composed by all 2^m possible binary assignments to $\mathbf{x}$ conditioned to $\mathbf{x}_{\bar{I}} = \mathbf{v}_{\bar{I}}$, the sum of $p(\mathbf{x}, \mathbf{y})$ over $C_I(\mathbf{v}_{\bar{I}})$ yields [9]

$$\sum_{\mathbf{v} \in C_I(\mathbf{v}_{\bar{I}})} p(\mathbf{v}, \mathbf{y}) = p_{S(I)}(\mathbf{v}_{\bar{I}}, \mathbf{y}) = p_{S(I)}(\mathbf{y}) \tag{2.1}$$

regardless of the values assigned to $\mathbf{y}$. In other words, we can find $p_{S(I)}(\mathbf{y})$ through an exhaustive sum over $\mathbf{x}_I$.

The following example aims at clarifying the notation.

Example 1. Let $n = 3$, $k = 1$, and

$$p(x_1, x_2, x_3, y_1) = x_1x_2y_1 + x_1x_3 + x_1x_2 + x_2y_1 + x_3y_1 + 1$$

[1] The standard assumption is $\mathbf{v}_{\bar{I}} = \mathbf{0}$, but this is not actually required.

If $I = \{1, 2\}$ (*i.e.*, $\mathbf{x}_I = \{x_1, x_2\}$, $\mathbf{x}_{\bar{I}} = \{x_3\}$), we have

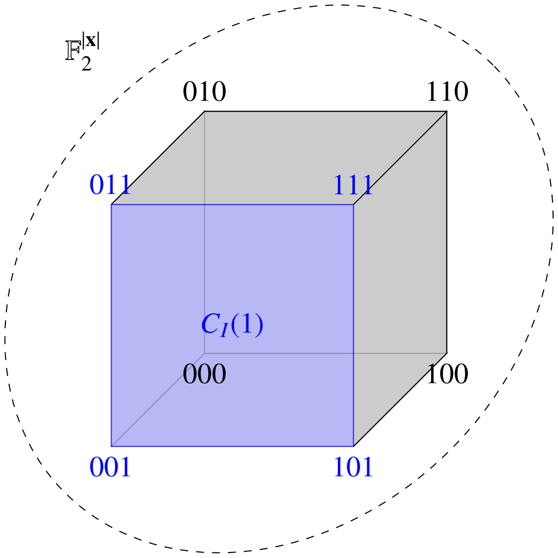
$$p(x_1, x_2, x_3, y_1) = \underbrace{x_1 x_2}_{t_I} \underbrace{(y_1 + 1)}_{p_{S(I)}} + \underbrace{(x_1 x_3 + x_2 y_1 + x_3 y_1 + 1)}_{q(x_1,x_2,x_3,y_1)}$$

If $x_3 = 1$ (*i.e.*, $\mathbf{v}_{\bar{I}} = \{1\}$), summing p over the cube $C_I(\mathbf{v}_{\bar{I}}) = \{001, 011, 101, 111\}$, we have:

$$\sum_{\mathbf{v} \in C_I(\mathbf{v}_{\bar{I}})} p(\mathbf{v}, \mathbf{y}) = \underbrace{y_1 + 1}_{p(001,y_1)} + \underbrace{y_1 + y_1 + 1}_{p(011,y_1)} + \underbrace{1 + y_1 + 1}_{p(101,y_1)} + \\ + \underbrace{y_1 + 1 + 1 + y_1 + y_1 + 1}_{p(111,y_1)} = y_1 + 1$$

that is exactly $p_{S(I)}(y_1)$.

In Figure 2.1, the representation of the cube described in Example 1.



The blue vertices define the cube $C_I(\mathbf{v}_{\bar{I}})$.

FIGURE 2.1: A toy example of *cube*.

2.2.2 Stream Cipher

Symmetric key ciphers can be divided in block ciphers and stream ciphers [43]. The formers apply the encryption function at the same time and with the same key to an entire block of plaintext bits; this means that each bit of the ciphertext depends on all the bits of the corresponding plaintext block. On the contrary, stream ciphers encrypt the bits one-by-one; they generate a keystream starting from a key K and Initialization Vector (IV). Each bit of the ciphertext is typically obtained by adding a single bit of the keystream to a single bit of plaintext. There exist synchronous ciphers where the keystream depends only on K and IV while asynchronous ciphers use also the generated ciphertext. Stream ciphers are typically used in all the scenarios where resources are limited and speed is critical, as they tend to be small and fast. As example, they are used to guarantee confidentiality and integrity of mobile communications [3, 4, 44]. They are generally built upon a cryptographic primitive called Linear Feedback Shift Register (LFSR) along with some other components, like Finite State Machines, in order to introduce non-linearity; sometimes it is possible to use just LFSRs by properly combine their output through a non-linear equation.

A shift register is a cascade of cells, sharing the same clock, in which the content of each cell is transferred into the next cell at each clock tic, resulting in a circuit that shifts by one position the array stored in it. The content of the register is called its *state*, and the initial state of the register is called the *seed*. The output of the register is the content of the first cell, whereas its input is the new content that fills the last cell after the clock strikes. If the input of the register is a linear function of its state, the registers is called a Linear Feedback Shift Register (LFSR). In cryptographic applications, LFSRs are used to produce pseudo-random streams from a secret key. The key is used to initialize the register, *i.e.*, to define its seed, and the LFSR is designed to produce an output stream that looks as random as possible.

Since the operation of a LFSR is deterministic, the seed completely determines the whole stream of values produced by the register. In general, knowing the state of the register at any time t allows to predict the output of the register at any time $t' \geq t$, based on the feedback relation. If the register is composed of l cells, denoting the state of the register at time t as the l-tuple $(s_0^t, s_1^t, \ldots, s_{l-1}^t) \in \mathbb{F}_q^l$, the behaviour of a typical LFSR can be described as

$$\begin{cases} s_i^{t+1} &= s_{i+1}^t \quad \text{for all } i = 0, \ldots, l-2 \\ s_{l-1}^{t+1} &= \sum_{i=1}^{l} \alpha_i s_{l-i}^t \end{cases}$$

where $\alpha_i \in \mathbb{F}_q$ for all i.[2] The feedback relation can be described by the so-called *feedback polynomial*, defined as

$$P(X) = 1 - \sum_{i=1}^{l} \alpha_i X^i$$

As the register has a finite number of possible states, it must eventually enter a repeating cycle. More precisely, the maximum possible number of states is q^l and the register enters a cycle as soon as its state equals the seed. Since the all-zero state $(0, 0, \ldots, 0) \in \mathbb{F}_q^l$ is an absorbing state (*i.e.*, once the register enters that state it never exits it), the best possible scenario is a register that crosses all $q^l - 1$ possible non-zero states before returning in its initial state. It can be proved that a LFSR has a maximum period $q^l - 1$ if and only if its feedback polynomial is *primitive*.

[2] The sum notation $\sum_{i=1}^{l}$ can practically correspond to a bitwise xor, $\bigoplus_{i=1}^{l}$, or to a sum modulo q, $\boxplus_{i=1}^{l}$.

The period of an LFSR plays an important role in terms of security. As we mentioned before, the Stream Cipher tries to mimic the OTP by generating a pseudorandom key stream that is xored with the plaintext; if we use the same keystream more than once, it is trivial to break the cipher.

2.3 Target Ciphers Specifications

In the following, we briefly describe the two ciphers we selected as test case targets for our attack.

2.3.1 Trivium

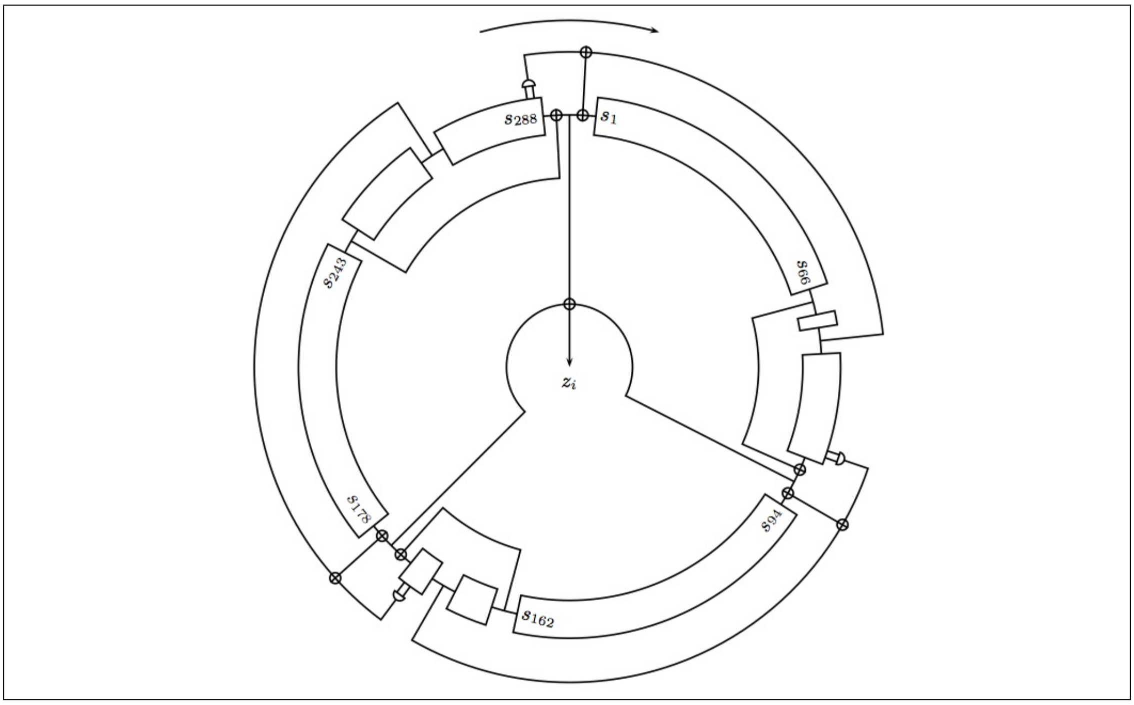


FIGURE 2.2: Trivium Cipher [24].

Trivium [17] is a synchronous stream cipher conceived by Christophe De Cannière and Bart Preneel, not patented, and specified as an International Standard under ISO/IEC 29192-3. It is part of the eSTREAM portfolio [45]. Trivium combines a flexible trade-off between speed and gate count in hardware, and a reasonably efficient software implementation. Quoting [24]: “Trivium is a hardware oriented design focused on flexibility. It aims to be compact in environments with restrictions on the gate count, power-efficient on platforms with limited power resources, and fast in applications that require high-speed encryption”. Particularly interesting is the fact that any state bit stays unused for at least 64 iterations after it has been modified. This means that up to 64 iterations can be parallelized and computed at once, allowing for a factor 64 reduction in the clock frequency without affecting the throughput.

Trivium generates up to 2^{64} bits of output from an 80-bit key $K = \{x_1, ..., x_{80}\}$ and an 80-bit Initial Vector $IV = \{v_1, ..., v_{80}\}$, and it shows remarkable resistance to cryptanalysis despite its simplicity and its excellent performance. It is composed by a 288-bit internal state $s_1, ..., s_{288}$ consisting of three shift registers R1, R2 and R3 of length 93, 84 and 111, respectively. The feedback to each of these registers and the output bit of the cipher are obtained through non-linear combinations involving in total 15 out of the 288 internal state bits (see Figure 2.2). During the initialization phase the key bits filled the first 80 bits of R1 whereas the IV bit fill the first 80 bits of R2; all the remaining unfilled bits of R1, R2, and R3 are filled with 0 except for the last three bits of R3 that are filled with 1. The three registers are updated simultaneously using the same rules in both initialization and keystream generation modes; the only difference is that the output z is produced only in keystream mode. The update function is defined as follow where + denotes the bit XOR operation and the · represents the logical AND:

$$
\begin{gathered}
t_1 \leftarrow s_{66} + s_{93} \\
t_2 \leftarrow s_{162} + s_{177} \\
t_3 \leftarrow s_{243} + s_{288} \\
z \leftarrow t_1 + t_2 + t_3 \\
t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171} \\
t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264} \\
t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69} \\
(s_1, ..., s_{93}) \leftarrow (t_3, s_1, ..., s_{92}) \\
(s_{94}, ..., s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176}) \\
(s_{178}, ..., s_{288}) \leftarrow (t_2, s_{178}, ..., s_{287}).
\end{gathered}
$$

The initialization phase involves 1152 rounds and it guarantees that the output begins to be produced only after all key-bits and IV-bits have been sufficiently mixed together to define the internal state of the registers.

2.3.2 Grain-128

Grain-128 is a variant of Grain v1 [46], a stream cipher belonging to eSTREAM portfolio, proposed by Hell, Johansson, Maximov and Meier [18]. It is very compact and easy to implement, especially in hardware as stated by the authors. It supports 128 bit keys and 96 bits IV. It is composed by a Linear Feedback Shift Register combined with a Non-linear Feedback Shift Register (NFSR) and a boolean function $h(x)$. Both the LFSR and the NFSR have 128 bits and altogether represent the state of the cipher. They are updated by two different feedback polynomials, respectively $f(x)$ and $g(x)$. The update functions corresponding to $f(x)$ and $g(x)$ are provided below, where $s_i, s_{i+1}, ..., s_{i+127}$ denote the LFSR states and $b_i, b_{i+1}, ..., b_{i+127}$ the NFSR states:

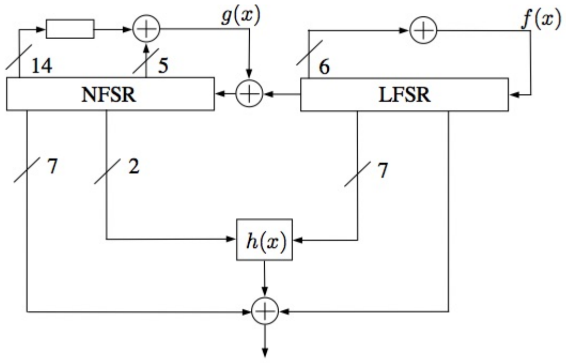


FIGURE 2.3: Grain-128 [18].

$$s_{i+128} = s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96}$$

$$b_{i+128} = s_i + b_i + b_{i_26} + b_{i+56} + b_{i-91} + b_{i+96} + b_{i+3}bi + 67 + bi + 11b_{i+13} + b_{i+17}b_{i+18} + \\ b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + b_{i+68}b_{i+84}.$$

Two states from the NFSR and seven from LFSR define the nine input variables $x_0, ..., x_8$ of the boolean function $h(x)$, corresponding to b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+60} and s_{i+95} respectively, hereunder reported:

$$h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8.$$

The output function takes as input the output of $h(x)$ added to s_{i+93} and b_{i+j} where $j \in A = \{2, 15, 36, 45, 64, 73, 89\}$.

In the initialization phase, the first 96 bits of the LFSR are filled with the IV bits whereas the remaining 32 bits are filled with 1. On the other hand, the key bits are used to completely fill the NFSR. Grain-128 defines 256 initialization rounds where the output of $h(x)$ is used as feedback both to NFSR and LFSR.

In Figure 2.3, the overview of the cipher in keystream mode is reported. A very interesting feature of Grain-128 is that the last 31 bits of both the NFSR and the LFSR are not used in the update function, and this allows to easily speedup by a factor of 32.

2.4 Related Work

The cube attack is a widely applicable method of cryptanalysis introduced by Dinur and Shamir [9]. The underlying idea, similar to Vielhaber's AIDA [36], can be extended, *e.g.*, by assigning a *dynamic* value to *IV* bits not belonging to the tested cube [37, 38], or by replacing cubes with generic subspaces of the IV space [41]. It is used in the so-called *cube testers* to detect non-random behaviour rather than performing key extraction [39, 40]. Despite the cube attack and its variants have shown promising results against several ciphers (*e.g.*, Trivium [9], Grain [37], Hummingbird-2 [47], Katan and Simon [38], Quavium [48]), Bernstein [49] expressed harsh criticism to the feasibility and convenience of cube attacks. Indeed, a general trend for cube attacks is to focus on reduced-round variants of a cipher, without any evidence that the full version can be equally attacked. However, while Bernstein suggests that the cube attack only works if the ANF of the cipher has low degree, Fouque and Vannet [10] argue (and, to some extent, experimentally show) that effective cube attacks can be carried out not aiming at the maximum degree of the ANF, but rather exploiting a nonrandom ANF by searching for maxterms of significantly lower degree. Along this line, O'Neil [50] suggests that even the full version of Trivium exhibits limited randomness, thus indicating the potential vulnerability of this cipher to cube attacks.

In recent years, several implementations of the cube attack attempted at breaking Trivium, one of the target ciphers described in Section 2.3.1. Quedenfeld et al. [51] found cubes for Trivium up to round 446. Srinivasan [52] introduced a sufficient condition for testing a superpoly for linearity in $\mathbb{F}_2$ with a time complexity $O(2^{c+1}(k^2 + k))$, yielding 69 extremely sparse linearly independent superpolys for Trivium reduced to 576 rounds. In their seminal paper [9], Dinur and Shamir found 63, 53, and 35 linearly independent superpolys after, respectively, 672, 735, and 767 rounds. Fouque and Vannet [10] even improve over Dinur and Shamir, by obtaining 42 linearly independent superpolys after 784 rounds, and 12 linearly independent superpolys (plus 6 quadratic superpolys) after 799 rounds. To the best of our knowledge, these are the best results against Trivium to date, making our attack comparable to (or better than) the state-of-the-art.

Several attacks have been proposed against Grain-128 and the Grain Family [53]. In particular, Dinur and Shamir presented the *dynamic cube attacks* in [37] where they show an attack on a full version of Grain-128 that is able to recover the full key when it belongs to a subset of the all possible keys. Moreover, in [11] the authors present a distinguished attack using another variant of cube attacks called *cube testers* where they rely on Field Programmable Gate Array (FPGA). Another interesting work that uses FPGA is [12], where the authors built a dedicated FPGA-based hardware to show the benefits of using highly parallelized hardware for cryptanalytic tasks. However, to the best of our knowledge, there are not previous attempts against Grain-128 that use the classical cube attack. We ran only few experiments on round-reduced versions in order to validate the framework with it and to evaluate the performance of the attack on a different cipher. Moreover these experiments allow us to discover few maxterm candidates when the number of initialization rounds is 160. We are going to better describe and analyze these finding on a work currently being finalized.

Distributed computing and/or parallel processing have been explored in the literature to render attacks to crypto systems computationally or storage-wise feasible/practical. Smart et al. [54] developed a new methodology to assess cryptographic key strength using cloud computing. Marks et al. [55] provided numerical evidence of the potential of mixed GPU(AMD, Nvidia) & CPU technology to data encryption and decryption algorithms. Focusing on GPU, Milo et al [56] leverage GPUs to quickly test passphrases used to protect private keyrings of OpenPGP cryptosystems, showing that the time complexity of the attack can be reduced up to three-orders of magnitude with respect to a standard procedure, and up to ten times with respect to a highly tuned CPU implementation. A relevant result is obtained by Agostini [57] leveraging GPUs to speed up Dictionary Attacks to the BitLocker technology commonly used in Windows OSes to encrypt disks. Finally, and most closely related to the present work, Fan and Gong [47] made use of GPUs to perform side channel cube attacks on Hummingbird-2. They describe an efficient term-by-term quadraticity test for extracting simple quadratic equations, leveraging the cube attack. Just like us, Fan and Gong speed-up the implementation of the proposed term-by-term quadraticity test by leveraging GPUs and finally recovering 48 out of 128 key bits of the Hummingbird-2 with a data complexity of about 2^{18} chosen plaintexts. However, we present a complete implementation of the cube attack thoroughly designed and optimized for GPUs. Our flexible construction allows an exhaustive exploration of subsets of IV bits, thus overcoming the limitations of dynamic cube attacks, which try to find the most suitable assignment to those bits by analyzing the target cipher.

3 Cube Attack on GPUs

In this section, we present, detail, and discuss our attack, designed to run on a cluster equipped with Nvidia Graphics Processing Units (GPUs). As previously mentioned, the success of a cube attack is highly dependent on suitable implementation choices. In order to better explain our own approach, we start with an analysis of the cube attack from a more technical perspective. We do not provide a detailing description of GPU characteristics here, interested readers can find a brief description and some references in Appendix B.

3.1 Practical Cube Attack

At a high level, any practical implementation of the cube attack requires performing the following steps:

Choosing a candidate maxterm t_I Since the complexity of evaluating cube C_I scales exponentially with $|I|$, finding a convenient strategy to select the index set I (or, equivalently, the candidate maxterm t_I), is of primary importance. A common assumption is the existence of a threshold degree, cutting apart monomials that yield a nonlinear superpoly from monomials that yield a constant one. Even assuming that the degree of most maxterms lies around some threshold, the lack of information about the distribution of monomials in p prevents any prior educated guess at both the value of the threshold and the actual selection of variables in t_I. In the literature, a few approaches have been proposed to try to address both issues at once. Dinur and Shamir [9] proposed a *random walk* over index sets, starting from a random set I and iteratively updating I adding or subtracting random elements according to the experimentally tested degree of the superpoly $p_{S(I)}$.

Alternatively, Fouque and Vannet [10] used the *Moebius transform* to concurrently compute the sums over all subcubes of a maximal cube $C_{I_{\max}}$ characterized by having the variables $\mathbf{x}_{\bar{I}}$ set to $\mathbf{0}$. Additionally, they suggested a heuristic to identify cubes expected to behave better than others, and used it to select the most promising maximal set $I_{\max}$. However, none of these two strategies is suitable for GPUs, as we will show in Section 3.3.

Testing $p_{S(I)}$ for linearity In principle, assessing if t_I is a maxterm requires finding the symbolic expression of $p_{S(I)}(\mathbf{v}_{\bar{I}}, \mathbf{y})$, for all possible assignments $\mathbf{v}_{\bar{I}}$ to variables $\mathbf{x}_{\bar{I}}$. However, efficient *probabilistic* linearity tests [58, 59] can be safely used in practice. Additionally, aiming at minimizing the degree of $p_{S(I)}(\mathbf{v}_{\bar{I}}, \mathbf{y})$, only $p_{S(I)}(\mathbf{0}, \mathbf{y})$ is usually considered in the literature. We argue that this is not necessarily the best choice, as motivated by our results presented in Section 4.1. In any case, at this stage we can omit the dependence on $\mathbf{v}_{\bar{I}}$ and assume that $p_{S(I)}(\mathbf{y})$ only depends on $\mathbf{y}$. Probabilistic tests involve verifying if

$$p_{S(I)}(\mathbf{u}_1 + \mathbf{u}_2) = p_{S(I)}(\mathbf{u}_1) + p_{S(I)}(\mathbf{u}_2) + p_{S(I)}(\mathbf{0}) \tag{3.1}$$

holds for random pairs of vectors $\mathbf{u}_1, \mathbf{u}_2$. In fact, (3.1) must be true for all $\mathbf{u}_1, \mathbf{u}_2$ if $p_{S(I)}$ is linear, whereas, in general, it holds with probability $\frac{1}{2}$. Practically, (3.1) means

$$\sum_{\mathbf{v}\in C_I} E(\mathbf{v}, \mathbf{u}_1 + \mathbf{u}_2) = \sum_{\mathbf{v}\in C_I} E(\mathbf{v}, \mathbf{u}_1) + \sum_{\mathbf{v}\in C_I} E(\mathbf{v}, \mathbf{u}_2) + \sum_{\mathbf{v}\in C_I} E(\mathbf{v}, \mathbf{0})$$

thus requiring four *numerical* sums.

Finding linear equations Each maxterm t_I yields a linear equation

$$p_{S(I)}(\mathbf{y}) = \sum_{\mathbf{v}\in C_I} E(\mathbf{v}, K) \tag{3.2}$$

whose left side is a linear combination of the key variables $\mathbf{y}$ with coefficients found *offline*, whereas the right side is a number found *online*, and whose solution is the sought unknown assignment of the key K to $\mathbf{y}$. Finding the right side of (3.2) involves a single sum, assuming the availability of the 2^m keystream bits produced with K assigned to $\mathbf{y}$, as $\mathbf{x}$ takes all possible assignments $\mathbf{v} \in C_I$. Finding the *symbolic* expression on the left side of (3.2) instead requires $k + 1$ sums: the free term of $p_{S(I)}(\mathbf{y})$ is

$$p_{S(I)}(\mathbf{0}) = \sum_{\mathbf{v}\in C_I} E(\mathbf{v}, \mathbf{0})$$

whereas the coefficient of each variable y_i is

$$p_{S(I)}(\mathbf{e}_i) + p_{S(I)}(\mathbf{0}) = \sum_{\mathbf{v}\in C_I} E(\mathbf{v}, \mathbf{e}_i) + \sum_{\mathbf{v}\in C_I} E(\mathbf{v}, \mathbf{0})$$

where $\mathbf{e}_i$ is the unit vector with all null coordinates except $y_i = 1$.

Solving the linear system Finally, once a set of linear superpolys have been found, we need to solve the obtained linear system. This can be achieved with any suitable technique described in the literature.

3.2 The Setting

Generally speaking, GPUs are processing units characterized by the following advantages and limitations:

Computing Each unit features a large number (*i.e, thousands*) of simple cores, that make possible running a much higher number of parallel threads compared to a standard CPU. More precisely, the GPU's basic processing unit is the *warp* consisting of 32 threads each. Threads are designed to work on 32-bit words, and the performance is maximized if all threads belonging to the same warp execute exactly the same operations at the same time on different but contiguous data.

Memory The so-called *global* memory available on a GPU is limited, typically between 4 and 16 GB. Each thread can independently access data (random access is fully supported, but costly performance-wise). However, when threads in a warp access consecutive 32-bit words, the cost is equivalent to a single memory operation. Concurrent readings and writings by different threads to the same resources, which require some level of synchronization, should be avoided to prevent serialization that defeats parallelism.

The basic step of the attack is the sum of $E(\mathbf{v}, \mathbf{y})$ over all elements $\mathbf{v}$ of a cube C_I. Each time we sum over a cube, the key variables $\mathbf{y}$ are fixed, either to a random $\mathbf{u}_j$ for the linearity tests, or to $\mathbf{0}$ and to versors $\mathbf{e}_i$ for determining the superpoly. In both cases exactly the same sum $\sum_{\mathbf{v}\in C_I} E(\mathbf{v}, \mathbf{u}_j)$ must be performed for all elements of a set of keys $\{\mathbf{u}_1, \ldots, \mathbf{u}_M\}$.

We define the following strategy for carrying out the sums over a cube with the goal of maximizing the parallelization and fully exploiting at its best the computational power offered by GPUs:

- Assigning to all the threads within a warp the computation of the same cube C_I but with a different key $\mathbf{u}_j$. This choice guarantees that all threads perform the same operation at the same time for the entire computation.

- Leveraging the GPU computing power to calculate all the elements of a cube C_I, providing to the threads just a bit-mask representing the set I. With this approach we can exploit all available GPU memory to store the cubes evaluations and minimize, at the same time, the number of memory access operations.

- Defining a keystream generator function $E(\mathbf{x}, \mathbf{y})$ which outputs a 32-bit word, and letting each thread work on the whole word, fully leveraging the GPU computing model. This approach offers two remarkable benefits: (i) considering 32 keystream bits altogether is equivalent to concurrently attacking 32 different polynomials, and (ii) working on 32-bit integers fits much better with the GPUs features, whereas forcing the threads to work on single bits would critically affect the performance of the attack. As a drawback, attacking 32 keystream bits altogether increases (of a factor 32) the memory needed for storing the cubes' evaluation, thus imposing some limitations on the size of the cubes to be tested, as we will clarify later.

- Choosing the number M of keys to be a multiple of the warp size in order to perform the probabilistic linearity test on 32 keystream bits at the same time and for all M keys.

3.3 The Attack

The practical review of the cube attack presented in Section 3.1 highlighted that the attack requires, for each cube, a number of calls to the target cipher E that grows exponentially, followed by as many sums of the resulting outputs. In Section 3.2 we identified a few tips to tailor the computation of a cube so as to unleash the potential of GPUs. However, the proposed strategy prescribes considering altogether all M different keys per cube needed to run the linearity test. This means that, if T denoted the available memory and $|T|$ its size, the amount of usable memory is *de facto* reduced to $|T|/M$, thus making even more strict the already severe memory constraints characterizing GPUs.

In CPU-based cube attacks, main memory is mostly used to store single evaluations of the cipher over the vertices of some maximal cube $C_{I_{\max}}$. In [9], these evaluations are used to perform a *random walk* that, starting from a random subset $I \subset I_{\max}$, iteratively tests the superpoly $p_{S(I)}$ to decide whether the degree of t_I should be increased or decreased. In [10], the table storing these $2^{|I_{\max}|}$ values is *Moebius-transformed* to compute at once the sums over $\binom{|I_{\max}|}{d}$ subcubes of $C_{I_{\max}}$ of degree d, for $d = 0, \ldots, |I_{\max}|$. These cubes are all the possible subcubes C_I of $C_{I_{\max}}$ in which the variables $\mathbf{x}_{\bar{I}}$ have been set equal to $\mathbf{0}$. None of these two strategies is suitable for GPUs: the stochastic nature of the random walk prevents the sequence of steps from being determined *a priori*, since the computation is performed only when (and if) needed; the Moebius transform requires a rigid schema of calculations and a large number of alternating read and write operations in memory that must be synchronized. Both approaches are conceived for implementations in which computational power is a constraint (while memory is not), and all advantages of using the Moebius Transform are lost in case of parallel processing.

Additionally, storing single evaluations of the cipher in T means testing only subcubes of a maximal cube of size $|I_{\max}| = \log_2(|T|/M)$, but, with the memory available in current GPUs, $\log_2(|T|/M)$ is not large enough for any reasonably strong cipher.

The proposed design of our attack relies on the following rationale: exploring only a portion of the maximal cube $C_{I_{\max}}$, considering only subsets $I \subseteq I_{\max}$ characterized by a non-empty *minimal* intersection $I_{\min}$. Quite naturally, a similar design leads to two distinct CUDA[1] kernels, respectively responsible for: (1) computing many variants of the cube $C_{I_{\min}}$, one for each of the possible combinations of the indices in $I_{\max} \setminus I_{\min}$, and writing the results in memory; (2) combining the stored results to test all cubes C_I such that $I_{\min} \subseteq I \subseteq I_{\max}$. Following this approach, the size of the explored $I_{\max}$ can be raised to $|I_{\max}| = |I_{\min}| + \log_2(|T|/M)$, with read and write memory operations carried out by different kernels.

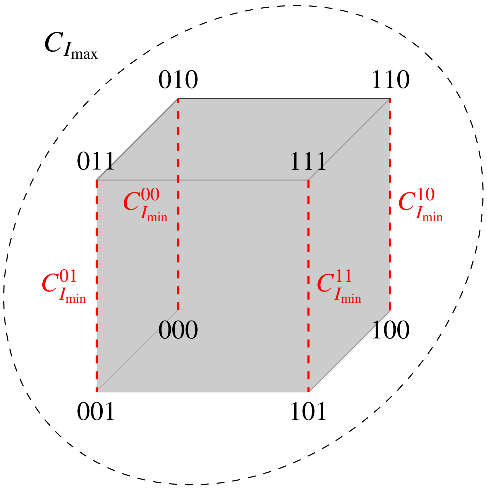
With respect to the notation introduced in Section 2.2.1, let us distinguish the public variables $\mathbf{x}$ into three sets $\mathbf{x}_{\text{fix}}$, $\mathbf{x}_{\text{free}}$, and $\mathbf{x}^*$, of size d_{fix}, d_{free}, and $n - d$, respectively, where $d = d_{\text{fix}} - d_{\text{free}}$. The variables $\mathbf{x}_{\text{fix}}$ correspond to the *fixed* components of $C_{I_{\max}}$ identified by $I_{\min}$, *i.e.*, $I_{\min} = \{i_1, \ldots, i_{d_{\text{fix}}}\}$, whereas the variables $\mathbf{x}_{\text{free}}$ correspond to the remaining *free* components of $C_{I_{\max}}$, *i.e.*, $I_{\max} \setminus I_{\min} = \{j_1, \ldots, j_{d_{\text{free}}}\}$ and $|I_{\max}| = d$. The variables $\mathbf{x}^*$ are the remaining public variables that fall outside $I_{\max}$.

The two kernels of our attack can be described as follows:

Kernel 1 It uses $2^{d_{\text{free}}}$ warps. Since, as described before, the 32 threads belonging to the same warp perform exactly the same operations but for different keys, in the following we simply consider a representative thread per warp and ignore the private variables $\mathbf{y}$.[2] For $t = 0, \ldots, 2^{d_{\text{free}}} - 1$, thread (*i.e.*, warp) s sums $E(\mathbf{v}, \mathbf{y})$ over each vertex of the cube $C^s_{I_{\min}}$ of size d_{fix} determined by the assignment of the d_{free}-bit representation $\mathbf{v}_{\text{free}}$ of integer s to the variables $\mathbf{x}_{\text{free}}$ and of $\mathbf{0}$ to the variable $\mathbf{x}^*$. Finally, thread s writes the sum in the s^{th} entry of table T, so that, at the end of the execution of the kernel, each entry of T contains the sum over a cube of size d_{fix}. These evaluations allow for testing the monomial $t_{I_{\min}}$ with all the aforementioned assignments to the

[1] CUDA is the software framework used for programming Nvidia GPUs.

[2] The work that here is assigned to a single thread can be actually split among any number of threads, reassembling the results at the end. We will not consider this possibility here for the sake of clarity.



$$I_{\max} = \{1,2,3\}$$

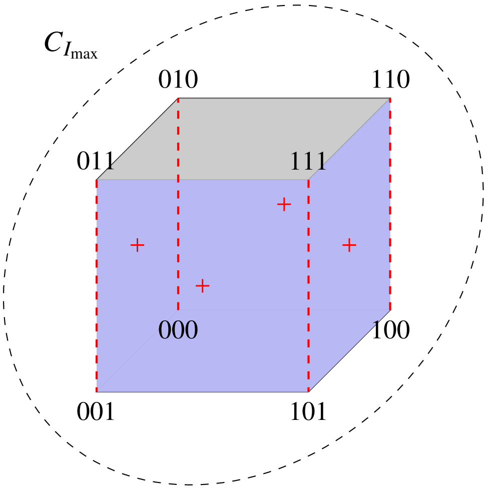
$$I_{\min} = \{2\}$$

$$I_{\max} \setminus I_{\min} = \{1,3\}$$

Kernel1 computes many variants of the cube $C_{I_{\min}}$, one for each of the possible combinations of the indices in $I_{\max} \setminus I_{\min} = \{1,3\}$ and then stores the sums in memory. In this example, the four cubes in red are those calculated by the **Kernel1**, $C^{00}_{I_{\min}}$, $C^{10}_{I_{\min}}$, $C^{01}_{I_{\min}}$, and $C^{11}_{I_{\min}}$ each one of dimension one. It is worth to notice that all the operations are performed module 2, so the sums over a cube are actually xor operations ($\oplus$).

FIGURE 3.1: Kernel1 example.

other $n - d_{\text{fix}}$ variables. An example of the evaluations done by **Kernel1** is depicted in Figure 3.1.



$$C_{I_1} = C^{00}_{I_{\min}} + C^{01}_{I_{\min}}$$

$$C_{I_2} = C^{00}_{I_{\min}} + C^{10}_{I_{\min}}$$

$$C_{I_3} = C^{01}_{I_{\min}} + C^{11}_{I_{\min}}$$

$$C_{I_4} = C^{10}_{I_{\min}} + C^{11}_{I_{\min}}$$

Kernel2 reads the results calculated and stored by kernel1 and it combines them to test all cubes C_I such that $I_{\min} \subseteq I \subseteq I_{\max}$. In these example, the four cubes C_{I_1}, C_{I_1}, C_{I_1}, C_{I_1} of dimension two calculated by **Kernel2** are those identified by the red +. It is worth to notice that all the operations are performed module 2, so the sums over a cube are actually xor operations ($\oplus$).

FIGURE 3.2: Kernel2 example.

Kernel 2 By simply combining the values stored in T at the end of Kernel 1, it is now possible to explore cubes of potentially any size $d_{\text{fix}} + \delta$, with $0 \le \delta \le d_{\text{free}}$. Although the exploration can potentially follow many other approaches (*e.g.*, a random walk as in [9]), the large computing power of our platform suggests to test cubes exhaustively. Moreover, we extend the exhaustive search to an area never reached, to the best of our knowledge, in the literature. For all I such that $I_{\text{min}} \subseteq I \subseteq I_{\text{max}}$, this kernel considers all variants of cube C_I obtained assigning all possible combinations of values to the variables in $I_{\text{max}} \setminus I$. More precisely, for each possible choice of $\delta \in [0, d_{\text{free}}]$, there are exactly $\binom{d_{\text{free}}}{\delta} 2^{d_{\text{free}}-\delta}$ distinct cubes of size $d_{\text{fix}} + \delta$. In fact, we can choose δ free variables (the additional dimensions of the cube) in $\binom{d_{\text{free}}}{\delta}$ different ways, and we can choose the fixed assignment to the remaining $d_{\text{free}} - \delta$ variables in any of the $2^{d_{\text{free}}-\delta}$ possible combinations. In Figure 3.2 as example, some of the cubes evaluated by the **Kernel2** are reported.

We would like to highlight that in our scenario usually the value of d_{fix} is grater that d_{free}, so actually Kernel 2 is computationally dominated by Kernel 1, so the cost of our exhaustive search is negligible. This way, our design allows considering any possible assignment to variables outside the cube, to finally address the common conjecture (never proved in the literature), that assigning $\mathbf{0}$ is the best possible solution. As a matter of fact, this means a significant gain in terms of number of cubes tested with respect to previous works: with $2^{d_{\text{free}}}$ bits of memory, the approach used in [10] allows considering $2^{d_{\text{free}}}$ cubes, whereas we are able to test $3^{d_{\text{free}}}$ different cubes.

Finally, let us underline the significant divergence between our work and [47], the first paper to ever use GPUs for a cube attack. In [47], Fan and Gong use GPUs to accelerate the summation of the polynomial p over all subcubes of a maximal cube of size 16. However, they use a rather trivial approach: for a cube of size k, they launch 2^k threads in charge of evaluating the cipher E over each of the 2^k vertices of the cube, followed by a parallel reduction process where these 2^k values are summed using so-called *shared* memory. Basically, they just use GPUs to call E in parallel over multiple inputs, but sums over different cubes are processed sequentially, and the impact of the warp structure is completely overlooked. Additionally, they only consider a very small maximal cube, not discussing whether their scheme scales to larger dimensions. Conversely, the design and implementation of our GPU kernels thoroughly covers all steps of the attack, maximizing the performance subject to all architectural constraints, such as warp size and limited memory.

Besides the high-level engineering described before, we implement a few low-level optimizations:

- all computations internal to our kernels only rely on registers, that are the fastest type of memory available on GPUs;
- the design of our architecture divides the workflow into two separate kernels and this allows us to separate reads and writes on global memory so we do not need synchronization;
- the memory pattern we define to store the results guarantees memory coalescing accesses both in reading and writing;

- we leverage some built-in CUDA functions, such as *warp shuffle*, to efficiently exchange values among threads belonging to the same warp, when needed (*e.g.*, when we perform the linearity tests).

3.3.1 Cluster setup

	Nvidia P100	**Nvidia K80**
Market segment	Server	Server
Number and Type of GPUs	1 Tesla GP100	2 Tesla GK210B
Micro-architecture	Pascal	Kepler
Number of processor cores (Per GPU)	3584	2496
Boost frequency	1353 MHz	875 MHz
Single Precision Processing Power (Per GPU)	9.3 TFLOPS	8.73 GFLOPS
Memory available (Per GPU)	16 GB	12 GB
Memory type	HBM2	DDR5
Memory Bandwdith (Per GPU)	732 GB/s	240 GB/s

Table 3.1: Tested GPUs characteristics - Cube Attack.

We ran our attack on a cluster composed by 3 nodes, each equipped with 2 Tesla K80 with 24 GB of *global* memory and 4 Intel Xeon CPU E5-2640 with 128 GB of RAM, running CentOS 6.6 and Cuda 7.5. We recall that each K80 is in turn composed by two K40 with 12GB of global memory each. We have also access to two additional Linux systems both equipped with Nvidia Pascal GPUs, the first one has one Tesla P100 with 16 GB of *global* memory and the second one has a Quadro P6000 with 24 GB of *global* memory. We used the P100 mainly to evaluate the performance of our framework on Pascal architecture, whereas we used the P6000 to run our biggest experiment on round-reduced version of Trivium where define $d_{\text{free}} = 26$ and $d_{\text{fix}} = 18$. Thanks to this last experiment, we discover few maxterm candidates up to 830 output bits. The description and the analysis of these results are going to appear in a work currently being finalized. The performance experiments are deeply discussed in Section 3.4 while the cryptographic results on the target ciphers are discussed in the corresponding Sections. See Table 3.1 for further information about the GPUs we use for performance experiments.

3.4 Performance Analysis

To evaluate the performance of our GPU based solution, we carried out an extensive experimental campaign on the previously described systems. For comparison, we also developed a parallel CPU version of the cube attack based on OpenMP. This implementation exploits the 32 cores of the four Intel(R) Xeon(R) CPU E5-2640 and scales linearly as the size of the problem. Each performance test has been repeated 5 times and the average time is reported. We evaluate

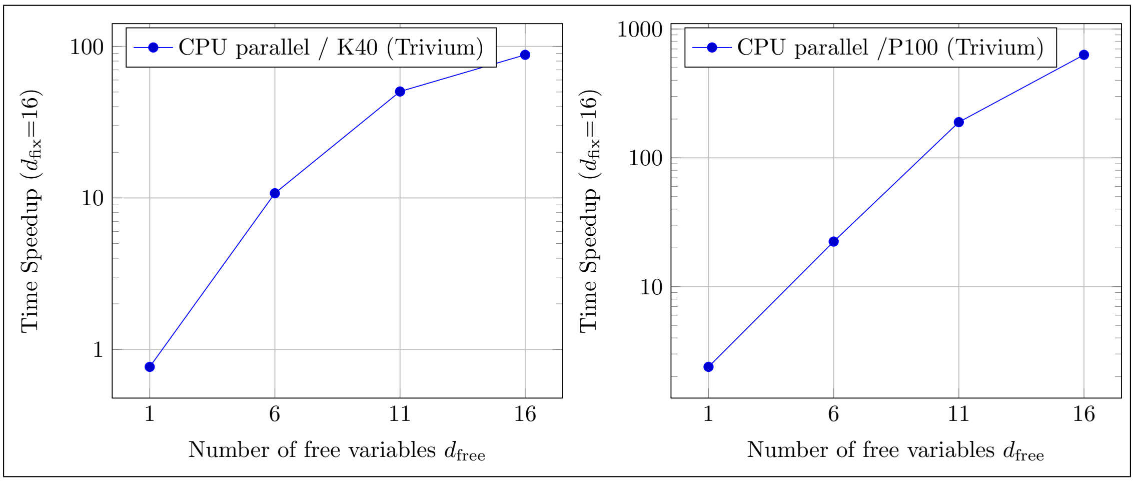


FIGURE 3.3: Speedup of Parallel CPU vs GPU on Trivium.

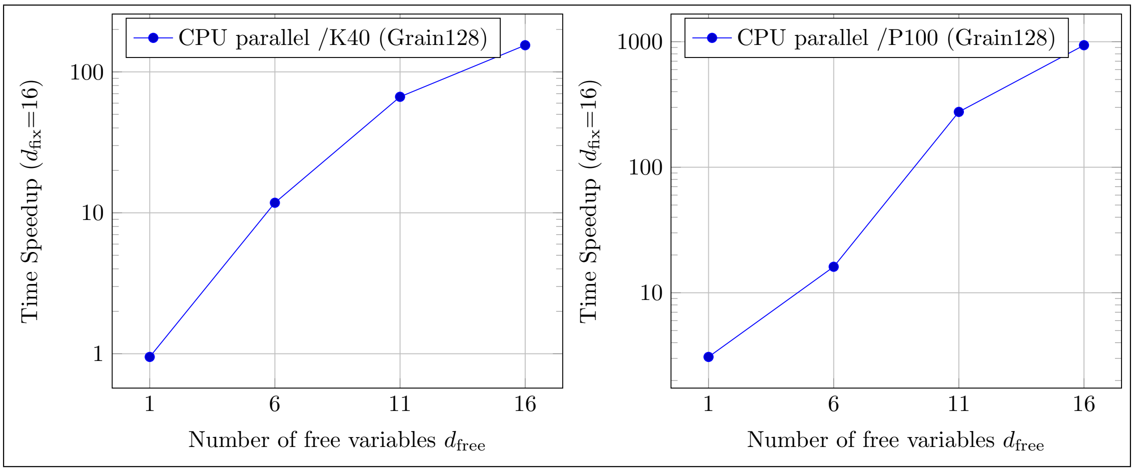


FIGURE 3.4: Speedup of Parallel CPU vs GPU on Grain128.

the performance for each of the ciphers we targeted as it is a crucial factor for the feasibility of the attack.

In Figures 3.3 and 3.4, we report the speedup gained by the GPU versions with respect to the parallel CPU version. Anchoring the size of I_{min} to $d_{\text{fix}} = 16$, we evaluated the two solutions over a growing maximal cube $C_{I_{\text{max}}}$, whose size is exponential in the cardinality d_{free} of the index set $I_{\text{max}} \setminus I_{\text{min}}$. The experiments show that the benefit of using the GPU version grows with the number of free variables d_{free} considered, outreaching a 80× speedup when $d_{\text{free}} = 16$ on K40 and 630× on P100 for Trivium, and up to 154× on K40 and 930× on P100 for Grain-128. It is worth noticing that all versions (GPU and CPU) rely on the same base functions to implement the ciphers.

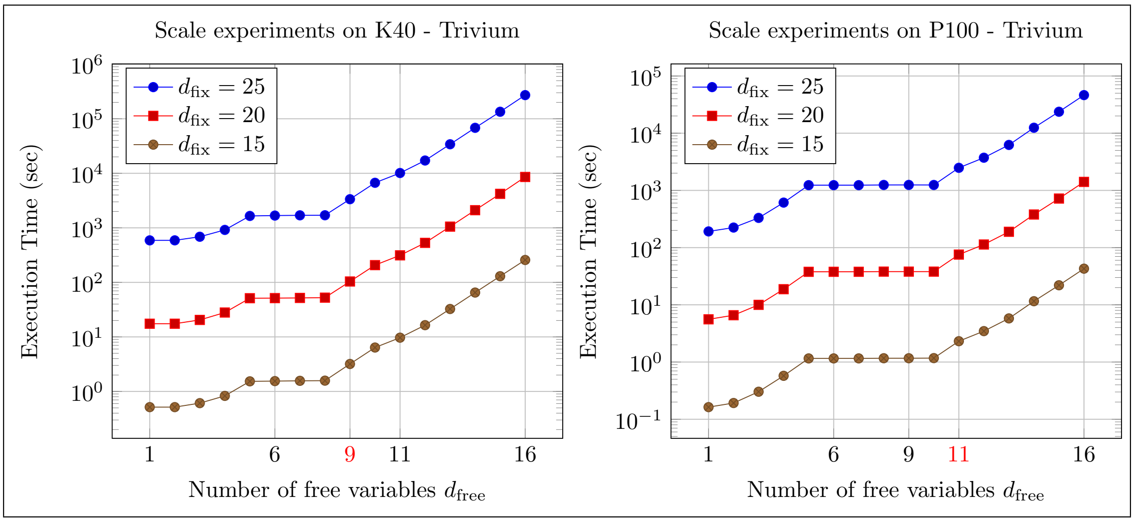


FIGURE 3.5: Scaling experiments on Trivium.

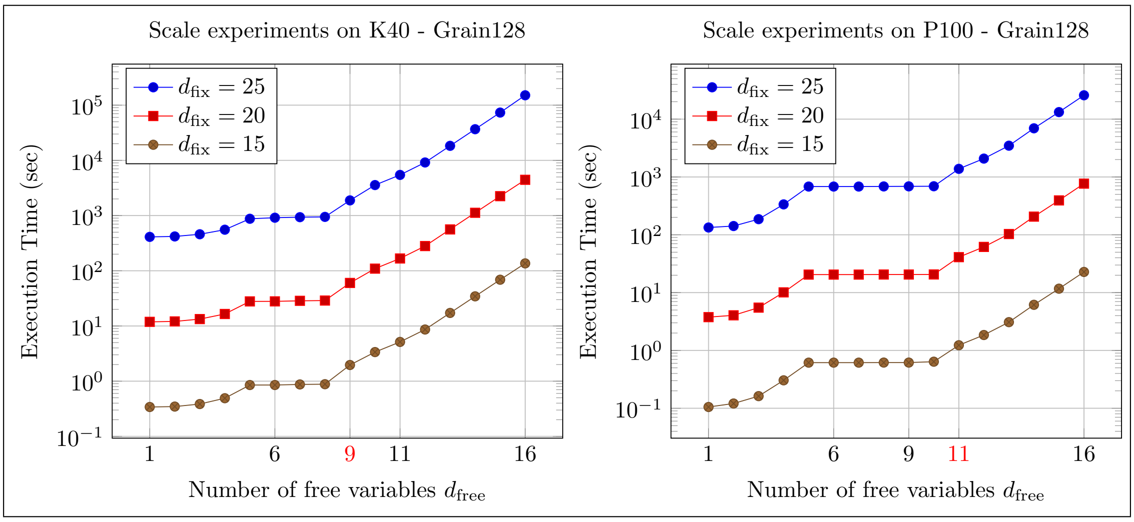


FIGURE 3.6: Scaling experiments on Grain128.

A second set of experiments aimed at evaluating how the GPU solution scales when d_{free} increases. To this end, we measured the execution time of the attack for all the target ciphers on a

Kepler K40 and Pascal P100, as a function of d_{free}, for different values of d_{fix}. These measurements, reported in Figure 3.5 and 3.6, show that: (i) varying d_{fix} only yields an additive shift; (ii) when the size of the problem is large enough to guarantee the full utilisation of computational resources ($d_{\text{free}} \geq 9$ on K40 and $d_{\text{free}} \geq 11$ on P100), our solution scales roughly linearly with the size of the problem, *i.e.*, exponentially with the number of dimensions of the explored cube.

To explain why $d_{\text{free}} = 9$ for K40 and $d_{\text{free}} = 11$ for the P100 behave as a threshold, we need to delve into some of the configuration aspects discussed in Section 3.3.

According to our design, each warp is in charge of the computation over a single subcube, so that all the information a thread needs are computed locally, and only the final results are stored in memory. We thus know how many warps are needed for a specific computation, and, based on our code optimization, we can predict the exact number of GPU blocks used for that computation. Specifically, the number of blocks N_{blocks} is determined as:

$$N_{\text{blocks}} = \left\lceil 2^{d_{\text{free}}} \cdot \frac{S_{\text{warp}}}{S^{\max}_{\text{block}}} \right\rceil$$

where S_{warp} is the warp size (*i.e.*, number of threads per warp), while $S^{\max}_{\text{block}}$ is the maximum block size (*i.e.*, maximum number of threads per block). In our case, $S_{\text{warp}} = 32$ and $S^{\max}_{\text{block}} = 1024$, thus yielding $N_{\text{blocks}} = \lceil 2^{d_{\text{free}}-5} \rceil$. This means that $d_{\text{free}} = 8$ and $d_{\text{free}} = 9$ correspond to, respectively, $N_{\text{blocks}} = 8$ and $N_{\text{blocks}} = 16$. These numbers should be compared with the number of Streaming Multiprocessors (SMs) of the GPU card used, that is 13 on the K40. $d_{\text{free}} = 9$ is the first value for which the number of allocated blocks is larger than the number of SMs, producing at least one active block per SM, and thus an use of the GPU at full operating speed. The same math can be applied when $d_{\text{free}} = 11$ on P100. In particular, when $d_{\text{fix}} = 25$, the execution time stays roughly constant (approximately 1010 sec) when $6 \leq d_{\text{free}} \leq 10$, before suddenly doubling when $d_{\text{free}} = 11$. This is perfectly coherent with the characteristics of the P100, as it has 60 SMs, and $2^{11-5} = 64$. As clearly highlighted in Figures 3.5, 3.6, the described behaviour is determined by the design of our framework and it does not depend on the cipher. It is worth noticing that the scalability of our framework does not depend on the target cipher or on a specific architecture (i.e. Kepler and Pascal).

In order to assess the impact of the GPU architecture on the performance of our framework, we compared the execution time of the attack on the Pascal P100 and on the Kepler K40. Interestingly, running the experiments on the P100 did not require any code adjustment, we just had to recompile our code for that architecture.

Figure 3.7b shows the significant speedup yielded by using the Pascal architecture. In particular, when $d_{\text{free}} = 16$, the execution time on the P100 becomes more than 5× smaller with respect to the K40. These very promising preliminary tests of the performance of the Pascal architecture support the viability of our approach, and suggest that the search for maxterms of a cube attack can significantly benefit from the progress of GPU technologies.

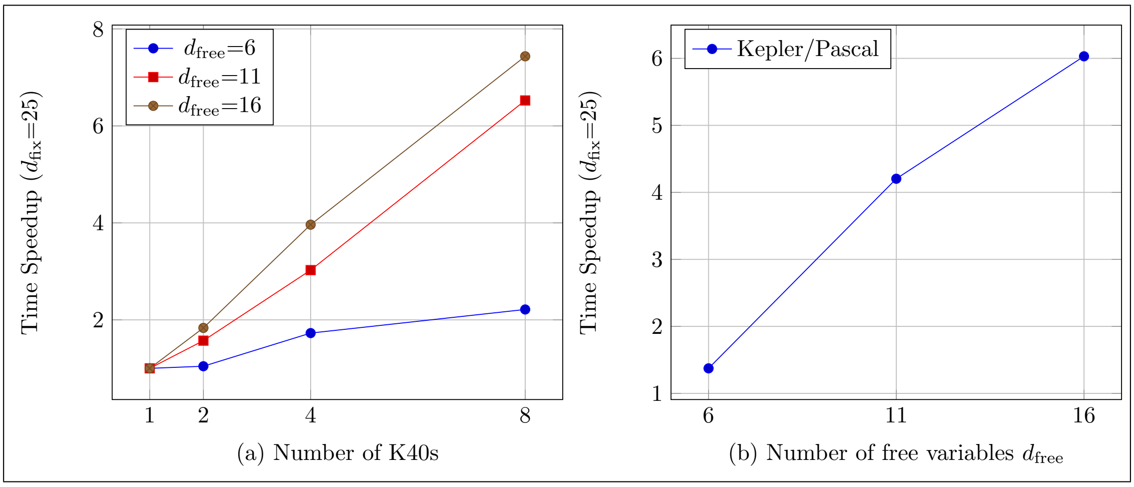


FIGURE 3.7: Speedup of multi-GPU w.r.t. mono-GPU (a) and of Pascal P100 w.r.t. Kepler K40 (b) (Trivum cipher).

Moreover, in order to evaluate the scalability of our framework on multi-GPU systems we compared the execution time of the attack on cubes of size $d_{\text{fix}} = 25$ and $d_{\text{free}} \in \{6, 11, 16\}$, when the workload is equally distributed among 1, 2, 4, and 8 K40s. The distribution process splits a cube of size $|I|$ among 2^l GPUs by assigning to each GPU a subcube of size $|I - l|$. It then merges all 2^l partial computations altogether obtaining the original cube. As mentioned before for the mono-GPU experiments, in order to use the GPUs at operating speed, for each of them the number of allocated blocks should be larger than the number of SMs. For this reason, as reported in Figure 3.7a, the speedup is linear with respect to the number of GPUs only for $d_{\text{free}} = 16$. We report the results of the multi-gpu experiments only for Trivium cipher as the scalability of our framework does not depend on the target cipher (as clearly stated in the previous scaling experiments). We could not carry out this last experiments on Pascal architecture as we have access to one P100 only.

Finally, we ran the attack, for both the target ciphers, under the control of the Nvidia profiler in order to measure the ALU occupancy achieved by our Kernels. Kernel1 is invoked just once to fill the whole table T, with an occupancy consistently over 95% when $d_{\text{free}} \geq 10$. Kernel2 is instead invoked once per each $\delta \in [0, d_{\text{free}}]$ to compute all available cubes of size $d_{\text{fix}} + \delta$. The maximum occupancy exceeds 95% as soon as $d_{\text{free}} =\geq 12$, with an average of approximately 50%. In either case the impact of d_{fix}, which determines the load of each thread, is negligible. Considering that d_{free} should be maximized to improve the attack success rate, our kernels guarantee an excellent use of resources in any realistic application. For instance in our experiments we set $d_{\text{free}} = 16$, which guarantees an occupancy above 99% for Kernel1, and a maximum occupancy above 98% for Kernel2.

4 Cryptanalysis Results

4.1 The Attack on Trivium

This Section reports the results obtained by our GPU implementation of the cube attack against reduced-round Trivium. We recall that the attack ran on a cluster composed by 3 nodes, each equipped with 2 Tesla K80 with 24 GB of *global* memory and 4 Intel Xeon CPU E5-2640 with 128 GB of RAM.

We performed a formal evaluation of our implementation, by checking our experimental results against Trivium's polynomials, explicitly computed up to 400 initialization rounds. In the following, the number of initialization rounds instead matches (and slightly overtakes) the best results from the literature, thus reaching a point where a symbolic evaluation would be prohibitive. Still, the results we exhibit are obtained from experiments specifically designed to reproduce tests carried out in the recent past [10], so as to provide, at the same time: (i) a direct comparison of our results with the state-of-the-art; (ii) an immediate means to assess the advantages of our approach, and (iii) a further validation of the correctness of our code.

4.1.1 Experimental setting

In our attack, we consider two different reduced-round variants of Trivium, corresponding to 768 and 800 initialization rounds, respectively. As explained and motivated in Section 3.2, in our scheme, each call to Trivium produces 32 key-stream bits, which we use in our concurrent search for superpolys. The most significant practical consequence of a similar construction is the ability to devise attacks to Trivium reduced to any number of initialization rounds ranging from 768 to 831, at the cost of just two attacks, although the number of available superpolys decreases with the number of rounds. As a matter of fact, the j^{th} output bit after 768 rounds can also be interpreted as the $(j - i)^{th}$ bit of output after $768 + i$ initialization rounds, for any $j \geq i$. In other words, an attack to Trivium reduced to $768 + i$ initialization rounds can count upon all superpolys found in correspondence of the j^{th} output bit after 768 rounds, for all $j \geq i$.

For each of the two attacks (768 and 800 initialization rounds), we ran 12 independent runs, corresponding to 12 different choices for the pair of sets of variables $I_{\text{min}}, I_{\text{max}}$ (with $I_{\text{min}} \subset I_{\text{max}}$) that define the minimal and maximal tested cubes $C_{I_{\text{min}}}$ and $C_{I_{\text{max}}}$. The size of I_{min} and $I_{\text{max}} \setminus I_{\text{min}}$ is $d_{\text{fix}} = 25$ and $d_{\text{free}} = 16$, respectively, for all runs, so that all maximal cubes have size $d = d_{\text{fix}} + d_{\text{free}} = 41$. Peculiarly to our implementation, when we test the monomial composed of all variables in some set $I_{\text{min}} \subseteq I \subseteq I_{\text{max}}$, we exhaustively assign values to all public variables in $I_{\text{max}} \setminus I$, thus concurrently testing the linearity of $2^{41-|I|}$ possibly different superpolys. This feature of our attack – a possibility overlooked in the literature, but almost free-of-charge in our framework – provides primary benefits, as described in Section 4.2.

In all the reported experiments, we use a complete-graph linearity test based on combining 10 randomly sampled keys.

As mentioned before, we implemented two attacks, against Trivium reduced to 768 (Trivium-768 in the following) and 800 (Trivium-800) initialization rounds, respectively. In both cases, our setting allows obtaining superpolys corresponding to 32 output bits altogether, at the cost of a single attack.

4.1.2 Results against Trivium-768

For the attack against Trivium-768, we took inspiration from [10]: we launched 12 runs based on 12 different pairs $I_{\text{min}}, I_{\text{max}}$, chosen so as to guarantee that each of the 12 linearly independent superpolys found in [10] after 799 initialization rounds had to be found by one of our runs. The rationale of reproducing results from [10] was to both test the correctness of our implementation, and provide a better understanding of the advantages of our implementation with respect to the state-of-the-art. In this sense, let us highlight that a single run of ours cannot be directly compared with all results presented in [10], because each of our runs only explores the limited portion of the maximal cube $C_{I_{\text{max}}}$ composed by all super-cubes of $C_{I_{\text{min}}}$.

To better describe our results, let us introduce the binary matrix A whose element $A(i, j)$ is the coefficient of variable y_j in the i^{th} available superpoly. The rank of A, denoted $\text{rk}(A)$, clearly determines the number of key bits that can be recovered in the online phase of the attack based on the available superpolys, before recurring to brute-force.

As described before, the superpolys yielded by the i^{th} output bit after round 768 are usable to attack Trivium for any number of initialization rounds between 768 and $768 + i$. It is possible to define 32 different matrices $A_{768}, \ldots, A_{799}$: A_{768} includes all superpolys found, while

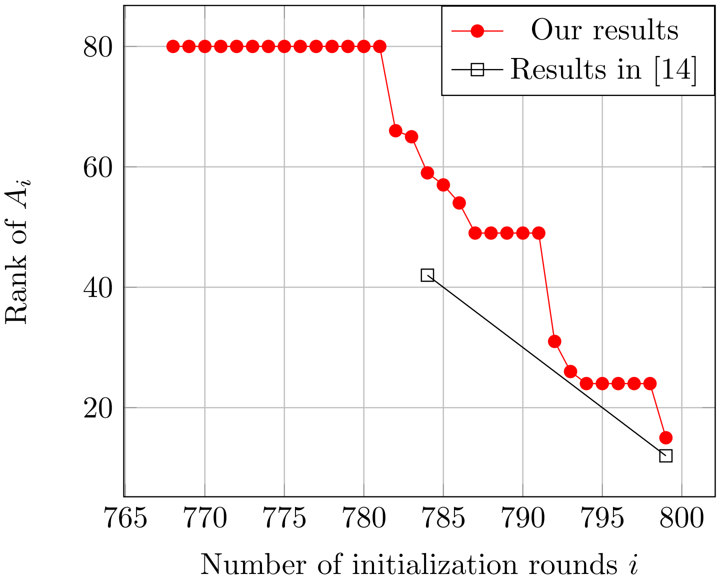


FIGURE 4.1: Comparison with previous work.

each matrix A_{768+i} is obtained by incrementally removing the superpolys yielded by output bits $0,\ldots,i-1$. Figure 4.1 shows $\mathrm{rk}(A_i)$ as a function of i, comparing our findings with those of [10].

Overall, our results extend the state-of-the-art in a remarkable way, especially if we consider that our quest for maxterms was circumscribed to multiples of 12 *base* monomials of degree 25. In particular, let us highlight a few aspects that emerge from Figure 4.1:

- Since our runs were designed to include all 12 maxterms found in [10] after 799 initialization rounds, it is not surprising that $\mathrm{rk}(A_{799})$ is at least 12. Yet, it is indeed larger: we found 3 more linearly independent superpolys, reaching $\mathrm{rk}(A_{799}) = 15$.
- Although we did not force our tested cube to include the maxterms found in [10] after 784 rounds, we have $\mathrm{rk}(A_{784}) = 59$, compared with rank 42 found in [10].
- Finally, and probably most important, our attack allows a full key recovery up to 781 initialization rounds.

Selected superpolys that guarantee the above ranks are reported in Section 4.4, together with the corresponding maxterms. Very interesting is also *how* novel superpolys were found, a point that is better described in the following.

4.1.3 Results against Trivium-800

To provide a further test of the quality of our attack, we launched a preliminary attack against Trivium-800. We kept unvaried all the parameters of the attack (12 independent runs, 25 fixed plus 16 free public variables considered per run, 32 output bits attacked altogether), but this time we chose the sets $I_{\text{min}}, I_{\text{max}}$ at random. In total, we could find a single maxterm corresponding to 800 rounds, and no maxterms afterwards. Although our findings only allow to cut in half the complexity of a brute force attack, this is the first ever superpoly found considering more than 799 initialization rounds. However, as stated in Section 3.3.1, we ran an additional experiment against Trivium-800 where $d_{\text{free}} = 26$ and $d_{\text{fix}} = 18$ and we discover few candidate maxterms up to 830 output bit; we are going to discuss deeply this last results as well as the corresponding superpolys in a work currently being finalized.

4.2 Further Discussion

Hereafter, we provide a more detailed analysis and a further discussion of our findings, considering two aspects in particular: the reliability of commonly used linearity tests, and the peculiar advantages of our attack design. Unless otherwise specified, in the following we always focus on Trivium-768.

4.2.1 On probabilistic linearity.

A common practice in the cube attack related literature consists in using a probabilistic linearity test, meaning that a (small) chance exists that the superpolys found by an attack are not actually linear. In particular, the best results obtained with the cube attack against Trivium use a complete-graph test, which, with respect to the standard BLR test, trades-off accuracy for efficiency. The viability of a similar choice is supported by previous works [59, 60], showing that the complete-graph test behaves essentially as a BLR test in testing a randomly chosen function f, with the quality of the former being especially high if the nonlinearity (minimum distance from any affine function) of f is large, that is, when the result of the test is particularly relevant.

Following the trend, we chose to implement a complete-graph test based on a set of 10 randomly chosen keys, exactly as done in [10]. However, while increasing the number of tests done *during* the attack was costly for us (it impacts on memory usage), implementing further tests on the superpolys found *at the end* of the attack was not. We therefore decided to put our superpolys through additional tests involving other 15 keys chosen uniformly at random. Figure 4.2 compares $\text{rk}(A_i)$ as a function of i, for our full results and our filtered results, in which all superpolys that failed at least one of the additional tests have been removed. Let us stress once more that these two sets of results cannot be defined as wrong and correct, but they rather correspond to two different levels of trust in the found superpolys. In a sense, choosing between the two sets is equivalent to selecting the desired trade-off between efficiency and reliability of the attack: our

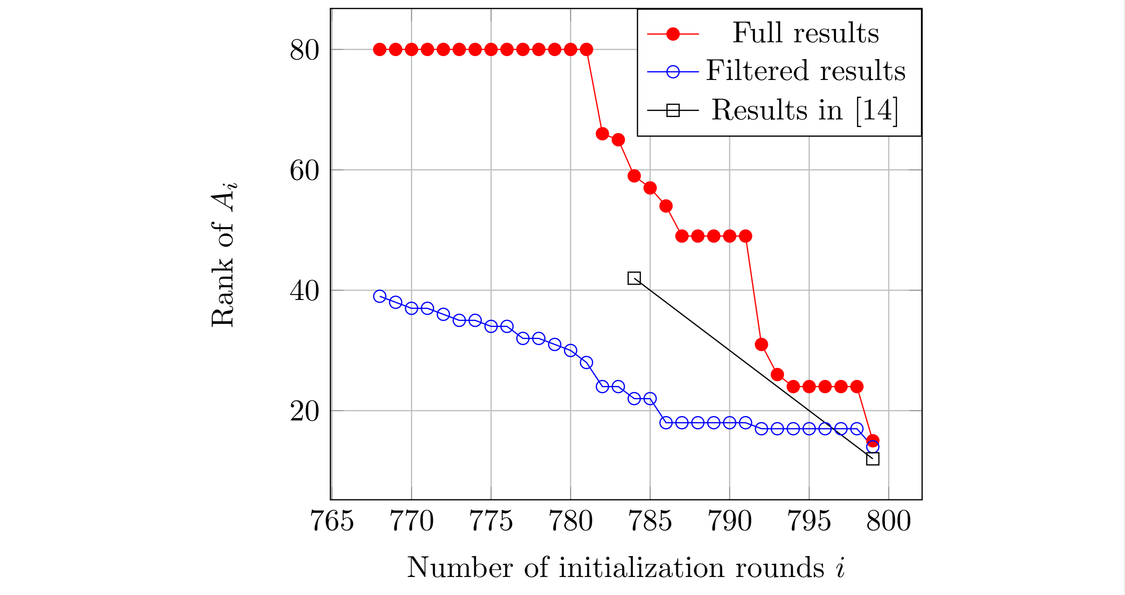


FIGURE 4.2: Impact of probabilistic linearity.

full results permit a faster attack, which however may fail for a subset of all possible keys. Of course, several intermediate approaches are possible.

4.2.2 On using 32 output bits

A significant novelty of our implementation consists in the ability to concurrently attack 32 different polynomials, which describe 32 consecutive output bits of the target cipher. This choice is induced by GPUs features – as discussed in Section 3.2 – yet it is natural to assess what benefits it introduces. In Section 3.2 we showed that looking at 32 output bits altogether can be considered a way to concurrently attack 32 different reduced-round variants of Trivium. However, aiming to extend the attack to the full version of the cipher, our implementation can be used to check whether the same set of monomials yield different superpolys, hopefully involving different key variables, when we focus on different output bits. To this end, let us introduce a new set of matrices $B^0_{768}, \ldots, B^{31}_{768}$, where each B^j_{768} is obtained considering only the superpolys yielded by output bits $0, \ldots, j$ after 768 initialization rounds (*i.e.*, $A_{768} = B^{31}_{768}$). Figure 4.3 shows $\mathrm{rk}(B^j_{768})$ as a function of j, for both our full results and our filtered results. What the figure highlights is that considering several output bits altogether for the same version of the cipher, albeit possibly causing issues related to memory usage, introduces the expected benefit, indeed a remarkable benefit if the matrix rank is initially (*e.g.*, when $j = 0$) low. This is the first ever result showing that considering a larger set of output bits is a viable alternative to exploring a larger cube.

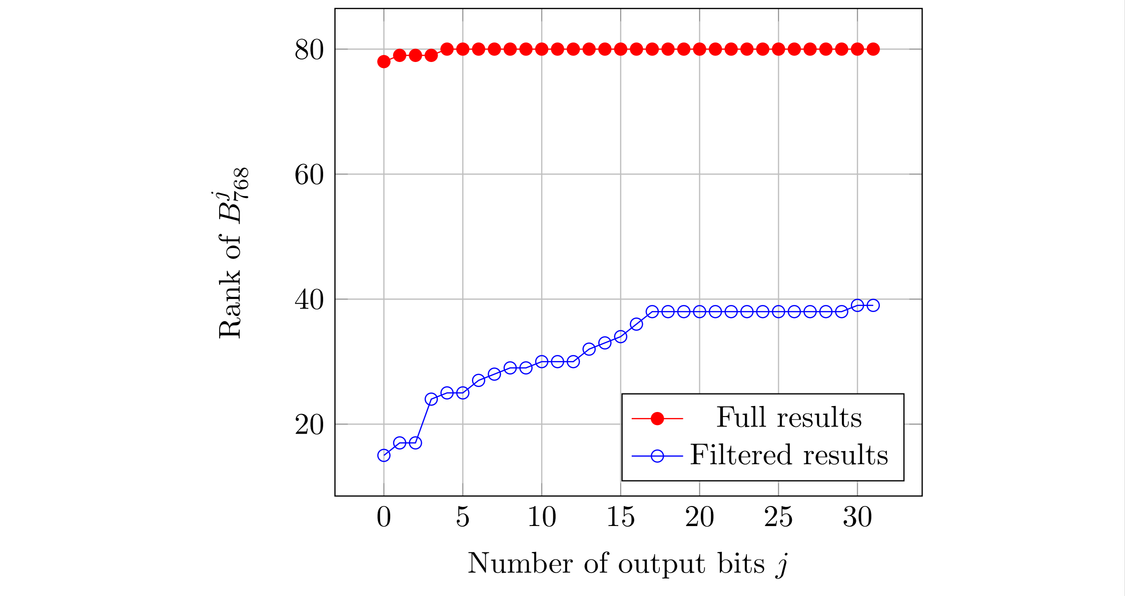


FIGURE 4.3: Impact of using 32 output bits.

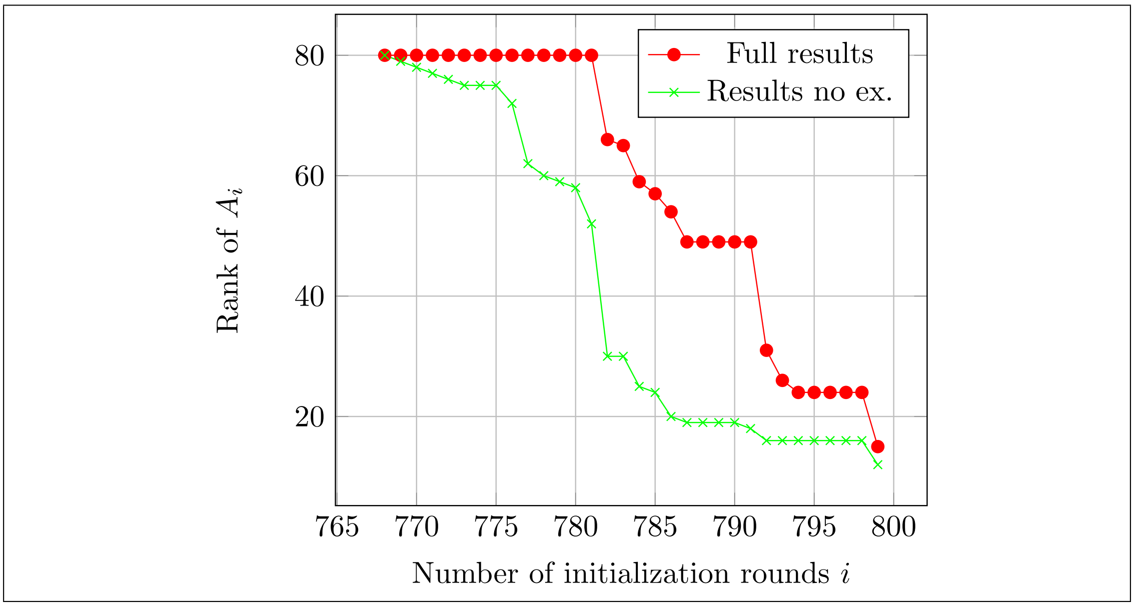


FIGURE 4.4: Impact of exhaustive search.

4.2.3 On the advantages of the exhaustive search

As described before, our implementation allows to find significantly more linearly independent superpolys than previous attempts from the literature. One of the reasons of our findings is the parallelization that makes possible to carry out, at a negligible cost, an exhaustive search over all public variables in $I_{\max} \setminus I$ when the cube C_I is under test. Figure 4.4 shows how $\text{rk}(A_i)$ varies with i for our full results, in comparison to what happens when we impose that all variables in $I_{\max} \setminus I$ are set to 0, as usually done in related work. What emerges is that through an exhaustive search it is indeed possible to remarkably increase $\text{rk}(A_i)$. Significantly, the exhaustive search is what allows us to improve on the state-of-the-art for $i = 799$, which, among other things, suggests that the benefits of the exhaustive search are particularly relevant when increasing the number of tested cubes would be difficult otherwise (*e.g.*, by considering other monomials).

Another consequence of implementing an exhaustive search is that we found many redundant superpolys, *i.e.*, superpolys that are identical or just linearly dependent with the ones composing the maximal rank matrix $\tilde{A}$. A similar finding is extremely interesting as we expect it to provide a wide choice of different IV combinations yielding superpolys that compose a maximal rank submatrix $\tilde{A}$, thus weakening the standard assumption that cube attacks require a completely tweakable IV.

4.3 The Attack on Grain-128

We perform a formal evaluation also of our Grain-128 implementation, by checking our experimental results against the polynomials, explicitly computed up to 96 initialization rounds. In literature there are some works focus on attacking Grain-128 by using Cube Attack variants, in particular in [12], [11]. However, to the best of our knowledge, Grain-128 has never been targeted using the original cube attack. We ran few preliminarily attacks against Grain-128 where the number of initialization rounds is reduced to 160 and the indexes of controlled IV-bits are randomly selected and we discovered few candidate maxterms. A detailed description of our findings against Grain-128 as well as the results against other new ciphers will appear in a work currently being finalized.

4.4 Tables of maxterms and superpolys

In the following there are reported the tables containing some maxterms and the corresponding superpolys we discover in the attack.

maxterm bits	**superpoly**	**round**
3, 6, 8, 10, 12, 14, 18, 19, 20, 23, 25, 27, 31, 33, 38, 40, 43, 45, 48, 53, 54, 56, 58, 60, 62, 63, 69, 75, 77, 79, 80	x_{55}	781
1, 5, 7, 8, 10, 15, 16, 18, 20, 23, 25, 27, 32, 33, 36, 38, 40, 41, 43, 47, 49, 52, 53, 54, 56, 58, 63, 69, 71, 75, 77, 80	x_{69}	781
1, 6, 7, 8, 10, 12, 16, 19, 21, 24, 25, 27, 31, 33, 36, 38, 40, 41, 43, 47, 49, 52, 53, 56, 58, 63, 67, 69, 71, 73, 77, 80	x_{60}	781
1, 2, 3, 5, 6, 7, 8, 12, 14, 15, 16, 19, 21, 23, 25, 27, 36, 38, 40, 43, 45, 47, 49, 54, 56, 58, 60, 62, 69, 71, 73, 74, 80	$x_{51} + 1$	781
1, 2, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 23, 25, 27, 36, 38, 40, 43, 45, 47, 52, 54, 56, 58, 60, 62, 69, 71, 73, 76, 79, 80	x_{45}	781
1, 2, 5, 6, 7, 8, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 43, 45, 47, 49, 52, 54, 56, 58, 62, 65, 69, 71, 73, 76, 80	$x_{43} + x_{58}$	781
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 23, 27, 33, 36, 38, 40, 43, 45, 47, 52, 54, 56, 58, 60, 62, 69, 71, 73, 74, 79, 80	x_{23}	781
1, 3, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 33, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 65, 69, 71, 75, 77, 79, 80	$x_8 + x_{35} + x_{64}$	781
1, 5, 7, 8, 10, 12, 14, 15, 16, 18, 20, 23, 24, 25, 27, 32, 33, 36, 40, 41, 43, 47, 49, 52, 53, 56, 58, 63, 69, 71, 75, 77, 80	$x_{67} + 1$	781
3, 5, 6, 8, 10, 12, 14, 16, 18, 19, 20, 23, 24, 25, 27, 31, 33, 38, 43, 45, 48, 53, 54, 56, 58, 60, 62, 63, 69, 75, 77, 79, 80	x_2	781

TABLE 4.1: Maxterms and superpolys after 781 initialization rounds of Trivium (continue).

maxterm bits	**superpoly**	**round**
6, 8, 10, 12, 14, 15, 16, 18, 19, 20, 21, 23, 25, 27, 29, 31, 33, 36, 38, 40, 41, 42, 45, 49, 54, 56, 60, 62, 63, 69, 73, 75, 80	x_{58}	781
1, 2, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 23, 27, 36, 38, 40, 43, 45, 47, 49, 52, 54, 56, 58, 60, 62, 69, 71, 73, 74, 76, 79, 80	$x_{62} + 1$	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 19, 21, 25, 30, 31, 32, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 63, 69, 71, 73, 75, 80	$x_3 + x_{25} + x_{39} + x_{40} + x_{51} + x_{66} + x_{67} + x_{78} + 1$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 19, 21, 25, 27, 31, 33, 38, 40, 41, 43, 45, 47, 48, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{10} + x_{13} + x_{14} + x_{19} + x_{25} + x_{28} + x_{29} + x_{31} + x_{37} + x_{40} + x_{46} + x_{52} + x_{53} + x_{55} + x_{56} + x_{57} + x_{60} + x_{61} + x_{62} + x_{64} + x_{66} + x_{68} + x_{69} + 1$	781
1, 3, 5, 7, 12, 14, 15, 16, 18, 19, 20, 21, 24, 25, 27, 31, 33, 36, 40, 41, 45, 49, 54, 56, 58, 60, 62, 63, 66, 71, 73, 75, 77, 80	x_{57}	781
1, 3, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 33, 36, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 65, 69, 71, 75, 77, 79, 80	$x_{43} + x_{58} + x_{64} + x_{66} + x_{70}$	781
1, 3, 5, 6, 7, 8, 12, 13, 14, 15, 16, 19, 21, 23, 25, 27, 36, 38, 40, 43, 45, 47, 49, 52, 54, 56, 58, 62, 65, 69, 71, 73, 76, 79, 80	x_{65}	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 24, 25, 31, 32, 33, 36, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 63, 69, 71, 73, 75, 80	$x_{23} + x_{39} + x_{50} + x_{66} + x_{67} + x_{79} + 1$	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 25, 30, 31, 32, 33, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_9 + x_{18} + x_{24} + x_{26} + x_{32} + x_{33} + x_{34} + x_{42} + x_{51} + x_{53} + x_{54} + x_{58} + x_{59} + x_{64} + x_{66} + x_{68} + x_{69} + x_{80} + 1$	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 19, 21, 24, 25, 31, 32, 33, 36, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 63, 69, 71, 73, 75, 80	$x_{52} + x_{66} + x_{67} + x_{79}$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 18, 19, 21, 25, 27, 31, 33, 38, 40, 41, 43, 45, 47, 48, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{13} + x_{14} + x_{19} + x_{25} + x_{27} + x_{28} + x_{29} + x_{31} + x_{39} + x_{41} + x_{42} + x_{46} + x_{51} + x_{52} + x_{54} + x_{55} + x_{56} + x_{57} + x_{61} + x_{62} + x_{64} + x_{65} + x_{66} + x_{69} + x_{78}$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 18, 19, 24, 25, 27, 31, 32, 33, 36, 38, 40, 41, 45, 47, 48, 49, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{16} + x_{26} + x_{27} + x_{38} + x_{43} + x_{53} + x_{54} + x_{56} + x_{65} + x_{67} + x_{80}$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 19, 21, 24, 25, 30, 31, 32, 33, 36, 38, 40, 41, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{25} + x_{27} + x_{30} + x_{54} + x_{57}$	781
1, 2, 3, 5, 6, 7, 8, 12, 14, 15, 16, 19, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 52, 54, 56, 58, 60, 62, 65, 69, 70, 71, 73, 74, 76, 80	x_{42}	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 21, 25, 27, 30, 31, 32, 33, 38, 40, 41, 43, 45, 47, 48, 49, 53, 54, 56, 63, 69, 71, 73, 75, 80	$x_{14} + x_{29} + x_{41} + x_{55} + x_{61} + x_{62}$	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 24, 25, 31, 32, 33, 36, 38, 40, 41, 43, 45, 47, 48, 49, 50, 53, 54, 56, 63, 69, 71, 73, 75, 80	$x_{39} + x_{66}$	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 25, 30, 31, 32, 33, 38, 40, 41, 43, 45, 47, 48, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{24} + x_{55} + x_{61} + x_{66} + x_{67} + 1$	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 19, 21, 24, 25, 30, 31, 33, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{12} + x_{27} + x_{32} + x_{33} + x_{40} + x_{42} + x_{51} + x_{53} + x_{57} + x_{58} + x_{60} + x_{64} + x_{80} + 1$	781
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 19, 21, 24, 25, 30, 31, 33, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{27} + x_{32} + x_{42} + x_{53} + x_{58} + x_{60} + x_{64} + x_{78} + x_{80} + 1$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 18, 19, 24, 25, 27, 30, 31, 32, 33, 38, 40, 41, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{11} + x_{24} + x_{25} + x_{29} + x_{30} + x_{31} + x_{40} + x_{41} + x_{45} + x_{50} + x_{52} + x_{53} + x_{54} + x_{56} + x_{58} + x_{61} + x_{65} + x_{66} + x_{67} + x_{68} + x_{77} + x_{79} + x_{80} + 1$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 19, 21, 24, 25, 30, 31, 32, 33, 36, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{14} + x_{16} + x_{27} + x_{29} + x_{30} + x_{31} + x_{40} + x_{41} + x_{42} + x_{43} + x_{54} + x_{55} + x_{56} + x_{57} + x_{58} + x_{64} + x_{79} + x_{80} + 1$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 19, 21, 24, 25, 30, 31, 32, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{28} + x_{29} + x_{32} + x_{33} + x_{40} + x_{41} + x_{42} + x_{44} + x_{50} + x_{51} + x_{55} + x_{56} + x_{57} + x_{59} + x_{61} + x_{62} + x_{64} + x_{66} + x_{67} + x_{68} + x_{70} + x_{78} + 1$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 19, 21, 24, 25, 30, 31, 33, 36, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{38} + x_{39} + x_{41} + x_{44} + x_{45} + x_{50} + x_{51} + x_{52} + x_{53} + x_{55} + x_{57} + x_{58} + x_{60} + x_{66} + x_{68} + x_{72} + x_{78} + x_{79} + 1$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 19, 21, 24, 25, 31, 33, 36, 38, 40, 41, 43, 45, 47, 48, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{43} + x_{50} + x_{52} + x_{55} + x_{58} + x_{66} + x_{70} + x_{77} + 1$	781
1, 5, 6, 8, 12, 14, 15, 16, 18, 19, 24, 25, 27, 30, 31, 32, 33, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{41} + x_{53} + x_{55} + x_{58} + x_{61} + x_{68}$	781
1, 5, 6, 8, 12, 14, 15, 16, 18, 19, 24, 25, 27, 30, 31, 32, 33, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{29} + x_{41} + x_{42} + x_{53} + x_{55} + x_{56} + x_{58} + x_{61} + x_{64} + x_{66} + x_{67} + x_{68} + x_{69}$	781
1, 5, 6, 8, 12, 14, 15, 16, 18, 19, 24, 25, 27, 30, 31, 32, 33, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_{14} + x_{55} + x_{58} + x_{61} + x_{64} + x_{66} + x_{68} + x_{80}$	781
1, 5, 6, 10, 12, 14, 15, 18, 19, 20, 21, 22, 23, 25, 27, 28, 29, 31, 33, 36, 38, 40, 41, 42, 45, 48, 49, 54, 60, 62, 63, 69, 73, 75, 77, 80	x_{64}	781
5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 25, 27, 29, 31, 33, 36, 38, 40, 41, 42, 45, 48, 49, 54, 60, 62, 63, 69, 73, 75, 77, 80	$x_{66} + 1$	781
1, 2, 3, 5, 6, 7, 8, 12, 13, 14, 15, 16, 19, 21, 23, 27, 33, 36, 38, 40, 43, 45, 47, 52, 54, 56, 58, 60, 62, 69, 70, 71, 73, 74, 76, 79, 80	x_{56}	781
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 33, 36, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 65, 69, 71, 74, 75, 77, 78, 79, 80	$x_{21} + x_{36} + x_{48} + x_{58} + x_{63} + 1$	781
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 33, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 65, 69, 71, 74, 75, 77, 78, 79, 80	$x_{19} + x_{27} + x_{45} + x_{54} + x_{64} + x_{66} + x_{72} + 1$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 18, 19, 24, 25, 30, 31, 32, 33, 36, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 70, 71, 73, 75, 80	$x_5 + x_8 + x_{24} + x_{26} + x_{32} + x_{33} + x_{39} + x_{40} + x_{41} + x_{42} + x_{44} + x_{47} + x_{51} + x_{54} + x_{57} + x_{59} + x_{60} + x_{65} + x_{66} + x_{68} + x_{69} + x_{78} + x_{79}$	781
1, 3, 5, 6, 8, 12, 14, 15, 16, 19, 21, 25, 27, 31, 32, 33, 36, 38, 40, 41, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{25} + x_{52} + 1$	782
1, 5, 6, 7, 8, 10, 12, 14, 15, 19, 21, 24, 25, 27, 31, 36, 38, 39, 40, 41, 45, 47, 49, 53, 56, 58, 62, 63, 66, 69, 71, 73, 77, 80	x_{40}	782
1, 3, 5, 6, 7, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 78	$x_{25} + 1$	782

TABLE 4.1: Maxterms and superpolys after 781 initialization rounds of Trivium (continue).

maxterm bits	superpoly	round
1, 3, 5, 6, 10, 12, 14, 15, 16, 18, 19, 21, 25, 27, 31, 32, 33, 38, 40, 41, 43, 45, 47, 48, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{13} + x_{16} + x_{19} + x_{25} + x_{29} + x_{33} + x_{35} + x_{36} + x_{37} + x_{38} + x_{39} + x_{40} + x_{42} + x_{45} + x_{51} + x_{52} + x_{53} + x_{54} + x_{55} + x_{62} + x_{63} + x_{64} + x_{65} + x_{67} + x_{69} + x_{70} + x_{71} + x_{73} + x_{79} + x_{80} + 1$	782
5, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 34, 36, 38, 40, 45, 47, 48, 49, 53, 54, 56, 58, 60, 62, 63, 69, 71, 80	$x_{38} + 1$	783
3, 5, 6, 7, 8, 10, 14, 15, 16, 21, 23, 25, 27, 33, 34, 36, 38, 40, 45, 47, 48, 49, 53, 54, 55, 56, 58, 61, 62, 63, 69, 71, 74, 80	$x_{27} + 1$	783
1, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 21, 24, 25, 27, 30, 31, 32, 33, 38, 40, 41, 45, 47, 48, 49, 50, 53, 54, 56, 63, 69, 71, 73, 75, 80	$x_{32} + x_{49} + x_{52} + x_{56} + x_{59} + x_{61} + x_{62} + x_{79} + 1$	783
1, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 21, 24, 25, 31, 33, 36, 38, 40, 41, 43, 45, 47, 48, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_7 + x_{16} + x_{40} + x_{43} + x_{49} + x_{52} + x_{58} + x_{62} + x_{70} + x_{79} + 1$	783
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 24, 25, 30, 31, 32, 33, 36, 38, 40, 41, 43, 45, 47, 49, 50, 53, 54, 56, 60, 63, 69, 71, 73, 75, 80	$x_{26} + x_{66} + x_{68} + 1$	783
1, 3, 6, 7, 10, 12, 15, 16, 19, 21, 23, 25, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 65, 69, 71, 75, 77, 80	x_4	784
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 55, 56, 58, 60, 62, 63, 67, 69, 71, 80	$x_{53} + 1$	784
1, 6, 8, 10, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 25, 27, 29, 33, 36, 38, 40, 41, 42, 45, 48, 49, 54, 60, 62, 63, 69, 73, 75, 80	x_{37}	784
3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 60, 62, 63, 69, 71, 74, 80	x_{36}	784
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 29, 31, 33, 36, 38, 40, 41, 45, 47, 49, 53, 58, 60, 63, 71, 75, 76, 80	$x_{12} + 1$	785
1, 3, 6, 7, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 63, 65, 69, 71, 74, 75, 77, 78, 80	x_{34}	785
1, 5, 6, 7, 8, 12, 13, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 52, 54, 56, 58, 60, 62, 69, 71, 73, 79, 80	x_{54}	785
1, 3, 5, 6, 7, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 69, 71, 74, 75, 77, 78, 80	$x_{13} + x_{55} + x_{60} + x_{64}$	785
1, 3, 5, 6, 7, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 65, 69, 71, 74, 75, 77, 78, 80	$x_{22} + x_{49} + x_{64}$	786
1, 3, 6, 7, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 78, 80	$x_{14} + x_{23} + x_{41} + x_{47} + x_{49} + x_{50} + x_{58} + x_{64}$	786
1, 3, 5, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 78, 80	$x_3 + x_4 + x_{20} + x_{22} + x_{30} + x_{34} + x_{38} + x_{40} + x_{42} + x_{45} + x_{49} + x_{51} + x_{58} + x_{61} + x_{65} + x_{67} + x_{69} + x_{72} + x_{78}$	786
1, 3, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 79	$x_9 + x_{29} + x_{30} + x_{32} + x_{42} + x_{43} + x_{49} + x_{51} + x_{57} + x_{58} + x_{59} + x_{60} + x_{62} + x_{64} + x_{66} + x_{67} + x_{68} + x_{69} + x_{70} + x_{72} + x_{76}$	791
1, 3, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 80	$x_{17} + x_{26} + x_{30} + x_{32} + x_{41} + x_{43} + x_{47} + x_{57} + x_{62} + x_{65} + x_{66} + x_{70} + x_{72} + x_{74} + 1$	791
1, 3, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 80	$x_{14} + x_{17} + x_{26} + x_{30} + x_{43} + x_{47} + x_{50} + x_{57} + x_{58} + x_{59} + x_{65} + x_{70} + x_{72} + x_{74} + x_{77} + 1$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 56, 58, 63, 65, 69, 71, 75, 77, 79	$x_{12} + x_{26} + x_{30} + x_{39} + x_{41} + x_{45} + x_{47} + x_{57} + x_{58} + x_{59} + x_{62} + x_{64} + x_{74} + x_{76} + 1$	791
1, 3, 5, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 65, 69, 71, 75, 77	$x_5 + x_{18} + x_{20} + x_{26} + x_{28} + x_{29} + x_{30} + x_{31} + x_{32} + x_{41} + x_{42} + x_{44} + x_{50} + x_{51} + x_{56} + x_{57} + x_{62} + x_{64} + x_{67} + x_{69} + x_{70} + x_{71} + x_{74} + x_{77} + x_{78} + 1$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77	$x_1 + x_{28} + x_{32} + x_{47} + x_{58} + x_{59} + x_{62} + x_{64} + x_{74}$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77, 80	$x_{10} + x_{11} + x_{12} + x_{13} + x_{15} + x_{17} + x_{19} + x_{20} + x_{29} + x_{31} + x_{32} + x_{33} + x_{37} + x_{39} + x_{40} + x_{41} + x_{42} + x_{44} + x_{46} + x_{48} + x_{49} + x_{50} + x_{53} + x_{57} + x_{60} + x_{67} + x_{70} + x_{71} + x_{76} + x_{78} + x_{79}$	791
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 29, 31, 33, 36, 38, 40, 41, 42, 45, 47, 49, 53, 58, 63, 69, 71, 72, 76, 79, 80	x_{61}	791
1, 3, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 69, 71, 74, 75, 77, 78, 80	$x_{43} + x_{47} + x_{58} + x_{70} + x_{74} + 1$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 80	$x_{12} + x_{17} + x_{26} + x_{27} + x_{29} + x_{30} + x_{32} + x_{40} + x_{43} + x_{45} + x_{46} + x_{49} + x_{53} + x_{54} + x_{56} + x_{59} + x_{62} + x_{64} + x_{65} + x_{67} + x_{69} + x_{72} + x_{74} + x_{75}$	792
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 63, 69, 71, 75, 77, 78, 79, 80	$x_{12} + x_{14} + x_{26} + x_{30} + x_{40} + x_{41} + x_{47} + x_{48} + x_{56} + x_{66} + x_{67} + x_{68} + x_{74} + x_{75} + 1$	792
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 63, 69, 71, 74, 75, 77, 78, 79, 80	$x_{16} + x_{43} + x_{56}$	792
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 65, 69, 71, 75, 77, 78, 79, 80	$x_{14} + x_{16} + x_{26} + x_{29} + x_{30} + x_{41} + x_{45} + x_{55} + x_{56} + x_{59} + x_{62} + x_{64} + x_{66} + x_{68} + x_{70} + x_{71} + x_{72} + 1$	792
1, 3, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 69, 71, 75, 77, 80	$x_{45} + x_{72}$	793
1, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 30, 31, 33, 38, 40, 43, 45, 47, 49, 51, 52, 56, 58, 63, 67, 69, 71, 73, 77, 80	$x_{10} + x_{55}$	798
1, 3, 5, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 65, 69, 71, 74, 75, 77, 80	$x_{36} + x_{52} + x_{60} + x_{63}$	798
1, 3, 5, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 80	$x_6 + x_{11} + x_{25} + x_{33} + x_{36} + x_{53} + x_{60} + x_{62} + x_{63} + x_{64} + x_{79}$	798

TABLE 4.1: Maxterms and superpolys after 781 initialization rounds of Trivium.

maxterm bits	**superpoly**	**round**
1, 3, 6, 7, 10, 12, 15, 16, 19, 21, 23, 25, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 65, 69, 71, 75, 77, 80	x_4	784
1, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 20, 21, 23, 25, 27, 29, 31, 33, 36, 38, 40, 41, 42, 45, 49, 54, 60, 62, 69, 73, 75, 80	x_{60}	784
5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 67, 69, 71, 80	$x_{56} + 1$	784
1, 3, 5, 6, 7, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 65, 69, 71, 75, 77, 78	$x_2 + x_9 + x_{13} + x_{14} + x_{22} + x_{23} + x_{30} + x_{36} + x_{38} + x_{39} + x_{40} + x_{42} + x_{47} + x_{48} + x_{51} + x_{56} + x_{65} + x_{67} + x_{68} + x_{69} + x_{74} + x_{75}$	784
1, 3, 5, 6, 8, 10, 12, 14, 15, 16, 19, 21, 25, 30, 31, 32, 33, 38, 40, 41, 43, 45, 47, 48, 49, 50, 53, 54, 56, 63, 69, 71, 75, 80	x_{38}	784
1, 3, 6, 7, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 69, 71, 75, 77, 78, 79	$x_{38} + x_{47} + x_{74}$	784
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 55, 56, 58, 60, 62, 63, 67, 69, 71, 80	$x_{53} + 1$	784
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 67, 69, 71, 74, 80	x_{58}	784
1, 6, 8, 10, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 25, 27, 29, 33, 36, 38, 40, 41, 42, 45, 48, 49, 54, 60, 62, 63, 69, 73, 75, 80	x_{37}	784
1, 6, 8, 10, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 25, 27, 29, 33, 36, 38, 40, 41, 45, 48, 49, 54, 56, 60, 62, 69, 73, 75, 77, 80	x_{64}	784
3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 60, 62, 63, 69, 71, 74, 80	x_{36}	784
6, 8, 10, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 25, 27, 29, 33, 36, 38, 40, 41, 45, 48, 49, 54, 56, 60, 62, 63, 69, 73, 75, 77, 80	x_{66}	784
5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 34, 36, 38, 40, 43, 45, 47, 49, 53, 54, 55, 56, 58, 60, 62, 63, 69, 71, 74, 80	$x_{67} + 1$	784
5, 6, 8, 10, 12, 14, 15, 18, 19, 20, 21, 22, 23, 25, 27, 28, 29, 31, 33, 36, 38, 40, 41, 42, 45, 49, 54, 60, 62, 63, 69, 73, 75, 77, 80	x_{62}	784
1, 5, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 34, 36, 38, 40, 43, 45, 47, 48, 49, 53, 54, 56, 58, 60, 61, 62, 63, 69, 71, 74, 80	$x_{69} + 1$	784
3, 5, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 48, 49, 53, 54, 56, 58, 60, 61, 62, 63, 67, 69, 71, 74, 80	x_{40}	784
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 29, 31, 33, 36, 38, 40, 41, 45, 47, 49, 53, 58, 60, 63, 71, 75, 76, 80	$x_{12} + 1$	785
1, 5, 6, 7, 10, 12, 16, 19, 21, 23, 24, 25, 27, 31, 33, 36, 38, 40, 41, 43, 47, 49, 52, 56, 58, 60, 63, 67, 69, 71, 73, 77, 80	x_{42}	785
1, 5, 6, 10, 12, 14, 15, 16, 19, 21, 25, 27, 30, 31, 33, 36, 38, 40, 41, 43, 45, 47, 48, 49, 53, 54, 56, 63, 69, 71, 73, 75, 80	x_{55}	785
1, 3, 6, 7, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 63, 65, 69, 71, 74, 75, 77, 78, 80	x_{34}	785
1, 5, 6, 7, 8, 12, 13, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 52, 54, 56, 58, 60, 62, 69, 71, 73, 79, 80	x_{54}	785
1, 3, 5, 6, 7, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 69, 71, 74, 75, 77, 78, 80	$x_{13} + x_{55} + x_{60} + x_{64}$	785
1, 3, 5, 6, 7, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 65, 69, 71, 74, 75, 77, 78, 80	$x_{22} + x_{49} + x_{64}$	786
1, 3, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 78, 80	$x_3 + x_4 + x_7 + x_{12} + x_{20} + x_{22} + x_{30} + x_{36} + x_{39} + x_{42} + x_{43} + x_{45} + x_{47} + x_{51} + x_{58} + x_{63} + x_{69} + x_{70} + x_{72} + x_{78} + 1$	786
1, 3, 6, 7, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 78, 80	$x_{14} + x_{23} + x_{41} + x_{47} + x_{49} + x_{50} + x_{58} + x_{64}$	786
1, 3, 5, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 78, 80	$x_3 + x_4 + x_{14} + x_{22} + x_{30} + x_{34} + x_{38} + x_{41} + x_{42} + x_{45} + x_{47} + x_{49} + x_{51} + x_{58} + x_{61} + x_{65} + x_{69} + x_{72} + x_{78}$	786
1, 3, 5, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 78, 80	$x_3 + x_4 + x_{20} + x_{22} + x_{30} + x_{34} + x_{38} + x_{40} + x_{42} + x_{45} + x_{49} + x_{51} + x_{58} + x_{61} + x_{65} + x_{67} + x_{69} + x_{72} + x_{78}$	786
1, 3, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 79	$x_9 + x_{29} + x_{30} + x_{32} + x_{42} + x_{43} + x_{49} + x_{51} + x_{57} + x_{58} + x_{59} + x_{60} + x_{62} + x_{64} + x_{66} + x_{67} + x_{68} + x_{69} + x_{70} + x_{72} + x_{76}$	791
1, 3, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 80	$x_{17} + x_{26} + x_{30} + x_{32} + x_{41} + x_{43} + x_{47} + x_{57} + x_{62} + x_{65} + x_{66} + x_{70} + x_{72} + x_{74} + 1$	791
1, 3, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 80	$x_1 + x_{17} + x_{26} + x_{28} + x_{41} + x_{43} + x_{47} + x_{49} + x_{59} + x_{62} + x_{64} + x_{65} + x_{66} + x_{70} + x_{72} + x_{74} + x_{76} + 1$	791
1, 3, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 80	$x_{14} + x_{17} + x_{26} + x_{30} + x_{43} + x_{47} + x_{50} + x_{57} + x_{58} + x_{59} + x_{65} + x_{70} + x_{72} + x_{74} + x_{77} + 1$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 40, 41, 43, 45, 47, 49, 53, 56, 58, 63, 65, 69, 71, 75, 77, 79	$x_{12} + x_{26} + x_{30} + x_{39} + x_{41} + x_{45} + x_{47} + x_{57} + x_{58} + x_{59} + x_{62} + x_{64} + x_{74} + x_{76} + 1$	791
1, 3, 5, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 65, 69, 71, 75, 77	$x_5 + x_{18} + x_{20} + x_{26} + x_{28} + x_{29} + x_{30} + x_{31} + x_{32} + x_{41} + x_{42} + x_{44} + x_{50} + x_{51} + x_{56} + x_{57} + x_{62} + x_{64} + x_{67} + x_{69} + x_{70} + x_{71} + x_{74} + x_{77} + x_{78} + 1$	791
1, 3, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77, 80	$x_7 + x_9 + x_{10} + x_{11} + x_{12} + x_{13} + x_{15} + x_{17} + x_{19} + x_{20} + x_{26} + x_{30} + x_{31} + x_{33} + x_{37} + x_{39} + x_{40} + x_{41} + x_{42} + x_{44} + x_{45} + x_{46} + x_{47} + x_{48} + x_{50} + x_{51} + x_{53} + x_{56} + x_{59} + x_{60} + x_{64} + x_{67} + x_{68} + x_{70} + x_{71} + x_{72} + x_{74} + x_{78} + x_{79} + 1$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77	$x_1 + x_{28} + x_{32} + x_{47} + x_{58} + x_{59} + x_{62} + x_{64} + x_{74}$	791

TABLE 4.2: Maxterms and superpolys after 784 initialization rounds of Trivium (continue).

maxterm bits	**superpoly**	**round**
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77, 80	$x_3 + x_4 + x_6 + x_7 + x_9 + x_{10} + x_{13} + x_{15} + x_{19} + x_{22} + x_{28} + x_{30} + x_{33} + x_{34} + x_{35} + x_{38} + x_{39} + x_{40} + x_{41} + x_{43} + x_{44} + x_{47} + x_{48} + x_{49} + x_{50} + x_{53} + x_{54} + x_{55} + x_{56} + x_{58} + x_{61} + x_{62} + x_{65} + x_{66} + x_{67} + x_{68} + x_{69} + x_{71} + x_{72} + x_{76} + x_{77} + x_{78}$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77, 80	$x_{10} + x_{11} + x_{12} + x_{13} + x_{15} + x_{17} + x_{19} + x_{20} + x_{29} + x_{31} + x_{32} + x_{33} + x_{37} + x_{39} + x_{40} + x_{41} + x_{42} + x_{44} + x_{46} + x_{48} + x_{49} + x_{50} + x_{53} + x_{57} + x_{60} + x_{67} + x_{70} + x_{71} + x_{76} + x_{78} + x_{79}$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77, 80	$x_{10} + x_{11} + x_{12} + x_{13} + x_{14} + x_{15} + x_{17} + x_{19} + x_{20} + x_{26} + x_{29} + x_{31} + x_{33} + x_{37} + x_{39} + x_{40} + x_{42} + x_{44} + x_{46} + x_{48} + x_{53} + x_{57} + x_{58} + x_{59} + x_{60} + x_{66} + x_{67} + x_{68} + x_{70} + x_{71} + x_{72} + x_{77} + x_{78} + x_{79} + 1$	791
1, 3, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77, 79	$x_3 + x_4 + x_6 + x_{11} + x_{15} + x_{17} + x_{19} + x_{20} + x_{22} + x_{30} + x_{34} + x_{35} + x_{37} + x_{38} + x_{43} + x_{47} + x_{51} + x_{54} + x_{57} + x_{58} + x_{60} + x_{61} + x_{64} + x_{65} + x_{67} + x_{68} + x_{70} + x_{72} + x_{74} + x_{77} + x_{79} + 1$	791
1, 3, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77, 79	$x_3 + x_4 + x_6 + x_{11} + x_{12} + x_{15} + x_{17} + x_{18} + x_{19} + x_{20} + x_{22} + x_{30} + x_{34} + x_{35} + x_{37} + x_{38} + x_{39} + x_{42} + x_{43} + x_{45} + x_{47} + x_{50} + x_{54} + x_{56} + x_{57} + x_{58} + x_{61} + x_{65} + x_{69} + x_{70} + x_{74} + x_{78} + x_{79} + 1$	791
1, 3, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 69, 71, 74, 75, 77, 78, 80	$x_1 + x_5 + x_9 + x_{14} + x_{18} + x_{20} + x_{26} + x_{28} + x_{32} + x_{41} + x_{42} + x_{43} + x_{45} + x_{47} + x_{49} + x_{66} + x_{67} + x_{69} + x_{70} + x_{76} + x_{78}$	791
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 29, 31, 33, 36, 38, 40, 41, 42, 45, 47, 49, 53, 58, 63, 69, 71, 72, 76, 79, 80	x_{61}	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 75, 77, 79	$x_3 + x_4 + x_6 + x_7 + x_9 + x_{11} + x_{17} + x_{18} + x_{20} + x_{22} + x_{26} + x_{28} + x_{29} + x_{31} + x_{34} + x_{35} + x_{37} + x_{38} + x_{39} + x_{41} + x_{46} + x_{50} + x_{51} + x_{54} + x_{55} + x_{56} + x_{58} + x_{61} + x_{65} + x_{66} + x_{67} + x_{74} + x_{76} + x_{78} + x_{79} + 1$	791
1, 3, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 69, 71, 74, 75, 77, 78, 80	$x_{43} + x_{47} + x_{58} + x_{70} + x_{74} + 1$	791
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 41, 43, 45, 47, 49, 53, 56, 58, 62, 63, 69, 71, 75, 77, 80	$x_{12} + x_{17} + x_{26} + x_{27} + x_{29} + x_{30} + x_{32} + x_{40} + x_{43} + x_{45} + x_{46} + x_{49} + x_{53} + x_{54} + x_{56} + x_{59} + x_{62} + x_{64} + x_{65} + x_{67} + x_{69} + x_{72} + x_{74} + x_{75}$	792
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 63, 69, 71, 75, 77, 78, 79, 80	$x_{12} + x_{26} + x_{39} + x_{56} + x_{68} + 1$	792
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 56, 58, 61, 63, 69, 71, 75, 77, 78, 79, 80	$x_{12} + x_{14} + x_{26} + x_{30} + x_{40} + x_{41} + x_{47} + x_{48} + x_{56} + x_{66} + x_{67} + x_{68} + x_{74} + x_{75} + 1$	792
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 63, 69, 71, 74, 75, 77, 78, 79, 80	$x_{16} + x_{43} + x_{56}$	792
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 65, 69, 71, 75, 77, 78, 79, 80	$x_{14} + x_{16} + x_{26} + x_{29} + x_{30} + x_{41} + x_{45} + x_{55} + x_{56} + x_{59} + x_{62} + x_{64} + x_{66} + x_{68} + x_{70} + x_{71} + x_{72} + 1$	792
1, 3, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 41, 43, 45, 47, 49, 53, 54, 56, 58, 61, 63, 69, 71, 75, 77, 80	$x_{45} + x_{72}$	793
1, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 27, 30, 31, 33, 38, 40, 43, 45, 47, 49, 51, 52, 56, 58, 63, 67, 69, 71, 73, 77, 80	$x_{10} + x_{55}$	798
1, 3, 5, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 65, 69, 71, 74, 75, 77, 80	$x_{36} + x_{52} + x_{60} + x_{63}$	798
1, 3, 5, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 65, 69, 71, 74, 75, 77, 80	$x_{10} + x_{17} + x_{27} + x_{36} + x_{37} + x_{40} + x_{52} + x_{59} + x_{60} + x_{63} + x_{66} + x_{67}$	798
1, 3, 5, 6, 7, 8, 10, 12, 14, 16, 19, 21, 23, 25, 33, 36, 38, 40, 43, 45, 47, 49, 53, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 78, 79	$x_{27} + x_{54} + x_{60}$	798
1, 3, 5, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 80	$x_6 + x_{11} + x_{25} + x_{33} + x_{36} + x_{53} + x_{60} + x_{62} + x_{63} + x_{64} + x_{79}$	798
1, 3, 5, 6, 7, 8, 10, 12, 15, 16, 19, 21, 23, 25, 27, 33, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 61, 62, 63, 65, 69, 71, 74, 75, 77, 80	$x_6 + x_{11} + x_{25} + x_{33} + x_{36} + x_{52} + x_{53} + x_{60} + x_{62} + x_{63} + x_{64} + x_{79}$	798
5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 34, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 60, 61, 62, 63, 67, 69, 71, 74, 80	$x_{65} + x_{66} + x_{67} + 1$	798
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 54, 56, 58, 62, 69, 71, 73, 80	$x_{25} + 1$	799
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 56, 58, 63, 69, 71, 75, 77, 80	$x_{12} + x_{38} + x_{39} + x_{40}$	799

TABLE 4.2: Maxterms and superpolys after 784 initialization rounds of Trivium.

maxterm bits	superpoly	round
1, 6, 8, 10, 12, 14, 18, 20, 23, 25, 27, 31, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 69, 73, 75, 77, 80	x_{60}	799
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 54, 56, 58, 62, 69, 71, 73, 80	$x_{25} + 1$	799
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 56, 58, 63, 69, 71, 75, 77, 80	$x_{25} + x_{40}$	799
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 40, 43, 45, 47, 49, 53, 56, 58, 63, 69, 71, 75, 77, 80	$x_{12} + x_{38} + x_{39} + x_{40}$	799
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 18, 20, 23, 25, 27, 33, 36, 38, 40, 41, 43, 47, 49, 53, 56, 58, 63, 69, 71, 75, 77, 80	$x_{67} + 1$	799
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 31, 33, 36, 38, 40, 41, 45, 47, 49, 53, 56, 58, 63, 69, 71, 75, 80	x_{42}	799
1, 5, 6, 7, 8, 10, 12, 14, 15, 19, 21, 23, 25, 27, 31, 36, 38, 40, 41, 45, 47, 49, 53, 56, 58, 63, 69, 71, 73, 75, 77, 80	x_{53}	799
1, 5, 6, 8, 10, 12, 14, 15, 16, 18, 19, 20, 21, 23, 25, 27, 31, 33, 36, 38, 40, 41, 45, 49, 54, 56, 62, 69, 73, 75, 77, 80	x_{64}	799
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 69, 71, 80	$x_{36} + 1$	799
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 40, 41, 43, 45, 47, 49, 53, 56, 58, 63, 65, 69, 71, 75, 77, 80	x_{38}	799
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 54, 56, 58, 60, 62, 65, 69, 70, 71, 73, 80	x_{56}	799
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 33, 34, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 69, 71, 74, 80	$x_{69} + 1$	799
1, 6, 7, 8, 10, 12, 14, 16, 19, 21, 25, 27, 30, 31, 33, 36, 38, 40, 41, 43, 45, 47, 49, 51, 52, 56, 58, 63, 67, 69, 71, 73, 77, 80	$x_{66} + 1$	799
1, 3, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 25, 27, 33, 34, 36, 38, 40, 43, 45, 47, 49, 53, 54, 56, 58, 62, 63, 67, 69, 71, 74, 80	x_{58}	799
1, 5, 6, 7, 8, 10, 12, 14, 15, 16, 19, 21, 23, 25, 27, 33, 36, 38, 40, 43, 45, 47, 49, 53, 54, 55, 56, 58, 62, 63, 67, 69, 71, 74, 80	x_{37}	799

Table 4.3: Maxterms and superpolys after 799 initialization rounds of Trivium.

maxterm bits	superpoly	round
0, 5, 6, 7, 9, 11, 13, 17, 19, 20, 22, 24, 26, 30, 32, 33, 35, 37, 39, 42, 44, 46, 48, 52, 55, 57, 61, 62, 66, 68, 72, 74, 76, 79	x_{63}	800

Table 4.4: Maxterms and superpolys after 800 initialization rounds of Trivium.

5 A Data-Driven Framework for Adaptive Libraries

5.1 Introduction

Mainstream High Performance Computing (HPC) scientific applications are built around monolithic fast parallel routines, that in most cases are customized for a specific target architecture. With the advent of Big Data era and data-driven applications such as graph analytics or image recognition, the traditional design of parallel routines does not provide performance portability mainly due to unpredictable size or structure of the data. As an example, due to the ubiquity of Matrix Multiplications in many scientific applications, Basic Linear Algebra Subprograms (BLAS) and, in particular, Generic Matrix Multiplication (GEMM) routines are the main target of optimizations. Several BLAS implementations provide outstanding performance on a target architecture by assuming a fixed data size or structure (i.e., square matrices) [61–63]. However, the matrices involved in the training of Deep Learning frameworks, for example, expose different sizes and shapes (usually strongly rectangular) [64]. As a consequence, it is hard to find a good optimization which takes into account the wide range of data sizes involved. In practice, most of BLAS libraries often provide several GEMM implementations for specific input characteristics. Such user-transparent implementations are selected by trivial heuristics based on customized decision rules. Moreover, with the wide variety of parallel architectures available on the market ranging from traditional parallel processors to accelerators (GPUs, FPGA) and System on Chip architectures (SoC), the development of portable, high performance code has become extremely challenging. Parametric implementations and auto-tuning techniques have partially mitigated the problem by reducing the problem of performance portability by adapting the underlying memory hierarchies and/or data thread mapping to a specific parallel architecture. Within this context, a plethora of hardware-oblivious BLAS libraries has been developed

[22, 65]. This work aims at offering a new prospective on adaptive libraries and performance portability. Focusing on GPU architectures, we present a new framework based on a predictive model to select the optimal algorithm and the related tuning parameters in order to improve the performance of data-driven applications. The contributions of this work are manifold:

- we describe and provide three different training datasets generated from a tunable BLAS library for GEMM routines on GPUs;
- we analyze several configurations of *Decision Trees*, a simple univariate supervised classifier, in order to predict an optimized GEMM implementation and related tuned parameters;
- we describe a suitable framework and workflow able to build the decision rule inside BLAS library;
- we validate our study by providing exhaustive experimental results where we evaluate several metrics ranging from the accuracy of each model to the overhead of the decision rules passing through a detailed analysis of the performance;
- we define two metrics to experimentally evaluate the quality of the models generated by our framework;
- the integration of our solution in a OpenCL BLAS library, CLBlast [22], shows up 3x and 2.5x of speed-up both on high-end NVIDIA GPU architectures and on a low-power embedded ARM Mali GPU.

5.2 Background

We first introduce the notation and basic concepts used hereunder starting from the *matrix multiplication problem* definition. Then, we provide a concise description of the CLBlast library. Finally, we introduce in a simplified form the concept of Decision Trees classifier.

5.2.1 Generic Matrix Multiplication

The matrix-multiplication is one of the key computational kernels of traditional scientific applications and more recently of deep learning and other Machine Learning algorithms. For instance almost all deep neural networks use dense multiplication for implementing fully connected layers. BLAS libraries define the general matrix multiplication as follows:

$$C = \alpha * A * B + \beta * C \quad s.t. \quad A \in \mathbb{C}^{M\text{x}K}, B \in \mathbb{C}^{K\text{x}N}, C \in \mathbb{C}^{M\text{x}N} \tag{5.1}$$

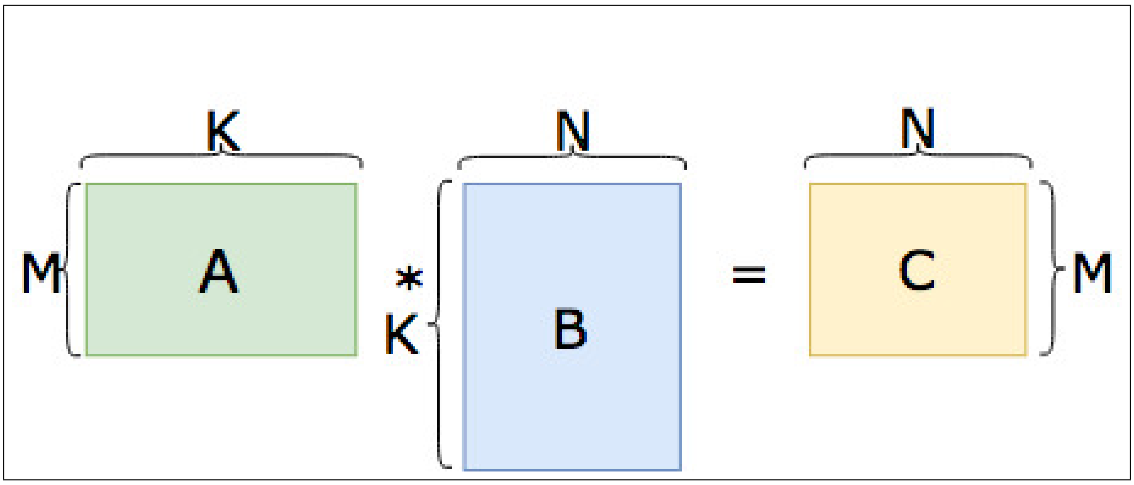


FIGURE 5.1: An example of matrix multiplication where $\alpha = 1$ and $\beta = 0$.

where A and B are the input matrices, C is the output and α and β are constants [66]. Furthermore, A and B can be either or both optionally transposed. In general, a matrix multiplication is represented by the triple (M, N, K) describing the sizes of the matrices involved (see Figure 5.1).

The naive algorithm sequentially calculates each element of C by using three nested loops. This algorithm solves the problem in $O(n^3)$ time step [67], assuming A and B are square matrices of size $n \times n$. However, in practice, the actual performance varies depending on the maximization of data-reuse and the minimization of latency. In general, parameters such as tiling, threads organization and scheduling influence the final performance [68]. For example, for most target architectures different values of tiles strongly impact on data-reusing. Tuners explore a huge search space looking for parameters that offer the best performance for a specific input size (M, N, K) and architecture. Notable solutions and techniques about BLAS libraries and auto-tuning are reported in the related work Section (5.3).

5.2.2 CLBlast Library

CLBlast is a modern, lightweight, fast and tunable OpenCL BLAS library written in C++11 [69]. It is designed to leverage the full performance potential of a wide variety of OpenCL devices from different vendors, including desktop and laptop GPUs, embedded GPUs, and other accelerators. The library implements BLAS routines: basic linear algebra subprograms operating on vectors and matrices. Specifically to GEMM, at a high level, there are two kernels to choose from: a 'direct' kernel covering all GEMM use-cases, and an 'in-direct' kernel making several assumptions about the layout and sizes of the matrices. The 'in-direct' kernel can thus not be used on its own and requires several helper kernels to pad and/or transpose matrices to meet those assumptions. Thus, there is a trade-off between running the more generic slower 'direct' kernel versus a faster 'in-direct' kernel ($O(n^3)$) plus several helper kernels ($O(n^2)$). Furthermore, at a kernel level, there are many tunable parameters. They include varying the work-group size, the amount of work per thread or work-group, the vector width of computation and loads/stores, the use of local memory for caching, and so on. In total the search space for a GEMM kernel can easily grow to a hundred thousand realistic combinations. For more details we refer to the CLBlast and CLTune papers [22, 70].

5.2.3 Decision Tree Classifier

Decision Trees are a non-parametric supervised learning method used for classification and regression [71, 72]. The aim is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. Some advantages of Decision Trees are:

- simple to understand and to interpret;
- requires little data preparation;
- most of the operation in a Decision Tree are logarithmic in the number of data points used to train the tree;
- uses a white box model, unlike (e.g., in an artificial neural network), results may be more difficult to interpret;
- easy code translation. Decision Trees naturally encode IF-THEN-ELSE statements.

On the other hand, Decision Trees exhibit several disadvantages:

- Decision Tree learners can create over-complex trees that do not generalise the data well (overfitting);
- it can be unstable because small variations in the data might result in a completely different tree being generated;
- the problem of learning an optimal Decision Tree is known to be NP-complete. Consequently, practical algorithms cannot guarantee to return the actual optimal Decision Tree (low accuracy).

We rely on the Python library scikit-learn to build several Decision Trees [20]. It implements an optimized version of the CART algorithm. This library provides several parameters in order to define different split criteria (e.g., the maximum high of tree), minimum number of samples required to split of an internal node and other metrics (e.g., Gini impurity).

5.3 Related Work

Several optimized linear algebra and BLAS libraries have been released [69, 73–75] in the past. Some of them have been designed for accelerators [22, 76, 77] or for specific GPU architectures only [62]. Several works have previously published auto-tuning [78] and optimization approaches to accelerate GEMM [79–81]. The problem of the exploration of huge search space of tunable parameters has been partially mitigated by the use of meta-heuristics optimization approaches [70] and Machine Learning techniques (e.g., regression). The formers are able to predict parameters by starting from the exploration of a small search space[82–84]. Input-aware auto-tuning arose recently [85, 86] as a way to address the problem of performance portability on data-driven applications [87, 88]. From industrial prospective, the vendors libraries (e.g., MKL [63], cuBLAS [62]) still use manual heuristics in order to select at runtime highly-optimized code for specific inputs. A notable recent approach, called ISAAC [89], exploits multi-layer perceptron (MLP) to generate high optimized parametric-code in the training step. Then, at runtime, the library infers the best parameters for the specific input.

6 Data-Driven Framework Description

In the present Chapter, we introduce a new framework for the generation of a data-driven runtime. We identify three desirable characteristics of the framework. First, it should be able to select the best routine among multiple possible choices according to a specific metrics for a given input. For example, let $\mathcal{A}$ be a finite set of the implementations which solve a specific problem (i.e. Matrix Multiplication), let f be a specific metric (i.e. Floating Point Operation per Second, FLOPS), and let I be an input domain for $\mathcal{A}$ (i.e. the set of all triples (M, N, K)); the framework should find $\overline{a} = \max_a f(a_i)$ where $a \in \mathcal{A}$ for each domain $i \in I$ of $\mathcal{A}$. Second, the framework should be able to build a predictive model starting from the set (training-set) of the collected $\overline{a}$. Third, the framework should be able to auto-generate the code which implements the model. The model and the code auto-generated should also satisfy the following requirements:

1. **soundness**: the model should work on the same input domain of the original library;
2. **cost-aware**: the auto-generated code should have negligible overhead. In other words, the cost to select the best routine must be lower than the improvement. Formally, $f(a) < f(\overline{a} + c)$ where c is the cost to select a.

From a implementation point of view, we decide to use the simplest classifier *Decision Trees* to build the predictive model. Specifically, we use the scikit-learn [20] implementation of them. Although Decision Trees, theoretically speaking, may provide low accuracy in several applications, as described in 5.2, they are easy to analyze. Furthermore, they can be naturally translated in IF-THEN-ELSE statements. We now provide the description of the framework from an architectural point of view and then, in Chapter 7.1, we analyze several predictive models and the experimental results.

6.1 Architectural Design

The framework is composed by two different parts, as depicted in Figure 6.1. In the first one, the training set (optionally) and the predictive model are built offline. The reason is that those operations are computationally expensive. In the end of the offline phase, a complex IF-THEN-ELSE statement is defined and plugged into the target library. In other words, the offline phase is responsible for collecting the dataset (if needed), building the model, extracting the decision rules from the model and use them to automatically generate the corresponding source code function F implementing the rules. As last step, F has to be plugged on top of the target library L and L has to be re-built.

We now describe more in detail all the actions and components composing our framework starting from those belonging to the offline phase.

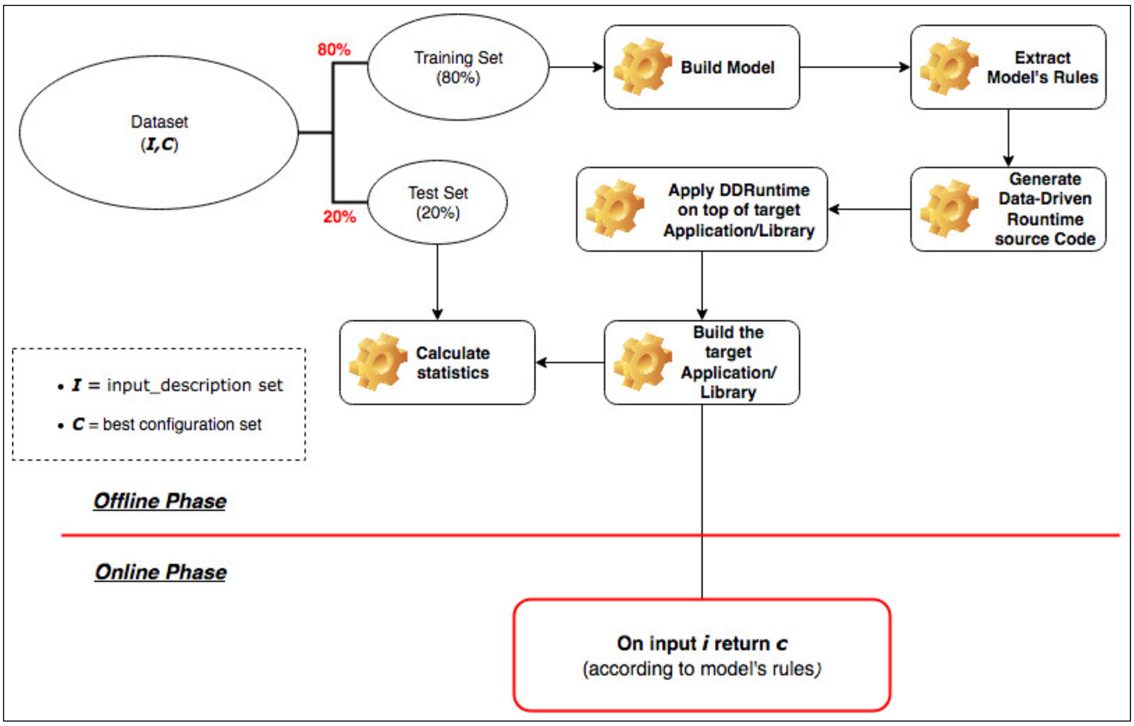

80%
Training Set (80%)
Build Model
Extract Model's Rules
Dataset (I,C)
Test Set (20%)
20%
Apply DDRuntime on top of target Application/Library
Generate Data-Driven Rountime source Code
Calculate statistics
Build the target Application/ Library
• I = input_description set
• C = best configuration set
Offline Phase
Online Phase
On input i return c
(according to model's rules)

FIGURE 6.1: The data-driven framework architecture.

The Datasets Generation

The generation of the datasets is the most expensive task in our framework. The dataset D is a collection of records, where each entry is composed by a pair (I, C): I is the input description and C is the corresponding class. The input description is an object containing several information about the input: the size (for instance the triple (M, N, K) for a matrix multiplication), the structure, and any further information or metrics that can characterise the input (i.e. the density of a matrix or the matrix layout). In our case, a class represents the *best* configuration for a given input according to a selected metrics. For example, for the matrix multiplication problem, a possible metrics is the number of floating point operations per second (FLOPS) whereas, C can represent the best (in terms of FLOPS) GEMM routine and related tuning parameters. As a note, since collecting the dataset is the most time consuming step, a collaborative/community-driven approach can easily provide several datasets [90]. Each dataset D is then divided into two disjoint subsets X and Y such that $D = \{X\} \cup \{Y\}$. The subsets X and Y are respectively called the *Training set* and the *Test set* of D and they correspond, respectively, to 80% and 20% of the whole dataset D. The training sets are used to build the models whereas the corresponding test sets to evaluate them.

The Model Building and the Generation of the Runtime

The next step, once the datasets are available, is the generation of the model. We use each training subset X to create the corresponding models. In order to generate a Decision Tree Classifier we have to identify the set of *features* and *labels* in the training set; we select as features the input descriptions I and the configuration descriptions C as labels. After the generation of the Decision Tree, the system automatically traverses it and extracts all the rules defined in the tree internal nodes and all the selected configurations that are the tree leaves. It then produces the source code of the function F, representing the Decision Tree, that we use at runtime to choose a configuration. Once F has been generated, it is plugged into the target library, and then the system is responsible of re-building the library to enable the runtime.

The Online Phase

In the online phase, the target library is ready to be used. It transparently provides the data-driven adaptation mechanism to any application that relies on the library. Our experiments, that are described in detail in Chapter 7.1, show that the proposed solution may provide up to 3× performance improvement over the traditionally optimized and tuned CLBlast library on two very different architectures. Moreover, the experiments show that the overhead of choosing the best configuration for each input impacts less than 2% in the worst case.

7 Proof Of Concept on Matrix Multiplication

We present a *proof of concept* for matrix multiplication that shows the effectiveness of our framework. We focused on GEMM routines as they are a building block for several applications ranging from Machine Learning to Cryptography. We selected execution time as reference metrics because we were interested in the optimisation of Machine Learning frameworks that heavily rely on Matrix Multiplication. We chose as target library CLBlast [22] (version 0.10.0), an OpenCL library that implements BLAS routines by exposing several tunable parameters. We extended some functionalities of the original CLBlast to support multiple configurations for the same kernel and we also developed a new client to easily validate a new configuration without rebuilding the library. Moreover, we used CLTune [70], to discover the best configurations for xgemm and xgemm_direct kernels for large set of matrices. CLTune allows for defining the search-space for all the tunable parameters and to select the search strategy. We customized the default search space of xgemm and xgemm_direct to the target architectures to maximize the performance of the two routines. We decided to exhaustively explore the search space as this guarantees the tuner always selects the best possible configuration respect to the search space. Moreover, search strategies based on heuristics may introduce a probability distribution on the search space that may negatively influence the prediction model.

We relied on the Collective Knowledge framework [90] for collecting the datasets, generating several Decision Trees and evaluating their performance.

We created three datasets called *AntonNet*, *PowerOf2 (po2)* and *GridOf2 (go2)*. The entries of each dataset are composed by a triple (M, N, K) representing the size of the matrices involved in the multiplication and the corresponding best configuration we discovered with the tuner.

The first dataset, *AntonNet*, contains all the matrix sizes collected from three widely known neural networks, AlexNet [91], GoogleNet [23] and SqueezeNet1.1 [92]. We collected the matrices where batch size ranging from 2 up to 128, and it is incremented every 2 steps.

We propose this dataset since we think it can be representative: for instance the collected matrices expose different sizes and shapes and they are usually highly rectangular (see Tables A.1, A.2 and, A.3 in Appendix A). Moreover, we identified a large number of unique best configurations respect to the number of entries in this dataset. This is probably due to the fact that the matrices sizes are irregular.

The second dataset *go2* is composed by all the matrices where M, N and K have values ranging from 256 to 3840 and the value has to be a multiple of 256. This dataset contains approximately 10 times the number of entries with respect to *AntonNet*. However the number of unique best configurations is about one third with respect to the *AntonNet* case.

The last dataset, *po2* collects the best configurations for all the matrices having M, N and K that are power of 2 ranging from 64 to 2048. We understand that those matrices may not be the best choice as they represent a special case. However, we could not collect the *go2* dataset on both the architectures we tested due to time constraints. For that reason, we decided to collect *po2* to evaluate at least two datasets for each architecture. A detailed summary of the three datasets for both the tested architectures (Nvidia Pascal and ARM Mali Midgard) is reported in Table 7.4 and Table 7.5.

We divided each dataset in the two disjoint subsets X (training-set) and Y (test-set) as described in Section 6.1. We then used the X sets to build the models by using the scikit-learn library.

In order to evaluate the accuracy and the effectiveness of our approach, we generated several Decision Trees where we tuned some tree parameters exposed by the scikit-learn library, like the maximum tree depth and the minimum samples to split an internal node. Moreover, we developed a Python script to extract other features from the generated Decision Tree, like for example the number of leafs and the tree height, as those information are not directly available. These statistics helped us to better understand the model and the experimental results. The same Python script is responsible of traversing the Decision Trees and extracting from them all the rules defined in the internal nodes and all the configurations that are the tree leaves.

Consequently the script uses the extracted information to automatically generate the C++ source code for the function F that implements the extracted model. This function is responsible for the selection at runtime of the best configuration according to the input. Then the framework inserts the source code of F into our extended version of CLBlast. It also adds the configurations extracted from the decision tree to the CLBlast database and then it builds the library.

An example of one Decision Tree generated by the proposed framework is depicted in Figure 7.1.

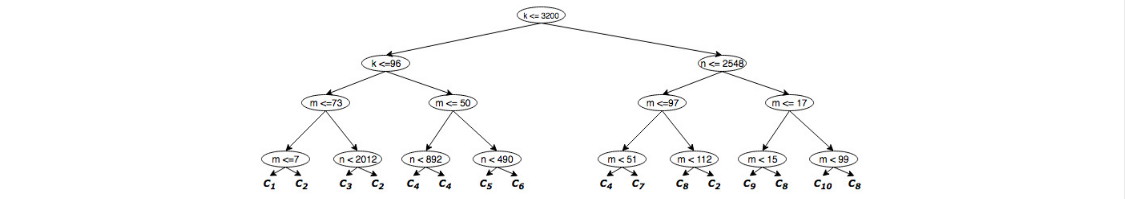


FIGURE 7.1: An example of one Decision Tree generated by the proposed framework.

7.1 Experimental Results

7.1.1 Experimental Setup

We evaluated the proof of concept based on several runtime generated by our framework on two different GPUs representing architectures quite different each other. The first device is the Nvidia Tesla P100 with 16GB of memory, based on Nvidia Pascal architecture, an accelerator for HPC systems and datacenter. The second device is the Mali-T860 GPU based on ARM Midgard architecture. More details on both the GPUs can be found in Table 7.1.

	Nvidia P100	**ARM Mali-T860**
Market segment	Server	System on Chip
Micro-architecture	Pascal	Midgard 4th gen
Number of available cores	3584 Cuda Core (GP100)	2 Cortex-A72 and 4 Cortex-A53
Boost frequency	1353 MHz	2000 MHz
Processing Power	9.7 TFLOPS	23.8 GFLOPS
Memory available	16 GB	4 GB
Memory type	HBM2	DDR3

TABLE 7.1: Tested GPUs characteristics - data-driven framework.

7.1.2 Experiments

We conducted an extensive experiments campaign in order to evaluate the runtime generated by our framework. The experiments aimed at investigating the following aspects:

- the quality of the models in terms of accuracy;
- the quality of the models experimentally evaluated;
- the overhead of the runtime generated by our framework;
- the performance comparison of the CLBlast library that uses our runtime against:

Decision Tree Name	**Accuracy (%)**	**DTPR**	**DTTR**	**Total number of Leaves**	**Decision Tree Height**	**Min Samples PerSplit**	**Number of Unique Config. Gemm**	**Number of Unique Config. GemmDirect**	**Number of Leaves Gemm**	**Number of Leaves GemmDir**
h1-L1	62	0.376	0.637	2	1	1	1	1	1	1
h1-L2	62	0.376	0.637	2	1	2	1	1	1	1
h1-L4	62	0.376	0.637	2	1	4	1	1	1	1
h1-L0.1	62	0.376	0.637	2	1	0.1	1	1	1	1
h1-L0.2	62	0.376	0.637	2	1	0.2	1	1	1	1
h1-L0.3	59	0.436	0.736	2	1	0.3	0	2	0	2
h1-L0.4	56	0.444	0.735	2	1	0.4	1	1	1	1
h1-L0.5	51.5	0.433	0.734	1	0	0.5	0	1	0	1
h2-L1	62	0.433	0.734	4	2	1	1	2	2	2
h2-L2	62	0.416	0.703	4	2	2	1	2	2	2
h2-L4	62	0.415	0.702	4	2	4	1	2	2	2
h2-L0.1	62	0.415	0.702	4	2	0.1	1	2	2	2
h2-L0.2	62	0.416	0.703	3	2	0.2	1	2	1	2
h2-L0.3	59	0.416	0.982	3	2	0.3	0	3	0	3
h2-L0.4	56	0.606	0.736	2	1	0.4	1	1	1	1
h2-L0.5	51.5	0.445	0.734	1	0	0.5	0	1	0	1
h4-L1	67	0.687	1.120	16	4	1	1	5	2	14
h4-L2	67	0.688	1.122	16	4	2	1	5	2	14
h4-L1	67	0.686	1.119	16	4	4	1	5	2	14
h4-L0.1	65.5	0.576	0.931	8	4	0.1	1	4	2	6
h4-L0.2	62	0.506	0.845	4	3	0.2	1	3	1	3
h4-L0.3	59	0.605	0.981	3	2	0.3	0	3	0	3
h4-L0.4	56	0.445	0.737	2	1	0.4	1	1	1	1
h4-L0.5	51.5	0.434	0.735	1	0	0.5	0	1	0	1
h8-L1	67	0.806	1.340	215	8	1	1	9	4	211
h8-L2	66.5	0.807	1.341	201	8	2	1	8	4	197
h8-L4	66	0.806	1.304	175	8	4	1	6	4	171
h8-L0.1	65.5	0.576	0.931	8	4	0.1	1	4	2	6
h8-L0.2	62	0.506	0.845	4	3	0.2	1	3	1	3
h8-L0.3	59	0.606	0.982	3	2	0.3	0	3	0	3
h8-L0.4	56	0.445	0.736	2	1	0.4	1	1	1	1
h8-L0.5	51.5	0.433	0.734	1	0	0.5	0	1	0	1
hMax-L1	**60**	**0.852**	**1.424**	**1290**	**19**	**1**	**1**	**11**	**4**	**1286**
hMax-L2	58.5	0.848	1.418	790	18	2	1	8	4	786
hMax-L4	64	0.846	1.412	430	15	4	1	6	4	426
hMax-L0.1	65.5	0.574	0.927	8	4	0.1	1	4	2	6
hMax-L0.2	62	0.506	0.844	4	3	0.2	1	3	1	3
hMax-L0.3	59	0.606	0.982	3	2	0.4	0	3	0	3
hMax-L0.4	56	0.445	0.737	2	1	0.4	1	1	1	1
hMax-L0.5	51.5	0.433	0.734	1	0	0.5	0	1	0	1

TABLE 7.2: Decision Trees on go2 dataset - P100. The row corresponding to the *best* Decision Tree for this dataset and architecture is highlighted.

- the auto-tuned version of CLBlast;
- the peak performance of the tuner.

We ran the experiments on all the datasets and for both the target architectures. We evaluated several decision trees where we specified different values for the maximum height and/or the minimum number of samples to split an internal node. In particular we defined H, the set of possible values that the depth of the tree can assume, as $H = \{1, 2, 4, 8, Max\}$, where Max means we do not specify any limitation, and the set L of possible values for the minimum number of samples as $L = \{1, 2, 4, 0.1, 0.2, 0.4, 0.5\}$ where the decimal values represent the percentage of the samples.

To estimate the quality of the models, we calculated the accuracy by using scikit-learn. The accuracy provides a measure of the percentage of right predictions the model does on the test set.

This is a standard measure for classification problems since it is very easy to calculate. Moreover, it represents well the quality of the prediction model when the classes are well defined. However, in our scenario the possible prediction classes are represented by the set of configurations we discover with the tuner. This is the only information we have on these configurations. For

Decision Tree Name	**Accuracy (%)**	**DTPR**	**DTTR**	**Total number of Leaves**	**Decision Tree Height**	**Min Samples PerSplit**	**Number of Unique Config. Gemm**	**Number of Unique Config. GemmDirect**	**Number of Leaves Gemm**	**Number of Leaves GemmDir**
h1-L1	55	0.692	1.085	2	1	1	0	2	0	2
h1-L2	55	0.560	0.828	2	1	2	0	2	0	2
h1-L4	55	0.600	0.895	2	1	4	0	2	0	2
h1-L0.1	**55**	**0.702**	**1.092**	**2**	**1**	**0.1**	**0**	**2**	**0**	**2**
h1-L0.2	55	0.631	0.955	2	1	0.2	0	2	0	2
h1-L0.3	42	0.619	0.918	2	1	0.3	0	1	0	2
h1-L0.4	42	0.559	0.822	2	1	0.4	0	1	0	2
h1-L0.5	42	0.418	0.691	2	1	0.5	0	2	0	2
h2-L1	52.5	0.638	1.012	4	2	1	0	2	0	4
h2-L2	52.5	0.544	0.823	4	2	2	0	2	0	4
h2-L4	52.5	0.500	0.749	4	2	4	0	2	0	4
h2-L0.1	55	0.572	0.863	4	2	0.1	0	2	0	4
h2-L0.2	55	0.540	0.820	3	2	0.2	0	2	0	3
h2-L0.3	42	0.555	0.831	2	1	0.3	0	1	0	2
h2-L0.4	42	0.560	0.838	2	1	0.4	0	1	0	2
h2-L0.5	42	0.499	0.715	2	1	0.5	0	2	0	2
h4-L1	56.5	0.641	1.005	16	4	1	1	2	1	15
h4-L2	58	0.517	0.781	16	4	2	1	2	0	15
h4-L1	56.5	0.677	1.062	15	4	4	1	2	0	14
h4-L0.1	55	0.577	0.878	7	4	0.1	0	4	0	7
h4-L0.2	55	0.446	0.681	4	3	0.2	0	3	0	4
h4-L0.3	42	0.502	0.742	2	1	0.3	0	1	0	2
h4-L0.4	42	0.529	0.778	2	1	0.4	0	1	0	2
h4-L0.5	42	0.440	0.617	2	1	0.5	0	2	0	2
h8-L1	55	0.584	0.863	84	8	1	5	13	6	78
h8-L2	56.5	0.466	0.669	60	8	2	2	9	3	57
h8-L4	52.5	0.551	0.826	45	8	4	1	7	2	43
h8-L0.1	55	0.473	0.682	8	5	0.1	0	5	0	8
h8-L0.2	55	0.466	0.669	4	3	0.2	0	3	0	4
h8-L0.3	42	0.571	0.850	2	1	0.3	0	1	0	2
h8-L0.4	42	0.592	0.885	2	1	0.4	0	1	0	2
h8-L0.5	42	0.591	0.865	2	1	0.5	0	2	0	2
hMax-L1	52.5	0.846	1.008	166	17	1	9	16	15	151
hMax-L2	54	0.570	0.858	95	15	2	3	12	4	91
hMax-L4	52.5	0.554	0.815	53	10	4	1	7	2	51
hMax-L0.1	55	0.487	0.708	8	5	0.1	0	5	0	8
hMax-L0.2	55	0.438	0.667	4	3	0.2	0	3	0	4
hMax-L0.3	42	0.628	0.954	2	1	0.3	0	1	0	2
hMax-L0.4	42	0.604	0.895	2	1	0.4	0	1	0	2
hMax-L0.5	42	0.496	0.714	2	1	0.5	0	2	0	2

TABLE 7.3: Decision Trees on AntonNet dataset - Mali-T860. The row corresponding to the *best* Decision Tree for this dataset and architecture is highlighted.

Dataset Name	**Dataset Size**	**Number of Unique Config. Xgemm**	**Number of Unique Config. XgemmDirect**	**Best Decision Tree Name**	**Best DecisionTree accuracy**	**Best Decision Tree DTPR**	**Best Decision Tree DTTR**
AntonNet	456	1	81	h4-L1	36	0.484	1.013
PowerOf2(po2)	216	2	41	hMax-L1	21	0.431	0.931
GridOf2(go2)	3375	6	22	hMax-L1	60	0.852	1.424

TABLE 7.4: Decision Trees statistics summary for each dataset - P100.

Dataset Name	**Dataset Size**	**Number of Unique Config. Xgemm**	**Number of Unique Config. XgemmDirect**	**Best Decision Tree Name**	**Best DecisionTree accuracy**	**Best Decision Tree DTPR**	**Best Decision Tree DTTR**
AntonNet	456	28	35	h1-L0.1	55	0.702	1.092
PowerOf2(po2)	216	29	1	h8-L0.1	45	0.551	1.121

TABLE 7.5: Decision Trees statistics summary for each dataset - Mali-T860.

instance, we do not know if, for a matrix m_i, a configuration $C'_{m_i} \neq C^{best}_{m_i}$ exists, whose performance are closer to the best. This means that there is not a sharp division among classes. For this reason, we defined the following two metrics in order to experimentally evaluate the quality of decision trees generated by the framework. The first metrics is the average ratio between the performance of a model and the peak performance of the tuner. The second one is the average ratio between the performance of a model over the performance of the auto-tuned version of CLBlast. We respectively named them **DTPR** (**DTP**eak**R**atio) and **DTTR** (**DTT**une**R**atio). These two metrics provide a better estimation of our models as they are able to take into account the cases where the model infers a wrong class in terms of classification, but that are actually quite good in terms of performance. In Tables 7.2 and 7.3, we report all the statistics and metrics about the decision trees that our framework generates starting from the training set of *go2* and *AntonNet*, respectively for the P100 and the Mali. The best model for P100, according to the latter two metrics, is the hMax-L1 even if it has not the highest value for the accuracy whereas for the Mali is the h1-L0.1.

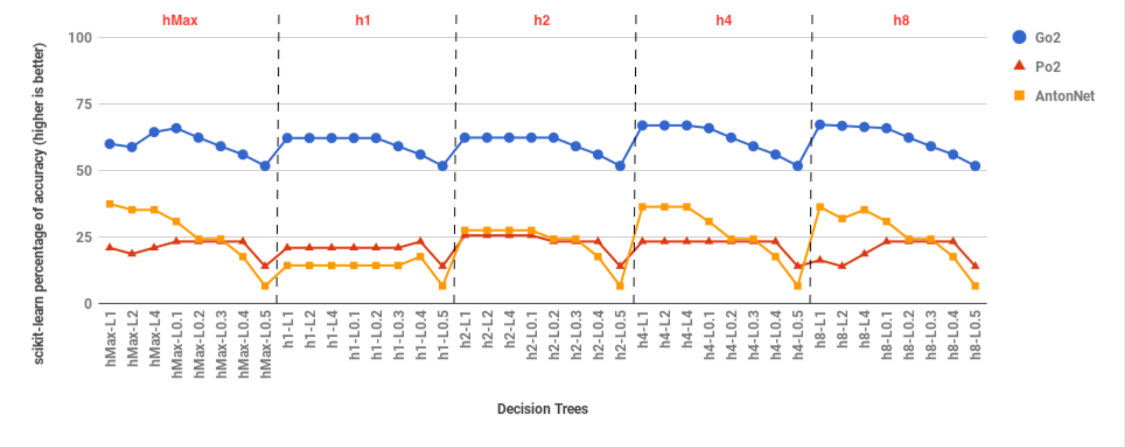


FIGURE 7.2: Percentage of accuracy of Decision Trees - P100.

In Figures 7.2 and 7.3, we report the accuracy of all the decision trees generated by our framework, respectively for the P100 and the Mali devices. The value of accuracy strictly depends on the distribution and the number of classes belong to the dataset (see Table 7.4 and Table 7.5),

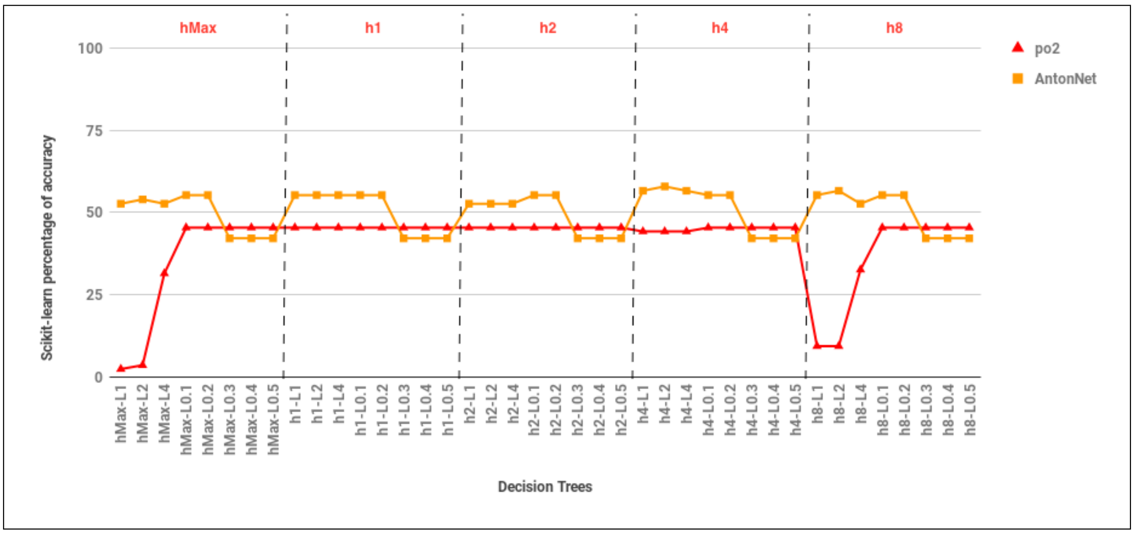


FIGURE 7.3: Percentage of accuracy of Decision Trees - Mali-T860.

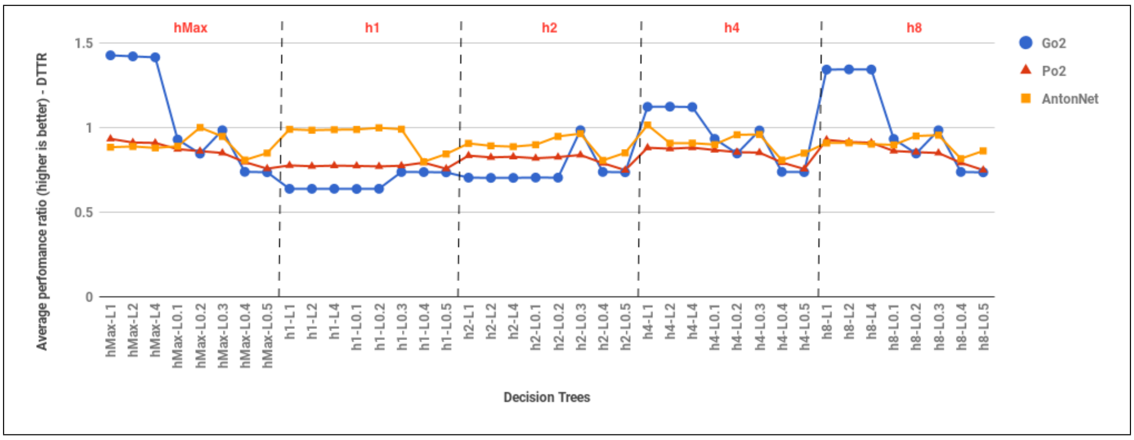


FIGURE 7.4: Average performance ratio between Decision Trees and the auto-tuned version of CLBlast (DTTR) - P100.

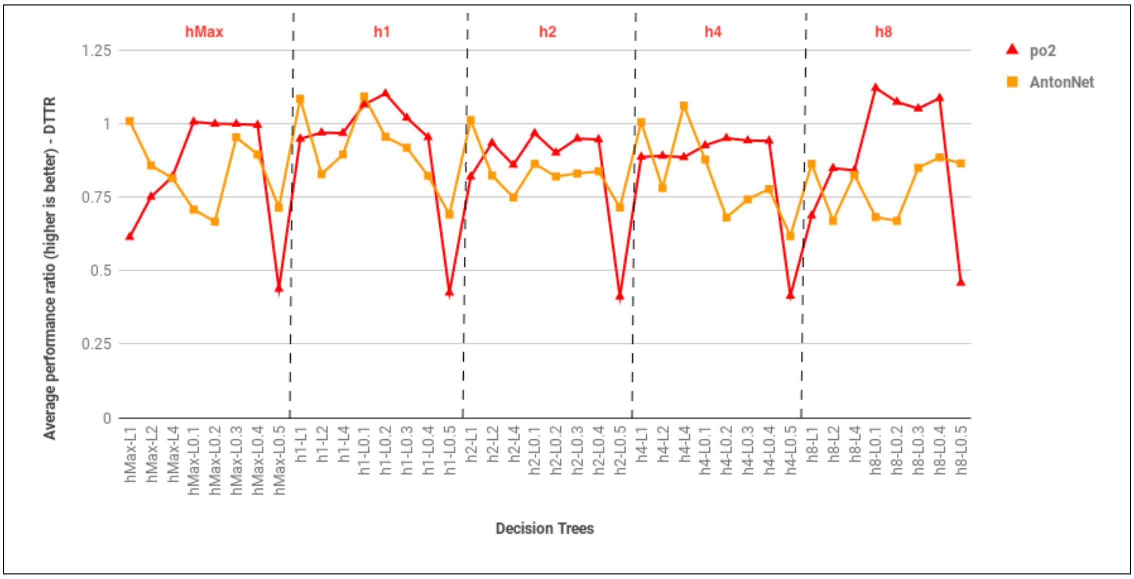


FIGURE 7.5: Average performance ratio between Decision Trees and the auto-tuned version of CLBlast (DTTR) - Mali-T860.

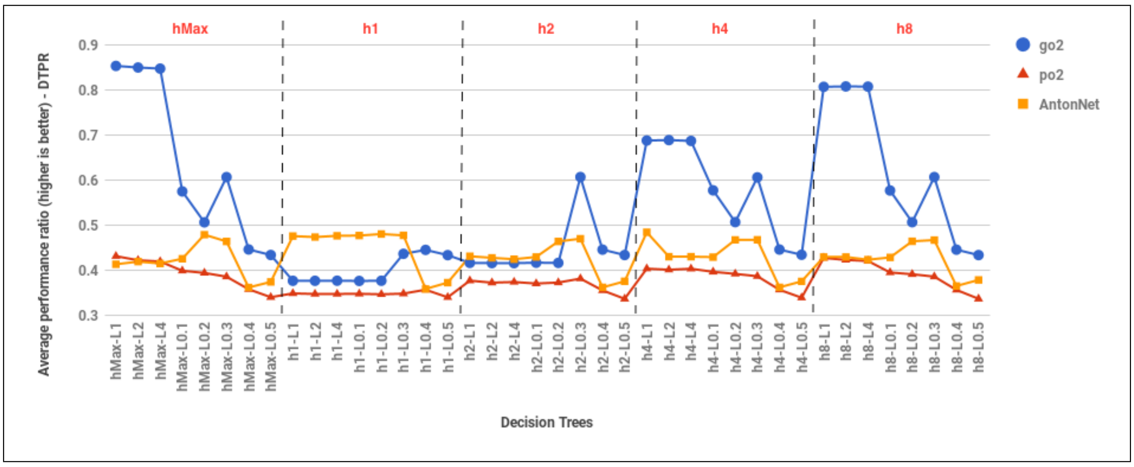


FIGURE 7.6: Average performance ratio between Decision Trees and the peak of the tuner (DTPR) - P100.

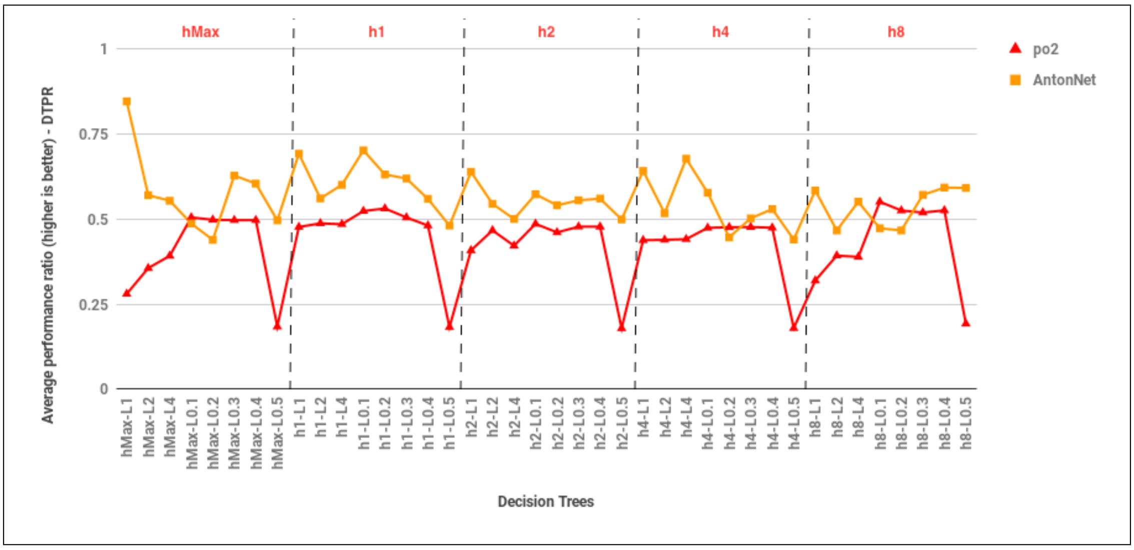


FIGURE 7.7: Average performance ratio between Decision Trees and the peak of the tuner (DTPR) - Mali-T860.

and it seems to be not really affected by the trees parameters. For example on P100, the accuracy value of *go2* is stable compared to *AntonNet* and *po2*; the same behaviour can be detected on Mali as well.

Figures 7.4 and 7.5, report the DTTR values of all the decision trees generated by our framework, respectively for the P100 and the Mali devices, while the DTPR values are reported in Figures 7.6 and 7.7. The values of these two metrics are indeed close related to the tree parameters. In particular, they are strictly affected by the value of the minimum number of samples splitting an internal node. We expected that behaviour, as that parameter implicitly assigns a weight to the classes; the values of these weights are proportional to the number of occurrences of the class in the training set. The DTTR values are also interesting as they provide a measure of how good it is a decision tree with respect to the standard CLBlast in terms of performance. Indeed, the

DTPR values give an estimation of how close a decision tree is to the theoretically best solution where we select for each matrix its best configuration.

As we are interested in evaluating the overhead of our generated runtime, we measured the time to pick up a configuration for all the matrices we tested. In the worst case, when we evaluated the hMax-L1 on *go2* that has more than 1200 leaves (see Table 7.2), our runtime introduced an overhead in terms of performance that is less than 2% on small matrices that decreases as the size of the matrices grow. On average, the overhead impacts less than 1% on execution time so it can be considered negligible.

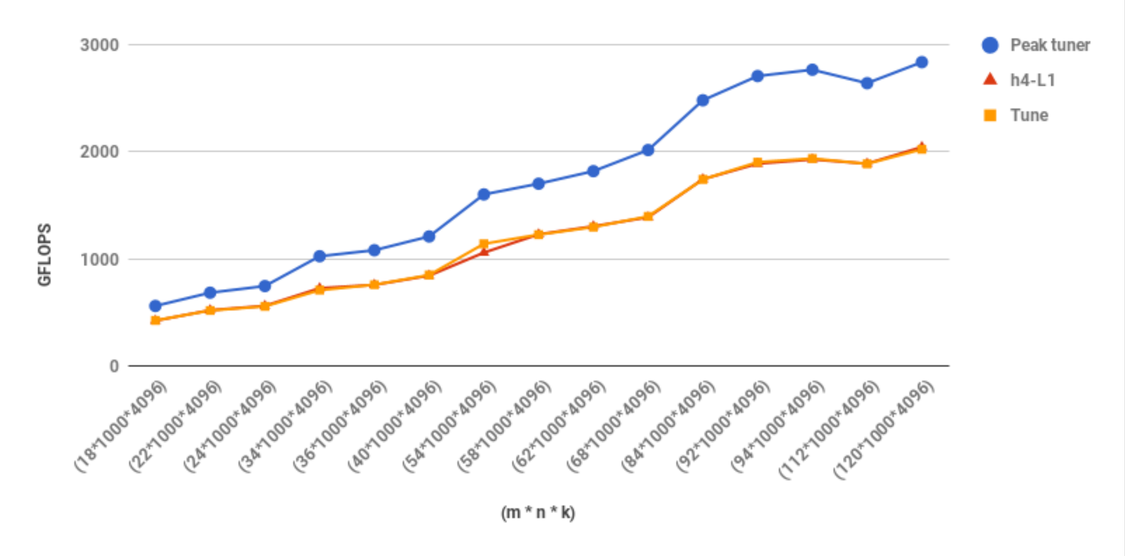


FIGURE 7.8: Comparison of three version of CLBlast(Peak of the tuner, h4-L1 and Tune) on AntonNet-P100 - N = 1000, K = 4096.

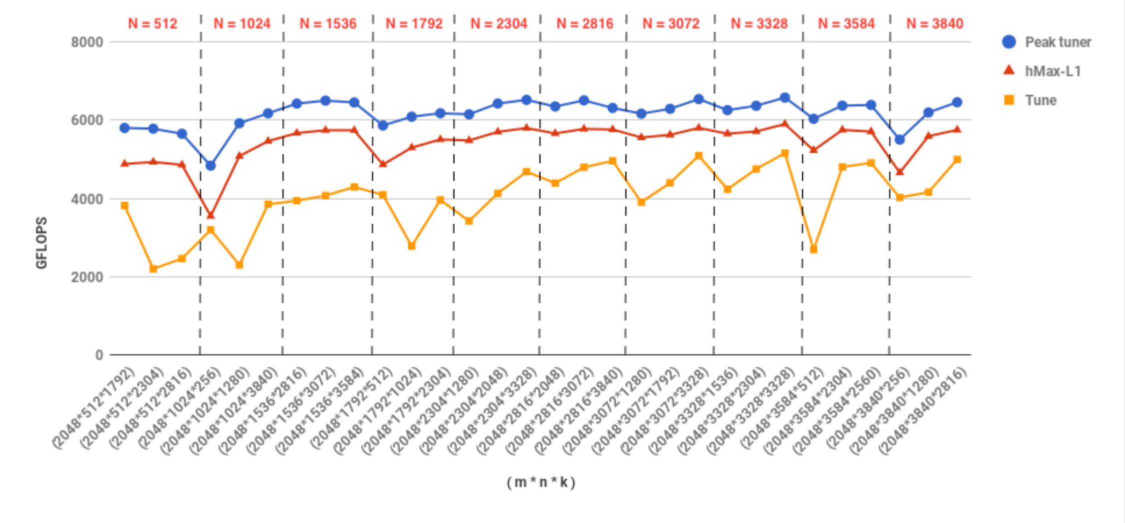


FIGURE 7.9: Comparison of three version of CLBlast(Peak of the tuner, hMax-L1 and Tune) on go2-P100 - M = 2048.

Finally, we show the performance of the best decision tree generated by our framework for each dataset. We report the comparison between the performance of CLBlast using the runtime, the

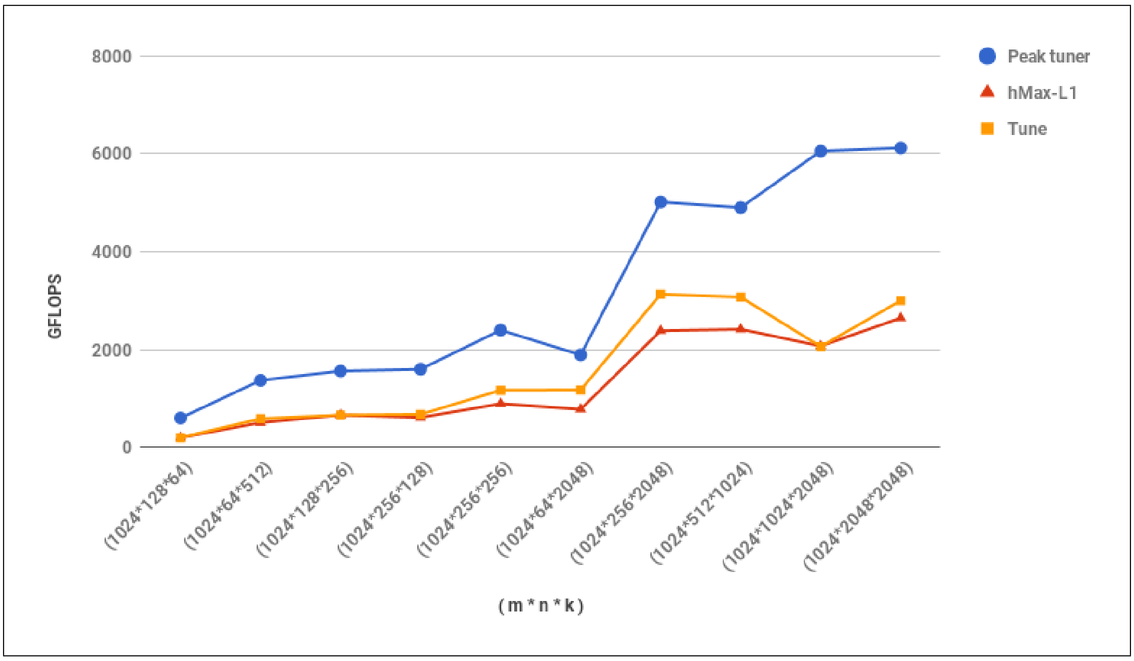


FIGURE 7.10: Comparison of three version of CLBlast(Peak of the tuner, hMax-L1 and Tune) on po2-P100 - M = 1024.

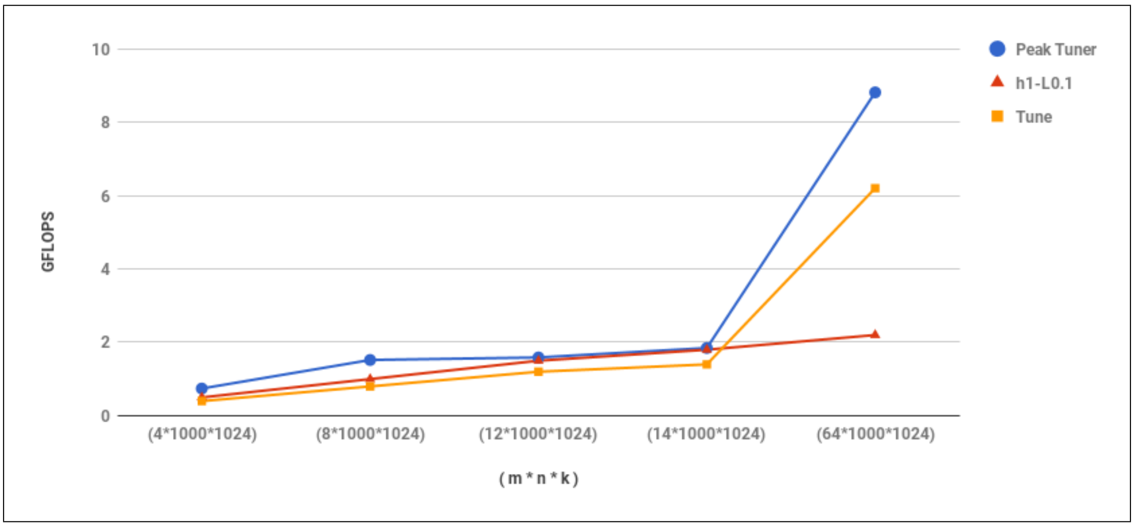


FIGURE 7.11: Comparison of three version of CLBlast(Peak of the tuner, h1-L0.1 and Tune) on AntonNet-Mali-T860 - N = 1000 and K = 1024.

auto-tuned CLBlast and the peak performance of the tuner. For the experiments, we used the matrices belonging to the test sets in order to fairly evaluate the models.

The results on *AntonNet*, *go2* and *po2* on P100 are reported respectively in Figures 7.8, 7.9 and 7.10, while the results on AntonNet and *po2* on Mali-T860 are presented in Figures 7.11 and 7.12. On P100, the performance of the best decision tree on *go2* is very close to the peak performance of the tuner and it outperforms up to 3× the performance of the auto-tuned CLBlast. On the other two datasets, our best decision tree performance is closer to the standard CLBlast. These results are in agreement with the values of the DTPR and DTTR metrics. Furthermore, both *AntonNet* and *po2* on P100 have an unbiased distribution of the configurations.

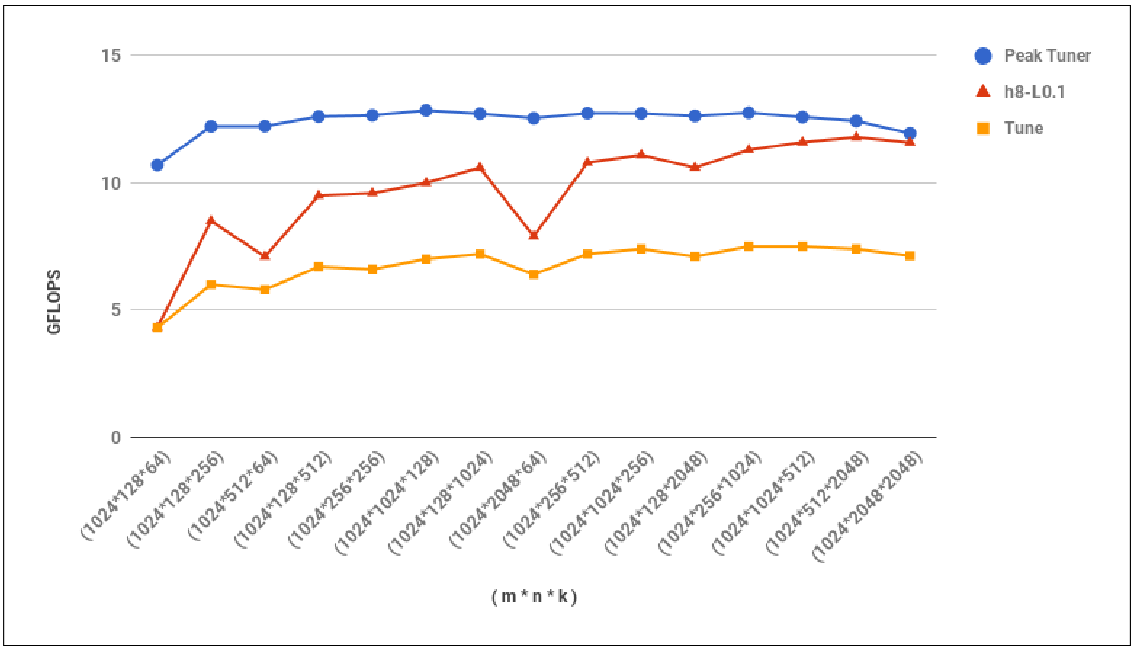


FIGURE 7.12: Comparison of three version of CLBlast(Peak of the tuner, h8-L0.1 and Tune) on po2-Mali-T860 - M = 1024.

By looking at Table 7.4, we can see that for the two datasets almost all configurations correspond to xgemm_direct; an unbiased distribution of the configurations may affect negatively the prediction as the experiments show.

On Mali-T860 instead, the configurations of *AntonNet* and *po2* belong to both the kernels xgemm and xgemm_direct (see Table 7.5). The experiments show that, in this case, the performance of our library is close to the peak of tuner for most of the matrices. In particular, our adaptive libraries are able to outperform up to 2.5× the performance compared to the by-hand optimised and tuned CLBlast Library on Mali-T860.

8 Concluding Remarks

In the present dissertation, the candidate presented the main contributions of his PhD related to two different topics: in the first part, a cryptanalysis framework of the Cube attack for GPUs is presented along with a detailed discussion about its design and implementation as well as the state-of-the-art cryptanalytic results obtained by using it against a round-reduced version of the Trivium cipher. In the second part, the candidate proposed a new runtime system to speedup data-driven applications that leverages on Machine Learning techniques. He also presented a proof of concept to speedup matrix multiplication for unpredictable matrix sizes along with a detailed performance analysis.

The main contributions of each part are hereunder summarized.

8.1 Part I

In the first part of the thesis, the candidate presented and discussed the first all-round GPU-tailored implementation of the Cube attack, resulting in a flexible and powerful framework, validated against known results in the literature, and soon to be released into the public domain. The tool can be used against any cipher (with minimal effort), and it supports both latest generation GPU architectures and workload distribution over multi-GPU systems. The framework allows to improve the state-of-art attacks against reduced-round versions of Trivium.

Moreover, he provided a careful performance analysis that shows the feasibility and the scalability of the proposed approach. This opens new prospects related to the possibility of expanding the quest for superpolys to a dimension never explored in previous works.

More generally, the proposed framework is expected to be the starting point of future attacks, thus paving the way for further research able to assess the real potential of the still disputed yet praised Cube attack.

In particular, the candidate:

1. shows how to tune the design and implementation of the Cube attack to the characteristics of GPUs, in order to fully exploit parallelization while coping with limited memory [16, 19].
2. presents a flexible framework to mount Cube attacks against any cipher, under the sole condition that the cipher is as well implemented in GPU. The tool is independent of the GPU architecture, and it supports extension to multi-GPU systems.

Moreover,

3. The framework proposed by the candidate improves the state-of-the-art against reduced-round versions of Trivium, yielding full key recovery up to 781 initialization rounds without bruteforce and the first ever maxterm after 800 initialization rounds [16]. In addition, some new candidate maxterms for both Trivium after 800 initialization rounds and for Grain-128 after 160 initialization rounds have been discovered by running some new experiments. The detailed description and analysis of these results are going to be released soon in a work that is now being finalized.
4. The candidate proposes an implementation that allows for exhaustively assigning values to (subsets of) public variables with negligible additional costs. This means extending the quest for superpolys to a dimension never explored in previous works, and, by not being tied to a very small set of IV combinations, potentially weakening one of the basic requirements of the Cube attack, that is, the assumption of a completely tweakable IV.
5. He carefully analyses the performance of the proposed implementation, in terms of: (i) speedup with respect to a parallel CPU implementation, (ii) dependence on system parameters, (iii) comparison among different architectures (including latest generation GPU cards), and (iv) impact of a multi-GPU distributed approach [19].
6. He provides the first GPU tailored implementation, to the best of his knowledge, for Trivium and Grain-128. He validates the framework on both the ciphers by extracting the symbolical representation of the polynomial corresponding to round-reduced versions of the ciphers. He then runs the attacks on some selected cubes, specifically selected from the symbolical representations to verify the consistency of the results.

8.2 Part II

The second part of the dissertation discussed the prospective of a new generation of adaptive high-optimized libraries based on Machine Learning techniques. The candidate started his analysis by evaluating a simple supervised classifier to build a predictive model for GEMM on GPUs. He analysed the performance of several models by taking into account different parameters. From a theoretical prospective, as expected, Decision Trees showed very low accuracy. However, from a more practical point of view the impact in terms of performance of the sub-optimal classification is mitigated. The candidate also showed how to generate datasets from tuners, as well as, how to integrate predictive models in existing libraries like CLBlast. Finally, he presented a proof of concept for generating adaptive library for matrix multiplication and he evaluated the generated libraries on two different GPU architectures (Nvidia Pascal and ARM Mali Midgard). In both cases, the adaptive libraries allow to obtain a speed-up by up to 3x and 2.5x over a traditionally optimized and tuned library. There are several possible directions of future work. First, it is worth investigating more complex techniques in order to improve the accuracy of the models. Second, it will be interesting to investigate how to generate representative and compact training sets. This aspect is particular crucial for embedded architectures where the generation of the training set is expensive (i.e., the generation of *po2* required 7 days). Both the source code and the datasets used in the experiments are going to be release soon.

The candidate in the second part:

1. describes and provides three different training datasets generated from a tunable BLAS library for GEMM routine on GPUs;
2. analyzes several configurations of a simple univariate supervised classifier, *Decision Trees*, used to predict an optimized GEMM implementation and related tuning parameters;
3. describes a suitable framework and workflow able to build the decision rule inside BLAS library;
4. validates the proposed study by providing exhaustive experimental results where he evaluates several metrics ranging from the accuracy of each model to the overhead of the decision rules passing through a detailed analysis of the performance;
5. shows as the integration of the proposed solution in a OpenCL BLAS library, CLBlast [22], offers up to 3x and 2.5x of speed-up both on high-end NVIDIA GPU architecture and a low power embedded ARM Mali GPU.

Appendix A

M	N	K
$2l, l \in \{1, ..., 64\}$	1000	1
$2l, l \in \{1, ..., 64\}$	1000	4096
$2l, l \in \{1, ..., 64\}$	4096	1
$2l, l \in \{1, ..., 64\} \setminus \{63\}$	4096	4096
$2l, l \in \{1, ..., 64\} \setminus \{63\}$	4096	9216
$\{128\}$	$\{169\}$	$\{1728\}$
$\{128\}$	$\{729\}$	$\{1200\}$
$\{192\}$	$\{169\}$	$\{1728\}$
$\{256\}$	$\{169, 729\}$	$\{1\}$
$\{384\}$	$\{169\}$	$\{1, 2304\}$

Table A.1: Matrix sizes collected from AlexNet - batch size 2 to 128.

M	N	K
$2l, l \in \{1, ... , 64\}$	1000	{1, 1024}
{16, 96, 192}	{196}	{1, 480}
{16}	{784}	{1, 192}
{24, 112, 144}	{196}	1 {1, 512}
{32, 48, 160, 192, 256}	{49}	{1, 832}
{32, 160}	{196}	{1, 512, 528}
{32}	{784}	{1, 192, 256, 400}
{48}	{196}	{1, 400}
{64}	{196}	{1, 480, 512, 600, 800}
{64}	{784}	{1, 192, 256}
{64}	{3136}	{1, 64}
{64}	{12544}	{1, 147}
{96}	{784}	{1, 192, 800}
{128}	{196}	{1, 512, 528, 800}
{128}	{784}	{1, 256, 864}
{192}	{784}	{1, 1152}
{192}	{3136}	{1, 576}
{208}	{196}	{1, 864}
{224}	{196}	{1, 1008}
{256}	{196}	{1, 528, 1152}
{288}	{196}	{1, 1296}
{320}	{49, 196}	{1, 1440}
{384}	{49}	{1, 832, 1728}

Table A.2: Matrix sizes collected from GoogleNet - batch size 2 to 128.

M	N	K
{16}	{3136}	{1, 64, 128}
{32}	{784}	{1, 128, 256}
{48}	{196}	{1, 256, 384}
{64}	{196}	{1, 384, 512}
{64}	{3136}	{1, 16, 144}
{64}	{12769}	{1, 27}
{128}	{784}	{1, 32, 288}
{192}	{196}	{1, 48, 432}
{256}	{196}	{64, 576}
{1000}	{196}	{1, 512}

Table A.3: Matrix sizes collected from SqueezeNet 1.1 - batch size 2 to 128.

Appendix B

Since their advent as general purpose devices, GPUs have been exploited in many scientific fields like Bioinformatics, Computational Finance, Physics, Data Science, Cryptography, Machine Learning and so on. Their high level of parallelism and computational power combined with high level API and libraries attracted researchers and companies and nowadays GPUs are considered to be crucial for many tasks, as demonstrated by the last Top 500 Supercomputer list, where 2 supercomputers equipped with GPUs belong to the top ten[1].

In the following, a brief description of the GPU architecture that summarises their characteristics is provided; it will use the Nvidia CUDA terminology as it is the same used in the first part of this dissertation.

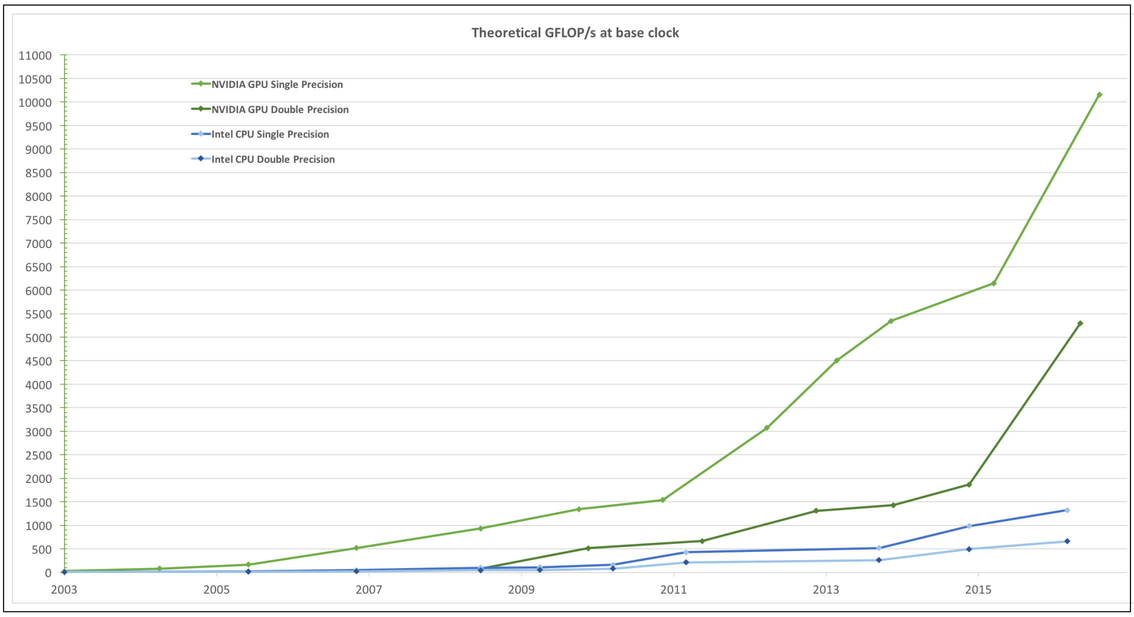


FIGURE B.1: Floating-Point Operations per Second for the CPU and GPU [25].

[1] https://www.top500.org/list/2017/11/

As illustrated in Figure B.1, the extraordinary computational horsepower of GPUs may offer up to 7× of FLOPS compared to CPU. This due the fact that GPUs are designed for compute-intensive and highly parallel computation therefore they have more transistors devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure B.2.

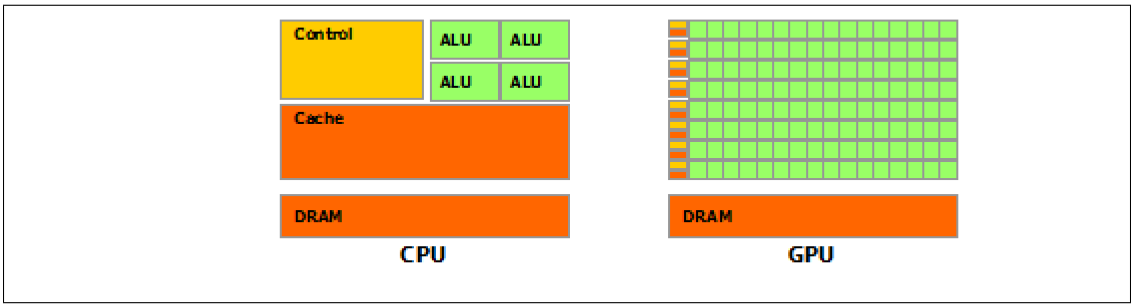


FIGURE B.2: The GPU Devotes More Transistors to Data Processing [25].

In particular, GPUs are ideal candidates to address problems where the same flow of operations, with high arithmetic intensity, is executed on many data elements in parallel: the main pros of executing the same flow of operations on each data element is a lower requirement for sophisticated flow control; the high arithmetic intensity combined with the concurrent execution on many data elements allow to hide the memory access latency with calculations, without the need of big data caches.

In 2006, Nvidia proposed a general purpose parallel computing platform and programming model, CUDA, that allow to exploit the parallel compute engine of its GPUs to solve many computational problems efficiently than the CPU. Nvidia also provides a software environment that allows developers to use C as high-level programming language and a rich set of libraries. From hardware perspective, the basic components of an Nvidia GPU are the general-purpose processors called *streaming multiprocessors* (SMs). Each SM contains many CUDA cores, so the total number of CUDA cores available is equal to the number of cores per multiprocessor multiplies to the number of SM. The multiprocessor is designed to execute hundreds of threads concurrently; to manage such a large amount of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread) [25]. The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. Although fully supported, divergent branch of threads should be avoided, if possible, as it forces divergent threads to be executed sequentially and negatively impacts the performance. Furthermore, each SM contains thousands of 32-bit registers that can be partitioned among threads of executions, warp schedulers, and a parallel data cache or shared memory.

From software perspective, Nvidia provides the CUDA programming, that has been designed to help programmers in developing parallel applications, composed by device functions called *kernel*, able to transparently exploit newest device capabilities and scale as more processor cores

become available. It exposes to developers a minimal set of language extensions that provides three main abstractions: a hierarchy of thread groups, shared memories and barrier synchronization. These abstractions support different kind of parallelism from fine-grained (data and thread) to coarse-grained (data and task) and they help the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads; within the blocks each sub-problem can be partitioned into finer pieces and all the threads belong to the block can cooperatively in parallel solve it. The blocks of threads are then executed independently from each other on SMs. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.
Thread blocks are required to execute independently: it must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure B.3, enabling programmers to write code that scales with the number of cores. Each block can contain up to a predefined number of threads (1024 on current GPUs), since all these threads are expected to reside on the same processor core and must share the limited memory resources of that core.

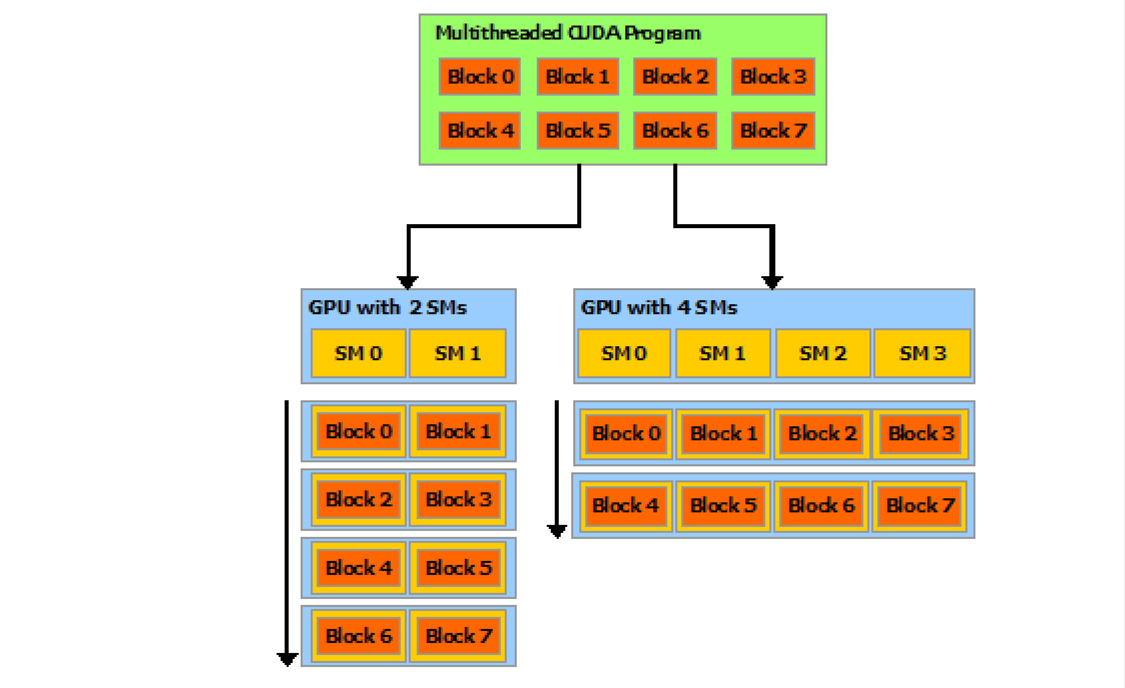


FIGURE B.3: Automatic Scalability [25].

Threads can access different types of GPU memory: *registers*, *local memory*, *shared memory*, and *global memory*. The fastest but smallest memory is provided by the registers that are located on-chip; they are equally divided across active threads in each multiprocessor. As each multiprocessor has a fixed number of registers, an excessive use of them (i.e. high register pressure) limits the number of threads that can run simultaneously and it may expose memory latency.
The global memory is the biggest and slowest available and it includes also the local memory;

the global can be accessed by all the threads and it is the default space for the GPU input data and the kernels output data. The local memory is private to each thread and it is used to store data that does not fit in the registers, reducing the register pressure. Both registers and local memory are managed by the compiler and the programmers do not have directly access to them. The shared memory is visible visible to all threads of the block and with the same lifetime as the block; it is quite fast as it resides on-chip. There are also two additional read-only memory spaces accessible by all threads: the *constant* and *texture* memory spaces. Each memory types is optimized for different memory usages and the memory access pattern may heavily influence the performance of the application; interested readers may find an in depth discussion about it in [93]. For example, for global memory, as a general rule, the more scattered the addresses are, the more reduced the throughput is. The reason is that global memory is accessed via 32-, 64, or 128-byte byte memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. It is therefore fundamental to maximize coalescing memory accesses in order to achieve high memory bandwidth. The throughput of global memory influences the overall application performance as the global memory is the default space for GPU's input and output; for this reason, the memory access pattern plays a fundamental role in the design and implementation of the applications.

In the present dissertation the problem of optimising GPU applications has been addressed from two different point-of-view. In the first part, a cryptanalysis framework specifically tailored on GPUs has been proposed, all the design and implementation choices are carefully described and the pros and cons highlighted. The framework's kernels and the ciphers were painstakingly designed and implemented by avoiding high registers pressure and divergent branches; moreover, the enforced memory access pattern provides full coalescing thus maximizing the throughput. In the second part of the present thesis, a new strategy for automatically tune GPU applications leverage on Machine Learning has been proposed. The tuning process plays with low-level implementation choices as, for example, memory padding and striding access to identify the memory access pattern that maximize the throughput and, consecutively, the overall performance. As already mentioned, developing GPU applications that are able to efficiently exploit the potentiality of GPUs is an hard process; the programmers have to design parallel algorithms and they have to take care about low-level implementation's details, like for example the memory access pattern, registers pressure, and divergent branches, to develop high performance applications that really exploit all the GPUs potentiality.

Bibliography

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 03 2009.

[2] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. ISBN 1584885513.

[3] ETSI/SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G specification, version 1.1. http://www.3gpp.org/ftp/, September 2006.

[4] ETSI/SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms128-EEA3 & 128EIA3; Document 2: ZUC specification. http://www.3gpp.org/ftp/, June 2011.

[5] Antoine Joux. *Algorithmic Cryptanalysis*. Chapman & Hall/CRC, 1st edition, 2009. ISBN 1420070029, 9781420070026.

[6] T. Siegenthaler. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Trans. Comput.*, 34(1):81–85, January 1985. ISSN 0018-9340. doi: 10.1109/TC.1985.1676518. URL http://dx.doi.org/10.1109/TC.1985.1676518.

[7] Willi Meier and Othmar Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1(3):159–176, Oct 1989. ISSN 1432-1378. doi: 10.1007/BF02252874. URL https://doi.org/10.1007/BF02252874.

[8] Nicolas Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, pages 345–359, 2003. doi: 10.1007/3-540-39200-9_21. URL https://doi.org/10.1007/3-540-39200-9_21.

[9] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In *Advances in Cryptology-EUROCRYPT 2009*, pages 278–299. Springer, 2009.

[10] Pierre-Alain Fouque and Thomas Vannet. *Improving Key Recovery to 784 and 799 Rounds of Trivium Using Optimized Cube Attacks*, pages 502–517. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-43933-3. doi: 10.1007/978-3-662-43933-3_26. URL http:\/\/dx.doi.org\/10.1007\/978-3-662-43933-3_26.

[11] Jean-Philippe Aumasson, Itai Dinur, Luca Henzen, Willi Meier, and Adi Shamir. Efficient FPGA implementations of high-dimensional cube testers on the stream cipher grain-128. *IACR Cryptology ePrint Archive*, 2009:218, 2009. URL http://eprint.iacr.org/2009/218.

[12] Itai Dinur, Tim Güneysu, Christof Paar, Adi Shamir, and Ralf Zimmermann. An experimentally verified attack on full Grain-128 using dedicated reconfigurable hardware. In *Proceedings of the 17th International Conference on The Theory and Application of Cryptology and Information Security*, ASIACRYPT'11, pages 327–343, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25384-3. doi: 10.1007/978-3-642-25385-0_18. URL http://dx.doi.org/10.1007/978-3-642-25385-0_18.

[13] Robert Szerwinski and Tim Güneysu. *Exploiting the Power of GPUs for Asymmetric Cryptography*, pages 79–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-85053-3. doi: 10.1007/978-3-540-85053-3_6. URL https://doi.org/10.1007/978-3-540-85053-3_6.

[14] Mauro Bisson. *Combining Algortihms and Technologies to Speedup Computing Intensive Applications*. PhD thesis, Department of Computer Science - Sapienza University of Rome, 2010.

[15] Massimo Bernaschi, Mauro Bisson, Emanuele Gabrielli, and Simone Tacconi. An Architecture for Distributed Dictionary Attacks to Cryptosystems. *JCP*, 4(5):378–386, 2009. doi: 10.4304/jcp.4.5.378-386. URL https://doi.org/10.4304/jcp.4.5.378-386.

[16] Marco Cianfriglia, Stefano Guarino, Massimo Bernaschi, Flavio Lombardi, and Marco Pedicini. *A Novel GPU-Based Implementation of the Cube Attack*, pages 184–207. Springer International Publishing, Cham, 2017. ISBN 978-3-319-61204-1. doi: 10.1007/978-3-319-61204-1_10. URL https://doi.org/10.1007/978-3-319-61204-1_10.

[17] Christophe De Cannière. *Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles*, pages 171–186. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-38343-7. doi: 10.1007/11836810_13. URL http://dx.doi.org/10.1007/11836810_13.

[18] M. Hell, T. Johansson, A. Maximov, and W. Meier. A Stream Cipher Proposal: Grain-128. In *2006 IEEE International Symposium on Information Theory*, pages 1614–1618, July 2006. doi: 10.1109/ISIT.2006.261549.

[19] M. Cianfriglia and S. Guarino. Cryptanalysis on GPUs with the Cube Attack: Design, Optimization and Performances Gains. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 753–760, July 2017. doi: 10.1109/HPCS.2017.114.

[20] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1953048.2078195.

[21] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[22] Cedric Nugteren. CLBlast: A Tuned OpenCL BLAS library. *CoRR*, abs/1705.05249, 2017. URL http://arxiv.org/abs/1705.05249.

[23] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, June 2015. doi: 10.1109/CVPR.2015.7298594.

[24] Christophe De Canniere and Bart Preneel. Trivium specifications. *eSTREAM, ECRYPT Stream Cipher Project*, 2006, 2005.

[25] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2017. Version 9.1.85.

[26] M. Lulli, M. Bernaschi, and G. Parisi. Highly optimized simulations on single- and multi-gpu systems of the 3d ising spin glass model. *Computer Physics Communications*, 196: 290 – 303, 2015. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2015.06.019. URL http://www.sciencedirect.com/science/article/pii/S0010465515002647.

[27] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX*

Conference on Operating Systems Design and Implementation, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026899.

[28] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[29] R. Collobert, K. Kavukcuoglu, and C. Farabet. Implementing neural networks efficiently. In G. Montavon, G. Orr, and K-R. Muller, editors, *Neural Networks: Tricks of the Trade*. Springer, 2012.

[30] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

[31] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[32] Ralph Jacobson. 2.5 quintillion bytes of data created evey day. how does CPG & Retail manage it? https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/, 2013.

[33] G. Totaro, M. Bernaschi, G. Carbone, M. Cianfriglia, and A. Di Marco. ISODAC: A high performance solution for indexing and searching heterogeneous data. *Journal of Systems and Software*, 118(Supplement C):115 – 133, 2016. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2015.11.043. URL http://www.sciencedirect.com/science/article/pii/S0164121215002678.

[34] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59(11):56–65, October 2016. ISSN 0001-0782. doi: 10.1145/2934664. URL http://doi.acm.org/10.1145/2934664.

[35] M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Inf. Theor.*, 26(4): 401–406, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.1980.1056220. URL http://dx.doi.org/10.1109/TIT.1980.1056220.

[36] Michael Vielhaber. Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack, 2007. URL http://eprint.iacr.org/2007/413.

[37] Itai Dinur and Adi Shamir. *Breaking Grain-128 with Dynamic Cube Attacks*, pages 167–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21702-9. doi: 10.1007/978-3-642-21702-9_10. URL http://dx.doi.org/10.1007/978-3-642-21702-9_10.

[38] Zahra Ahmadian, Shahram Rasoolzadeh, Mahmoud Salmasizadeh, and Mohammad Reza Aref. Automated dynamic cube attack on block ciphers: Cryptanalysis of SIMON and KATAN. *IACR Cryptology ePrint Archive*, 2015:40, 2015.

[39] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and trivium. In *Fast Software Encryption*, pages 1–22. Springer, 2009.

[40] Anubhab Baksi, Subhamoy Maitra, and Santanu Sarkar. New distinguishers for reduced round trivium and trivia-sc using cube testers. In *WCC2015-9th International Workshop on Coding and Cryptography 2015*, 2015.

[41] Richard Winter, Ana Salagean, and Raphael C.-W. Phan. Comparison of cube attacks over different vector spaces. In *Proceedings of the 15th IMA International Conference on Cryptography and Coding - Volume 9496*, IMACC 2015, pages 225–238, New York, NY, USA, 2015. Springer-Verlag New York, Inc. ISBN 978-3-319-27238-2. doi: 10.1007/978-3-319-27239-9_14. URL http://dx.doi.org/10.1007/978-3-319-27239-9_14.

[42] Andrea Agnesse and Marco Pedicini. Cube attack in finite fields of higher order. In *Proceedings of the Ninth Australasian Information Security Conference - Volume 116*, AISC '11, pages 9–14, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc. ISBN 978-1-920682-96-5. URL http://crpit.com/confpapers/CRPITV116Agnesse.pdf.

[43] Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-11709-9.

[44] D. Wagner M. Briceno, I. Goldberg. A pedagogical implementation of A5/1. http://www.scard.org/gsm/a51.html, 1999.

[45] Matthew Robshaw and Olivier Billet, editors. *New Stream Cipher Designs: The eSTREAM Finalists*. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-68350-6.

[46] Martin Hell, Thomas Johansson, and Willi Meier. Grain&#58; a Stream Cipher for Constrained Environments. *Int. J. Wire. Mob. Comput.*, 2(1):86–93, May 2007. ISSN 1741-1084. doi: 10.1504/IJWMC.2007.013798. URL http://dx.doi.org/10.1504/IJWMC.2007.013798.

[47] Xinxin Fan and Guang Gong. *On the Security of Hummingbird-2 against Side Channel Cube Attacks*, pages 18–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-34159-5. doi: 10.1007/978-3-642-34159-5_2. URL http://dx.doi.org/10.1007/978-3-642-34159-5_2.

[48] Shiyong Zhang, Gongliang Chen, and LiJianhua. Cube attack on reduced-round Quavium. ICMII-15 Advances in Computer Science Research, 2015.

[49] Daniel-J Bernstein. Why haven't cube attacks broken anything? https://cr.yp.to/cubeattacks.html last accessed 2016-11-11, 2009.

[50] Sean O'Neil. Algebraic structure defectoscopy, 2007. URL http://eprint.iacr.org/2007/378. Tools for Cryptanalysis 2007 Workshop sean@cryptolib.com 13859 received 23 Sep 2007, last revised 12 Dec 2007.

[51] Frank-M. Quedenfeld and Christopher Wolf. Algebraic properties of the cube attack. *IACR Cryptology ePrint Archive*, 2013:800, 2013.

[52] Chungath Srinivasan, Utkarsh Umesan Pillai, K.V. Lakshmy, and M. Sethumadhavan. Cube attack on stream ciphers using a modified linearity test. *Journal of Discrete Mathematical Sciences and Cryptography*, 18(3):301–311, 2015. doi: 10.1080/09720529.2014.995967. URL http://dx.doi.org/10.1080/09720529.2014.995967.

[53] Alexander Maximov. Cryptanalysis of the "grain" family of stream ciphers. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 283–288, New York, NY, USA, 2006. ACM. ISBN 1-59593-272-0. doi: 10.1145/1128817.1128859. URL http://doi.acm.org/10.1145/1128817.1128859.

[54] Thorsten Kleinjung, ArjenK. Lenstra, Dan Page, and Nigel-P. Smart. Using the Cloud to determine key strengths. In Steven Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012*, volume 7668 of *Lecture Notes in Computer Science*, pages 17–39. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34930-0. doi: 10.1007/978-3-642-34931-7_3. URL http://dx.doi.org/10.1007/978-3-642-34931-7_3.

[55] Michał Marks, Jarosław Jantura, Ewa Niewiadomska-Szynkiewicz, Przemysław Strzelczyk, and Krzysztof Góźdź. Heterogeneous GPU&CPU cluster for high performance computing in cryptography. *Computer Science*, 13(2):63–79, 2012.

[56] Fabrizio Milo, Massimo Bernaschi, and Mauro Bisson. A fast, GPU based, dictionary attack to OpenPGP secret keyrings. *J. Syst. Softw.*, 84(12):2088–2096, dec 2011. ISSN 0164-1212. doi: 10.1016/j.jss.2011.05.027. URL http://dx.doi.org/10.1016/j.jss.2011.05.027.

[57] Elena Agostini. Bitlocker dictionary attack using GPUs. Univ. of Cambridge Passwords Conference, 2015. https://www.cl.cam.ac.uk/events/passwords2015/preproceedings.pdf.

[58] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 73–83. ACM, 1990.

[59] Alex Samorodnitsky and Luca Trevisan. A PCP characterization of NP with optimal amortized query complexity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 191–199. ACM, 2000.

[60] Itai Dinur and Adi Shamir. Applying cube attacks to stream ciphers in realistic scenarios. *Cryptography and Communications*, 4(3-4):217–232, 2012.

[61] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS Project. *PARALLEL COMPUTING*, 27:2001, 2000.

[62] CUDA Nvidia. CUBLAS library. *NVIDIA Corporation, Santa Clara, California*, 15(27): 31, 2008.

[63] Intel Corporation, editor. *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2018. ISBN 630813-054US. URL https://software.intel.com/sites/default/files/managed/83/0a/mkl-2018-developer-reference-c_0.pdf.

[64] Sharan Narang. DeepBench. https://github.com/baidu-research/DeepBench, 2016. last access 20 October 2017.

[65] Philippe Tillet, Karl Rupp, and Siegfried Selberherr. An automatic OpenCL compute kernel generator for basic linear algebra operations. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 4:1–4:2, San Diego, CA, USA, 2012. Society for Computer Simulation International. ISBN 978-1-61839-788-1. URL http://dl.acm.org/citation.cfm?id=2338816.2338820.

[66] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990. ISSN 0098-3500. doi: 10.1145/77626.79170. URL http://doi.acm.org/10.1145/77626.79170.

[67] Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.

[68] K. Matsumoto, N. Nakasato, and S. G. Sedukhin. Implementing a code generator for fast matrix multiplication in OpenCL on the GPU. In *2012 IEEE 6th International Symposium on Embedded Multicore SoCs*, pages 198–204, Sept 2012. doi: 10.1109/MCSoC.2012.30.

[69] Zhang Xianyi, Wang Qian, and Zaheer Chothia. OpenBLAS. *URL: http://xianyi. github. io/OpenBLAS*, 2014.

[70] Nugteren Cedric and Codreanu Valeriu. CLTune: A Generic Auto-Tuner for OpenCL Kernels. *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 195–202, 2015. doi: doi.ieeecomputersociety.org/10.1109/MCSoC.2015.10.

[71] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[72] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011. ISBN 0123814790, 9780123814791.

[73] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[74] Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammerling, James Demmel, C Bischof, and Danny Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press, 1990.

[75] Jaeyoung Choi, James Demmel, I. Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clinton Whaley. Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, PARA '95, pages 95–106, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-60902-4. URL http://dl.acm.org/citation.cfm?id=645779.666179.

[76] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

[77] Karl Rupp, Florian Rudolf, and Josef Weinbub. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*, pages 51–56, 2010.

[78] Yinan Li, Jack Dongarra, and Stanimire Tomov. *A Note on Auto-tuning GEMM for GPUs*, pages 884–892. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-01970-8. doi: 10.1007/978-3-642-01970-8_89. URL https://doi.org/10.1007/978-3-642-01970-8_89.

[79] Junjie Lai and André Seznec. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[80] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning GEMM for GPUs. *Computational Science–ICCS 2009*, pages 884–892, 2009.

[81] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G Sedukhin. Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 396–405. IEEE, 2012.

[82] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, May 2012. doi: 10.1109/InPar.2012.6339587.

[83] B. Cosenza, J. J. Durillo, S. Ermon, and B. Juurlink. Autotuning stencil computations with structural ordinal regression learning. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 287–296, May 2017. doi: 10.1109/IPDPS.2017.102.

[84] Thomas L. Falch and Anne C. Elster. Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications. *Concurrency and Computation: Practice and Experience*, 29(8):e4029–n/a, 2017. ISSN 1532-0634. doi: 10.1002/cpe.4029. URL http://dx.doi.org/10.1002/cpe.4029. e4029 cpe.4029.

[85] Thomas L Falch and Anne C Elster. Machine learning based auto-tuning for enhanced OpenCL performance portability. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 1231–1240. IEEE, 2015.

[86] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. *SIGPLAN Not.*, 50(6):379–390, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737969. URL http://doi.acm.org/10.1145/2813885.2737969.

[87] Kaixi Hou, Wu-chun Feng, and Shuai Che. Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi-and many-core processors. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 713–722. IEEE, 2017.

[88] Alberto Magni, Dominik Grewe, and Nick Johnson. Input-aware auto-tuning for directive-based GPU programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 66–75, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2017-7. doi: 10.1145/2458523.2458530. URL http://doi.acm.org/10.1145/2458523.2458530.

[89] Philippe Tillet and David Cox. Input-aware auto-tuning of compute-bound hpc kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 43:1–43:12, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5114-0. doi: 10.1145/3126908.3126939. URL http://doi.acm.org/10.1145/3126908.3126939.

[90] G. Fursin, A. Lokhmotov, and E. Plowman. Collective Knowledge: Towards R&D sustainability. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 864–869, March 2016.

[91] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[92] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR*, abs/1602.07360, 2016.

[93] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, Jan 2011. ISSN 1045-9219. doi: 10.1109/TPDS.2010.107.