CAMPUS
DE EXCELENCIA
INTERNACIONAL

Grado en Matemáticas e Ingeniería Informática

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros Informáticos

TRABAJO DE FIN DE GRADO

Implementing Fully Homomorphic Encryption Schemes in FPGA-based Systems

Autor: Alejandro Ranchal Pedrosa
Director: Manuel Carro Liñares

MADRID, ENERO 2016

Contents

List of Figures

List of Tables

Chapter 1

Summary

1.1 Spanish

Las nuevas tecnologías orientadas a la nube, el internet de las cosas o las tendencias "as a service" se basan en el almacenamiento y procesamiento de datos en servidores remotos. Para garantizar la seguridad en la comunicación de dichos datos al servidor remoto, y en el manejo de los mismos en dicho servidor, se hace uso de diferentes esquemas criptográficos. Tradicionalmente, dichos sistemas criptográficos se centran en encriptar los datos mientras no sea necesario procesarlos (es decir, durante la comunicación y almacenamiento de los mismos). Sin embargo, una vez es necesario procesar dichos datos encriptados (en el servidor remoto), es necesario desencriptarlos, momento en el cual un intruso en dicho servidor podría a acceder a datos sensibles de usuarios del mismo. Es más, este enfoque tradicional necesita que el servidor sea capaz de desencriptar dichos datos, teniendo que confiar en la integridad de dicho servidor de no comprometer los datos. Como posible solución a estos problemas, surgen los esquemas de *encriptación homomórficos completos*.

Un esquema homomórfico completo no requiere desencriptar los datos para operar con ellos, sino que es capaz de realizar las operaciones sobre los datos encriptados, manteniendo un homomorfismo entre el mensaje cifrado y el mensaje plano. De esta

manera, cualquier intruso en el sistema no podría robar más que textos cifrados, siendo imposible un robo de los datos sensibles sin un robo de las claves de cifrado.

Sin embargo, los esquemas de encriptación homomórfica son, actualmente, drásticamente lentos comparados con otros esquemas de encriptación clásicos. Una operación en el anillo del texto plano puede conllevar numerosas operaciones en el anillo del texto encriptado. Por esta razón, están surgiendo distintos planteamientos sobre como acelerar estos esquemas para un uso práctico.

Una de las propuestas para acelerar los esquemas homomórficos consiste en el uso de *High-Performance Computing (HPC)* usando *FPGAs (Field Programmable Gate Arrays)*. Una FPGA es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser reprogramada. Al compilar para FPGAs, se genera un circuito hardware específico para el algorithmo proporcionado, en lugar de hacer uso de instrucciones en una máquina universal, lo que supone una gran ventaja con respecto a CPUs. Las FPGAs tienen, por tanto, claras difrencias con respecto a CPUs:

- Arquitectura en pipeline: permite la obtención de outputs sucesivos en tiempo constante
- Posibilidad de tener multiples pipes para computación concurrente/paralela.

Así, en este proyecto:

- Se realizan diferentes implementaciones de esquemas homomórficos en sistemas basados en FPGAs.
- Se analizan y estudian las ventajas y desventajas de los esquemas criptográficos en sistemas basados en FPGAs, comparando con proyectos relacionados.
- Se comparan las implementaciones con trabajos relacionados

1.2 English

New cloud-based technologies, the internet of things or "as a service" trends are based in data storage and processing in a remote server. In order to guarantee a secure communication and handling of data, cryptographic schemes are used. Traditionally, these cryptographic schemes focus on guaranteeing the security of data while storing and transferring it, not while operating with it. Therefore, once the server has to operate with that encrypted data, it first decrypts it, exposing unencrypted data to intruders in the server. Moreover, the whole traditional scheme is based on the assumption the server is reliable, giving it enough credentials to decipher data to process it. As a possible solution for this issues, *fully homomorphic encryption(FHE) schemes* is introduced.

A fully homomorphic scheme does not require data decryption to operate, but rather operates over the cyphertext ring, keeping an homomorphism between the cyphertext ring and the plaintext ring. As a result, an outsider could only obtain encrypted data, making it impossible to retrieve the actual sensitive data without its associated cypher keys.

However, using homomorphic encryption(HE) schemes impacts performance drastically, slowing it down. One operation in the plaintext space can lead to several operations in the cyphertext space. Because of this, different approaches address the problem of speeding up these schemes in order to become practical.

One of these approaches consists in the use of *High-Performance Computing (HPC)* using *FPGAs (Field Programmable Gate Array)*. An FPGA is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable". Compiling into FPGA means generating a circuit (hardware) specific for that algorithm, instead of having an universal machine and generating a set of machine instructions. FPGAs have, thus, clear differences compared to CPUs:

- Pipeline architecture, which allows obtaining successive outputs in constant

time.

- Possibility of having multiple pipes for concurrent/parallel computation.

Thereby, In this project:

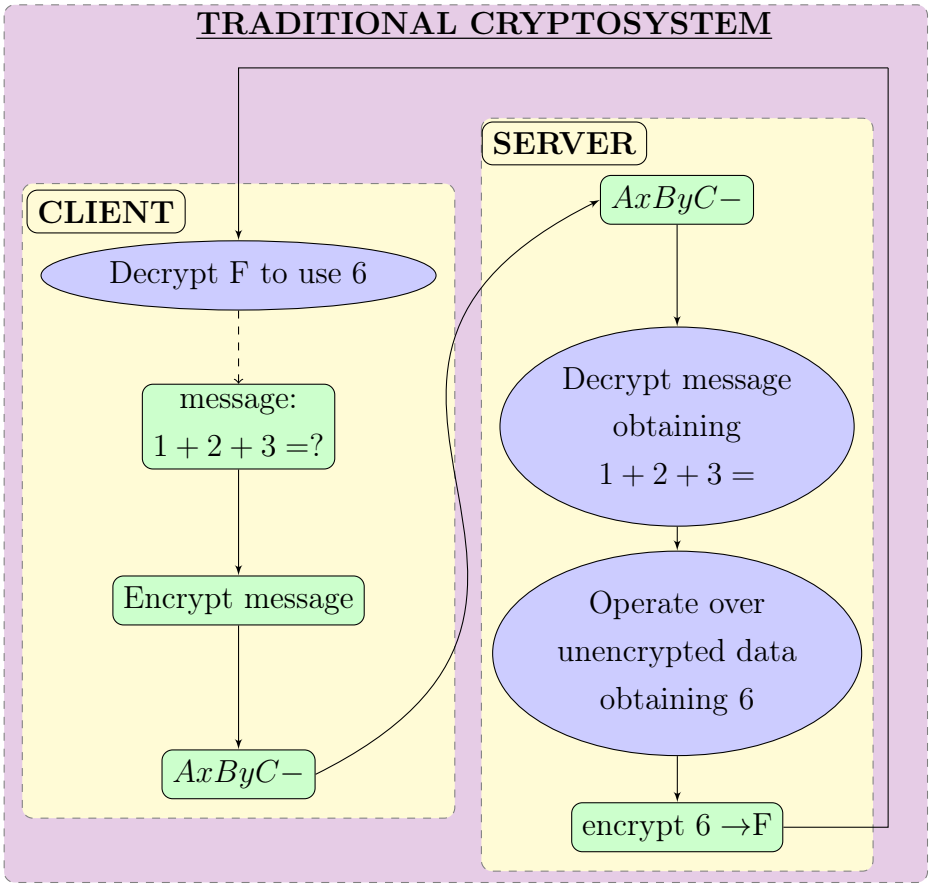
- We present different implementations of FHE schemes in FPGA-based systems.
- We analyse and study advantages and drawbacks of the implemented FHE schemes, compared to related work.

Chapter 2

Introduction

Cryptography, the use of codes and ciphers to protect data has been used for centuries. Cryptographic methods have traditionally focused on encrypting data only when it is stored. Therefore, whenever someone wants to use such encrypted data, decryption needs to be done first, after which one can use the data for whatever purposes and encrypt the result (if stored).

With the rise of the Internet and "as a Service" approaches, however, this traditional workflow seems inappropriate. As shown in figure 2.1, traditional encryption systems ensure the data is secure throughout the communication with the server. Once the data is in the server and needs to be used for arithmetic operations or program executions, it needs to be decrypted. For instance, in a client-server model, as soon as the client data is transferred to the server and decrypted there, that data can be seen by any intruder in the server, as well as by the server itself (which might not be reliable). If the client has high security requirements because the data he is sharing is private and sensible then not use these traditional cryptosystems inappropriate. Besides, the data might be confidential and whoever or whatever will perform the computation in the server is not be trusted. From the server point of view, ensuring high levels of security ensures more clients would be more willing to share more data and, therefore, use more the servers.



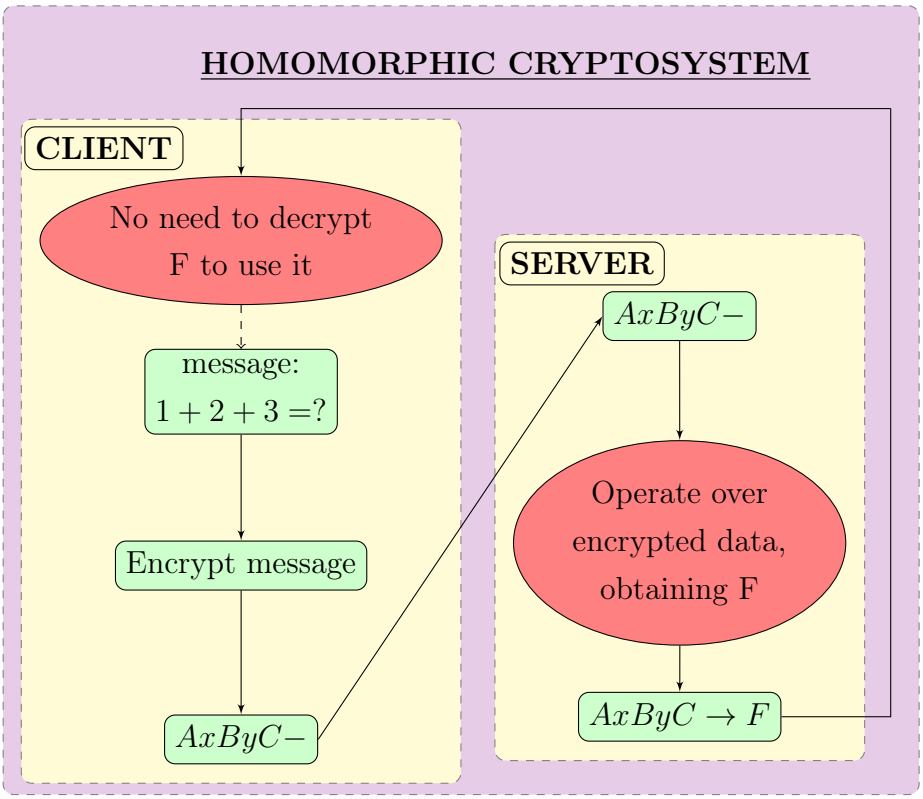


Figure 2.1: Client-server communication using HE and a traditional approach.

A possible solution to the previous issue would be to follow the protocol below:

1. Client encrypts data
2. Client sends encrypted data to server
3. **Server computes over encrypted data**
4. Client receives encrypted Output
5. Client decrypts output, receiving the same answer as he would have gotten if he had sent unencrypted data.

Since traditional approaches cannot compute over encrypted data (step 3), *Homomorphic Encryption (HE)* is introduced. From the mathematical point of view, encrypting is roughly applying an injective application $\varphi : M \to \mathcal{M}$, M being the set of elements of the unencrypted message (plaintext) and $\mathcal{M}$ the set of elements of the encrypted message (ciphertext). Following the previous mathematical definition, an *Homomorphism* between M and $\mathcal{M}$ implies that for any given $a, b \in M$ (that is, for two messages), and a defined operation over elements in M (that is, a desired computation over the two messages), $*_M$, then $\varphi(a *_M b) = \varphi(a) *_{\mathcal{M}} \varphi(b)$. In other words, the computation over two messages can be performed by encrypting both of them, doing a different computation and decrypting. As the definition of homomorphism requires one operation per set, we speak of homomorphism over groups (if only one operation) or rings (if each of the sets have two operations). Taking the mathematical definition into the computation, the two operations refer to multiplication and addition, with equality (since the application is injective). Any known program can be rewritten using only these three operations, being possible therefore to encrypt a whole program using HE and compute it without knowing the actual computation.

However, using HE, one operation in the plaintext space can lead to several operations in the cyphertext space. Consequently, HE and decryption operations take a rather significant amount of time compared to non-Homomorphic approaches. For this reason, great effort goes to speeding up HE.

One of the approaches to speeding up Homomorphic schemes consists of using *High-Performance Computing (HPC)* using *FPGAs (Field Programmable Gate Array)*, which covers *HPRC (High-Performance Reconfigurable Computing)*. An FPGA is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable". FPGAs have, thus, clear advantages for some algorithms , being the main one that, once a specific algorithm is compiled into FPGA, a specific hardware circuit is generated, instead of generating instructions to be interpreted. The generated circuit is made of a finite set of different modules that can be nested together in different ways and controlled through signals in order to perform the specific algorithm. FPGAs use a pipeline architecture, which allows obtaining successive outputs in constant time and gives the possibility of having multiple pipelines in parallel.

As a result of these differences between CPUs and FPGAs, the latter have specific requirements that complicate programming for them. Furthermore, because compiling for FPGAs means programming a circuit, it is necessary to control circuit signals as well as to design an interface for CPU-FPGA communication. Also, some operations that can be performed in constant time for CPUs, such as accessing to a random position in an array, are, if not impossible, rather expensive in terms of time and resources. Finally, if the circuit to be generated for some algorithm requires more modules than available in the specific board used, such algorithm simply cannot be implemented in that specific FPGA. Consequently, the goal is optimizing execution times and resources in a challenging programming paradigm and environment.

Chapter 3

Architecture Overview

As explained during the Introduction, a way of reducing the time consumption of using FHE cryptosystems is through *FPGAs (Field Programmable Gate Array)*, which covers *HPRC (High-Performance Reconfigurable Computing)*. An FPGA is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field programmable". FPGAs offer clear advantages for some algorithms:

- pipeline architecture, which allows obtaining successive outputs in constant time.
- Possibility of having multiple pipes for concurrent/parallel computation.

All industries have adopted FPGA-based systems since they offer hardware-timed speed and reliability the same way *application-specific integrated circuits* (ASIC) do, but they do not require costs higher than processor-based systems.

Every FPGA is made of a finite number of modules interconnected in a reconfigurable way. These modules are basically of four different types:

- Flip-Flops (FFs): Flip-flops are binary shift registers used to synchronize logic and save logical states between clock cycles within an FPGA circuit. On every

clock edge, a flip-flop latches the 1 or 0 (TRUE or FALSE) value on its input and holds that value constant until the next clock edge.

- Lookup tables (LUTs): LUTs are the essence of the logic units in the FPGA. They are basically just a truth table that can be reconfigured depending on the combinatorial logic needed (AND,XOR,OR,...). The programmer can also configure its own lookup tables.

- DSPs: given the high resource consumption of implementing multipliers using combinatorial logic, DSPs are prebuilt multiplier circuits.

- Blocks of RAM (BRAMs): Another key specification when programming for FPGAs is the memory resource. BRAMs can be combined into a larger structure or used separated enhancing the parallelism within FPGAs.

The basic logic unit of an FPGA is referred to as *configurable logic blocks*(CLB), also known as slices or logic cells, they are made from FFs and LUTs.

As previously mentioned, parallel pipes are one of the advantages of using FPGAs instead of processor-based systems. As long as the parallelized algorithm fits within the resources of the target FPGA board, such algorithm can be implemented efficiently. As an example , the datapaths shown in figure 4.2 can be implemented as a parallel circuit, having each of the coefficients running parallel, assuming that amount of multiplications and logic is small enough to fit in the FPGA resources.

Thanks to their pipeline architecture, FPGAs can benefit from *instruction pipelining*, illustrated in figure 3.1, which enables different data to queue through the circuit, enabling a faster throughput.

Data: **x**
Result: **y**
for $i = 1$ *to* $length(x) - 2$ **do**
  $y_i \leftarrow (x_i + x_{i-1} + x_{i+1})/3$;
end
return y;

Algorithm 1: Sample algorithm for figure 3.1.

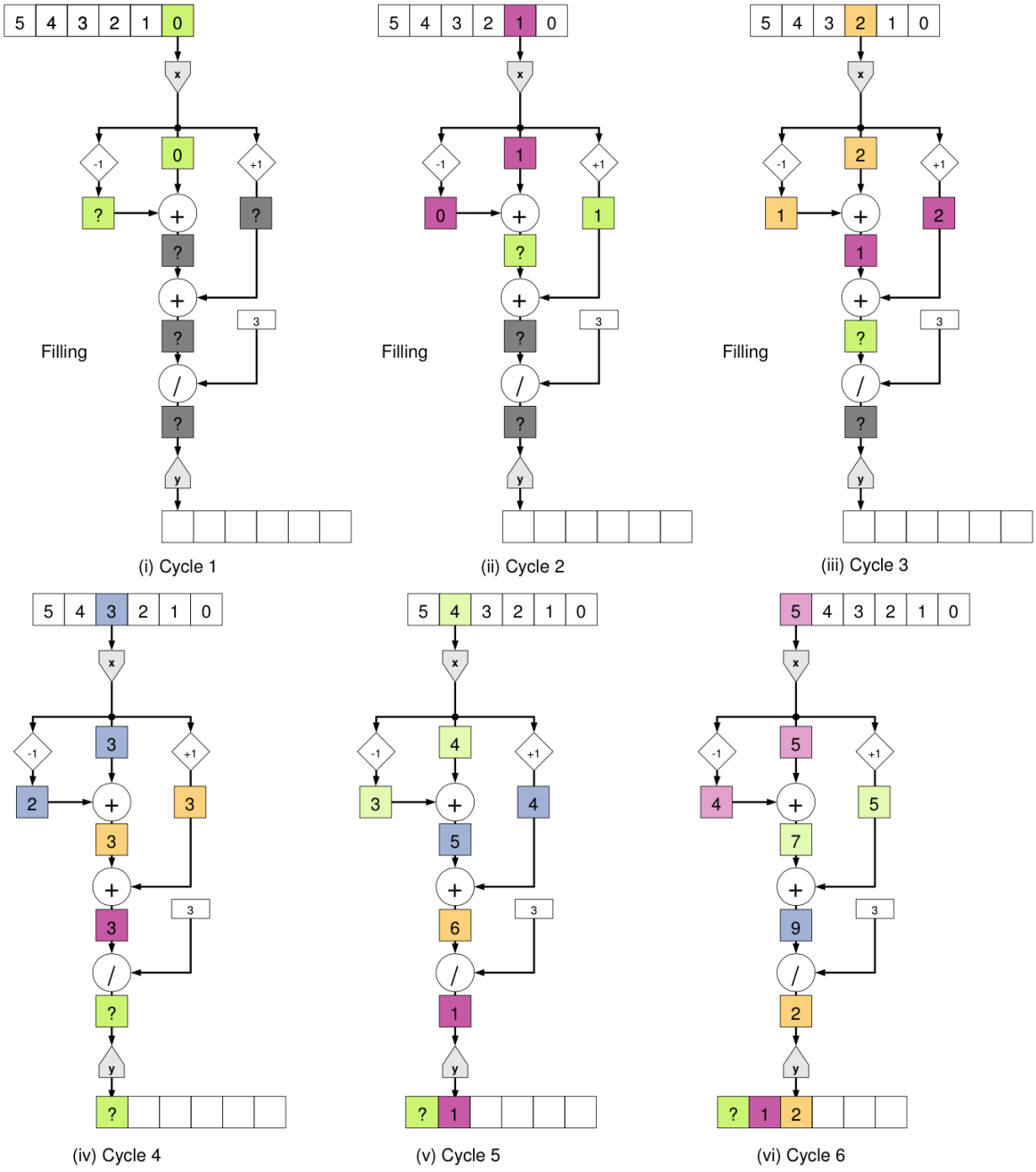


Figure 3.1: Circuit result of implementing algorithm 1 in a FPGA. Note the pipeline architecture allows the elements to advance in the circuit at the same cycle (tick). The diamonds represent accessing to the current position plus an offset, the squares are just the values at that point of the circuit, the circles represent operations and the upper and lower sequence of numbers represent the input and output streams, respectively.

Figure 3.1 shows a circuit for algorithm 1. A *tick* is the term used when referring to a cycle in the FPGA execution. A tick can be thought of as a high-level clock tick in an ideal FPGA in which every native operation takes exactly one tick to execute. As such, figure 3.1 illustrates 6 ticks (or cycles) of algorithm 1. Notice during the first 2 ticks the circuit performs no operation whatsoever, since it needs to fill the offset -1, corresponding the x_{i-1} index in algorithm 1 (illustrated as a diamond with -1 inside).

One can find a good illustration of instruction pipelining in the tick 5 of figure 3.1. Prior to tick 5, no real output has been generated yet. However, by the end of tick 5, the first output, 1, is computed. At the same time, the result 5 is computed from the first adder, the result 6 from the second one and the result 1 from the division by 3. Since the first real output is calculated at tick 5, the *latency* (the number of ticks from the first input to the first output) of algorithm 1 is 5. Note that, after 5 (the latency) ticks, each output elements comes in constant time, in this case one per tick.

Besides the previously explained, one of the most interesting advantages of FPGAs compared to CPUs has to do with its energy efficiency. When Fowers et al. [14] compared the same algorithm in three different average targeted devices (CPU,GPU and FPGA), they found that, being FPGA the fastest of them, it is at the same time the most energy efficient. More recently, Microsoft is reporting a 3x higher energy efficiency when using FPGAs as opposed to NVIDIA hardware [22]. However, as Mittal et al. [19] recently stated when observing a general energy efficiency hierarchy, even though FPGAs are clearly more energy efficient than GPUs and CPUs in a big majority of works, the energy consumption greatly depends on the algorithm and its specific implementation in each of the three units.

However, programming for FPGAs has some drawbacks that should be remarked. The main drawback is its resource limitation. An FPGA has a number of resources (FFs,LUTs,DSPs,BRAMs), which means that an algorithm can be executed by a

specific FPGA only if the amount of resources needed is lower than the amount of resources available in the FPGA, obviating resource-related workarounds (such as using LUTs to implement Multipliers instead of DSPs). Also, loops or if-then control statements cannot be directly coded in FPGAs. Finally, random access to arrays (data streams) is also not possible (apart from some specific exceptions which will be detailed in section 5). This, among many other FPGA-specific programming requirements, makes programming for FPGA a much more difficult task than just adapting the algorithm from CPU.

Storing values within ticks, though possible using BRAMs, is not simple given the impossibility of having more than two ports per RAM. Also, these ports cannot point at the same memory position in the same tick. This basically means that one can only read or write from two positions in each tick, not being possible to read from the same position compute and store in the same position later. Also, compared to CPUs and GPUs, FPGAs have low clockspeed, ranging from 60-300 MHz.

3.1 Maxeler IDE and MaxJ Language

When programming for FPGAs, there are some possible options. Two of the most used programming languages are Verilog and VHDL (VHSIC Hardware Description Language), syntactically similar to C and Assembly, respectively. Nevertheless, these languages need to control hardware logic and signals, being clearly low-level. As any other low-level language, they are tedious to program and maintain, being perhaps not the best option when developing complex algorithms. Other option is to use a cross-compiler generating VHDL or Verilog code out of a different language, such as *ROCCC*[5], a C-to-HDL compiler. However, one of the most tedious tasks when programming for FPGAs is not only generating FPGA-compatible code but also interfacing with CPU. Other options similar to ROCCC are Bambu[1] (C-to-Verilog), Nios-II C2H Acceleration Compiler [4] (by Altera) or FpgaC[2].

Maxeler Technologies [3] offers a set of tools, a Java-like language for FPGAs,

named MaxJ; MaxJ-to-HDL compiler with built-in interface to CPU through PCIe (a serial bus standard used for high speed communication between devices); an extensive documentation, debugging tools and FPGAs (hardware) already prepared to work out of the box with some additional plugins (such as intercommunication between FPGAs using shared Memory) in what they referred to as a *Dataflow Engine* (DFE).

Maxeler recommends using MaxIDE,an integrated development environment based on Eclipse IDE. Every Maxeler project consists of three main parts:

- **Engine Code**: it is the code executed by the DFE (i.e. by the FPGAs), and the code that defines its configuration and the type of interface with CPU.
- **CPU Code**: it is the code executed by the CPU, in which there should be an specific invocation to the Engine Code.
- **Run Rule**: it defines the different compile options (Simulation or DFE, name of the executable, etc.).

DFEs consist of a set of FPGAs and two types of memory: *FMem* (Fast Memory) which can store several megabytes of data on-chip with terabytes/second of access bandwidth and *LMem* (Large Memory) which can store many gigabytes of data off-chip.

The dataflow engine is programmed with one or more Kernels and a Manager. Kernels implement computation while the Manager controls data movement within the DFE and with CPU. Given Kernels and a Manager, MaxCompiler generates dataflow implementations which can then be called from the CPU via the SLiC interface. The SLiC (Simple Live CPU) interface is an automatically generated interface to the dataflow program, making it easy to call dataflow engines from attached CPUs. This interface can be configured within the Manager. The overall system is managed by MaxelerOS, which sits within Linux and also within the Dataflow Engine's manager. MaxelerOS manages data transfer and dynamic optimization at runtime.

Code 3.1: AdderManager.maxj

```
package adder;
class AdderManager {
    public static void main(String[] args) {
      Manager manager = new Manager(new EngineParameters(args));
      Kernel kernel = new AdderKernel(manager.makeKernelParameters());
      manager.setKernel(kernel);
      manager.setIO(IOType.ALL_CPU);
      manager.createSLiCinterface();
      manager.build();
    }
}
```

Code 3.2: AdderKernel.maxj

```
package adder;
class AdderKernel extends Kernel {

    AdderKernel(KernelParameters parameters)
    {
      super(parameters);

      // Input
      DFEVar x = io.input("x", dfeUInt(32));
    DFEVar y = io.input("y", dfeUInt(32));


      // Output
      io.output("z", x+y, dfeUInt(32));
    }
}
```

Code 3.3: adder.c

```
uint32_t dataIn[1024];
uint32_t dataIn2[1024];
uint32_t dataOut[1024];
const int size = 1024;

void main()
{
    for(int i = 0; i < size; i++) {
      dataIn[i] = i;
    dataIn2[i] = size-i-1;
      dataOut[i] = 0;
    }
    printf("Running DFE.\n");
    Adder(size, dataIn, dataIn2, dataOut);
}
```

Code 3.4: adder_roccc.c

```
typedef int ROCCC_int32 ;
```

```
2         void Adder(ROCCC_int32 in,ROCCC_int32 in2, ROCCC_int32& out)
3         {
4             out=in+in2;
5         }
```

Codes 3.1,3.2 and 3.3 illustrates the code of an example of adding two streams using Maxeler tools. Also, code 3.5 shows the same example in VHDL, without interfacing with CPU, generated by the subset of C needed for ROCCC, also shown in code 3.4. Note that, although ROCCC C takes few lines of code, the generated VHDL has to deal with signals (clk, rst, etc.). As ROCCC does not generate an interface VHDL-CPU, using the generated VHDL code would mean understanding each of the signals and write code for the interface. Maxeler generates such interface automatically with the code in the Manager. At the same time, the Kernel implements the addition itself, offering a code easy to understand and develop.

Code 3.5: adder.vhdl

```
1         library IEEE ;
2         use IEEE.STD_LOGIC_1164.all ;
3         use IEEE.STD_LOGIC_ARITH.ALL;
4         use IEEE.STD_LOGIC_UNSIGNED.all ;
5         use work.HelperFunctions.all;
6         use work.HelperFunctions_Unsigned.all;
7         use work.HelperFunctions_Signed.all;
8
9         entity PassThrough is
10        port (
11          clk : in STD_LOGIC;
12          rst : in STD_LOGIC;
13          inputReady : in STD_LOGIC;
14          outputReady : out STD_LOGIC;
15          done : out STD_LOGIC;
16          stall : in STD_LOGIC;
17          in_in : in STD_LOGIC_VECTOR(31 downto 0);
18          in2_in : in STD_LOGIC_VECTOR(31 downto 0);
19          out_out_out : out STD_LOGIC_VECTOR(31 downto 0)
20          );
21        end entity;
22
23        architecture Synthesized of PassThrough is
24        signal in21024 : STD_LOGIC_VECTOR(31 downto 0) ;
25        signal out_out1428 : STD_LOGIC_VECTOR(31 downto 0) ;
26        signal in719 : STD_LOGIC_VECTOR(31 downto 0) ;
27        signal stall_previous : STD_LOGIC ;
28        signal activeStates : STD_LOGIC ;
29        signal PassThrough_done : STD_LOGIC ;
30        begin
31        done <= PassThrough_done;
```

```
32          out_out1428 <= ROCCCADD(in719, in21024, 32);
33          process(clk, rst)
34          begin
35          if (rst = '1') then
36          stall_previous <= '0';
37          activeStates <= '0';
38           in719 <= "00000000000000000000000000000000";
39           in21024 <= "00000000000000000000000000000000";
40           outputReady <= '0';
41           PassThrough_done <= '0';
42           out_out_out <= "00000000000000000000000000000000";
43           elsif( clk'event and clk = '1' ) then
44           outputReady <= '0';
45           stall_previous <= stall;
46          if( ((stall /= '1') or (stall_previous = '0')) ) then
47           activeStates <= inputReady;
48          end if;
49          if( ((inputReady = '1') and ((stall /= '1') or (stall_previous = '0'))) )
                then
50           in719 <= in_in;
51           in21024 <= in2_in;
52          end if;
53          if( ((activeStates = '1') and ((stall /= '1') or (stall_previous = '0')))
                ) then
54          end if;
55          if( ((activeStates = '1') and ((stall /= '1') or (stall_previous = '0')))
                ) then
56           outputReady <= '1';
57           PassThrough_done <= '1';
58           out_out_out <= out_out1428;
59          end if;
60          end if;
61          end process;
62          end Synthesized;
```

Chapter 4

Background & Related Work

Our project implements a somewhat homomorphic version of the Brakerski-Gentry-Vaikuntanathan cryptosystem (BGV) [9] inspired by Fiore et al. [13]. Details of this scheme will be explored later in this section (see section 4.5). The most time-consuming part of the BGV scheme is the *Polynomial Multiplication (PM)*, being critical speeding it up in order to speed up the whole scheme. The PM we implement, explained in section 4.2, uses the *Number Theoretic Transform (NTT)* to reduce its complexity to $O(n\log(n))$, compared to the quadratic complexity ($O(n^2)$)) of a straight forward approach, described in section 4.1. Also, to enable new levels of parallelism when using the, one can use the *Chinese Remainder Theorem (CRT)* so as to split the computation of one big PM into several small PMs, illustrated in section 4.3. Each of the multiplication of two integers over a bounded ring, such as $\mathbb{Z}_p$, requires the result to be reduce by the value that bounds it (p, in this case). Section 4.4 explores different modular reductions. Finally, in order to be able to execute for a wide range of different spaces bounded by different numbers and with different configurations, we spent some time into developing an algorithm to generate different precomputed parameters that define the whole context of the cryptosystem, detailed in section 4.6.

The BGV cryptosystem allows HE using as plaintext space $\mathcal{M}$ a ring (that is, a set of elements and two operations over them that represent addition and multiplication)

$R_p := \mathbb{F}_p[X]/\Phi_m(X)$, being $\Phi_m(X)$ the m-th cyclotomic polynomial. As we have a quotient ring (that is, a ring reducted by a generator, $\Phi_m(X)$ in this case), the maximum degree of polynomial is the degree of $\Phi_m(X)$, being that degree $\Phi(m)$ (the Euler's totient function). However, as we will see later in section 4.1, the chosen PM algorithm needs the polynomial length to be of maximum degree of the form $2^n - 1,\ n \in \mathbf{Z}$, so that the number of coefficients is a power of two, which means that $\Phi_m(X)$ must be of degree 2^n. Consequently, we have $\Phi(m) = 2^n \iff m = 2^{n+1}$. Finally, as $\Phi_m(X) = \Phi_{2^{n+1}}(X)$ one can see that $\Phi_{2^{n+1}}(X) = x^{2^n} + 1$. Also, the notation $N = 2^n$ will will be used, which means $\Phi_{2^{n+1}}(X) = x^{2^n} + 1 = x^N + 1$. The operations of this ring are denoted with $+$ *and* $\cdot$ for polynomial addition and multiplication, respectively. For the implementation, elements in $\mathcal{M}$ are represented as elements in $\mathbb{Z}^n \simeq \mathbb{Z}[X]/\Phi_m(X)$ with infinity norm bounded by p; in other words, represented as elements in $\mathbb{Z}_p^n$.

Similarly, ciphertext elements are elements in a ring $R_q[Y]$ such that:

1. $R_q = \mathbb{Z}/q\mathbb{Z}[X]/\Phi_m[X]$. In other words, elements in R_q are polynomials of maximum degree $N - 1$ and with coefficients of value smaller than q.
2. $gcd(q, p) = 1$ (q and p are co-prime).
3. q is big enough (compared to p) to satisfy lemma 1 of [13].

Operations on the ciphertext space (also denoted with $+$, $\cdot$) are as follows:

$$\sum_{i=0}^{\infty} a_i Y^i \ + \ \sum_{i=0}^{\infty} b_i Y^i = \sum_{i=0}^{\infty} (a_i + b_i) Y^i \,; \tag{4.1}$$

$$\sum_{i=0}^{\infty} a_i Y^i \ \cdot \ \sum_{i=0}^{\infty} b_i Y^i = \sum_{i=0}^{\infty} \sum_{j=0}^{i} a_j b_{i-j} Y^i \tag{4.2}$$

One should notice that the coefficients a_i in equation 4.2 are elements in R_q. That is, the multiplication shown is over elements in $R_q[Y]$, which is the space of the

encrypted messages. As such, elements in $R_q[Y]$ are polynomials which coefficients are, in turn, polynomials. One can notice that multiplying two elements in $R_q[Y]$ increments the degree of the polynomial in Y. However, in order to keep this degree low, the main focus lies on the polynomial addition. This means that this work focuses in level 1 ciphertexts which means that the elements in $R_q[Y]$ that we are going to work with are elements of the form $a_0 + a_1 Y$, with $a_0, a_1 \in R_q$. Multiplying two level 1 ciphertexts would return a level 2 ciphertext, (with degree 2), which is not in the scope of this project. It is perfectly possible to work with bigger levels of ciphertexts, focusing in multiplication as much as in addition. However, because the degree of the resulting polynomial increases when multiplying, it should be necessary to consider how to keep the degree of the polynomial as low as possible. Such task should have to be performed entirely in CPU, without affecting encryption nor decryption of the BGV Cryptosystem. Moreover, this implementation in CPU has already been studied by other works, such as Fiore et al. [13] or HELib [28]. Consequently, ciphertexts of levels higher than one are not the main focus of this work. It is for this reason that the presented implementation of the BGV is referred to as *somewhat homomorphic*. Nevertheless, this is an habitual practice when measuring speedup for HE cryptosystems given that speeding up the computation for level 1 ciphertexts is directly related with speeding up for higher levels ciphertexts.

4.1 Number Theoretic Transform

The PM is the most consuming part during the encryption. Compared to the quadratic complexity of the straight forward approach for the PM, using the *Fast Fourier Transform* (FFT) for PM offers a linearithmic time approach (that is, its complexity is $O(n \log(n)))$. In turn, the FFT, and its inverse (IFFT), are the core operations within the PM. Therefore, our work focus on speeding up the FFT-IFFT operations in each PM, which ultimately speeds up the whole BGV cryptosystem. As one can see in figure 4.1, the complexity of performing the FFT is $O(n \log(n))$, dominating over the actual multiplication of two polynomials, which can be performed element-wise (that is, only multiply coefficients of the same degree) with a complexity

of $O(n)$, resulting in an overall complexity of $O(n\log(n))$, compared to the complexity $O(n^2)$ of directly multiplying $a \cdot b$ instead of multiplying $FFT(a) * FFT(b)$, being $*$ the element-wise multiplication.

Given the chosen representation, the FFT algorithm has to operate over elements of a finite field of the form $\mathbb{Z}_q[X]/\Phi_m(X)$, being q prime. This type of FFT is generally referred to as NTT. Notice the NTT operates over coefficients of the ciphertext $R_q[Y]$, being q relatively prime to p. That is, it operates over elements in R_q. Thus, the NTT algorithm over finite fields restricts q to be prime. There are possible workarounds for using co-primes, which will be explored later in this section (see section 4.3). Also, there are several approaches when implementing the NTT ([12],[24],[17],[25]). Typically, non-recursive NTT algorithms consist of two loops. Iterations of the outer loop are commonly known as "stages", whereas iterations of the inner loop are referred to as "iterations" of the respective stage. It is worth mentioning that in some Cooley-Tukey's [12] implementation there are two outer loops which perform the stages. Moreover, the notation NTT_{ω}^{N} is generally used for the NTT of polynomials of degree N using the primitive Nth root of unity ω, that is, a value ω such that $\omega^N \equiv 1 \ mod\, p$, $\omega^r \not\equiv 1 \, mod\, q$; $0 < r < N$, necessary for the computation of the NTT.

Pease's [24] and Cooley-Tukey's (see figure 4.1) are two of the most common NTT algorithms. Pease's constant geometry (i.e.,regular input/output patterns in the inner **for** loop) eases the programming complexity of the algorithm in the FPGA, optimizing FPGA resources (in comparison with Cooley-Tukey's, approach), which can be used to speed up parallelizing later. However, the patterns between the stages the stages of Cooley-Tukey's are better suited for algorithm devices with pipeline architectures (such as the FPGA) which, again, optimizes FPGA resource usage (see figure 4.2).

Figure 4.1 shows both FFT (NTT) approaches. It should be remarked that, to obtain the indexes in natural order, at the end of FFT-IFFT it is necessary to place

the input vector in bit reversed order of the indexes. That is, given a polynomial $a(x)$ of degree $N = 2^n$, the elements in positions i and j, where i and j are bit-reversed, are swapped. This operation is performed calling $Bit\_reverse(polynomial)$ in the algorithms (see figure 4.1).

Data: $\mathbf{a}$,ω,ω^{-1},n,p,N=2^n
Result: $\mathbf{A}$=$NTT_w^N(\mathbf{a})$
$\mathbf{a} \leftarrow Bit\_reverse(\mathbf{a})$;
for *i=0* ***to*** $n-1$ **do**
  for *j=0* ***to*** $N/2 - 1$ **do**
    $P_{ij} \leftarrow \lfloor \frac{j}{2^{n-1-i}} \rfloor \cdot 2^{n-1-i}$;
    $u \leftarrow a_{2j}$;
    $t \leftarrow a_{2j+1} \cdot \omega^{P_{ij}}$;
    $A_j \leftarrow u + t$;
    $A_{j+N/2} \leftarrow u - t$;
  end
  if $i \neq n-1$ **then**
    $\mathbf{a} \leftarrow \mathbf{A}$
  end
end
return $\mathbf{A}$;
Algorithm 2: Pease NTT

Data: $\mathbf{a}$,ω,ω^{-1},n,p,N=2^n
Result: $\mathbf{A}$=$NTT_w^N(\mathbf{a})$
$\mathbf{a} \leftarrow Bit\_reverse(\mathbf{a})$;
for $i = 2$ ***to*** N ***by*** $i = 2i$ **do**
  $\omega_i \leftarrow \omega^{N/i}$;
  $\omega \leftarrow 1$;
  for *m=0* ***to*** $i/2 - 1$ **do**
    for *j=0* ***to*** $N-1$ *by* i **do**
      $u \leftarrow a_{j+m}$;
      $t \leftarrow a_{j+m+i/2} \cdot \omega$;
      $A_{j+m} \leftarrow u + t$;
      $A_{j+m+i/2} \leftarrow u - t$;
    end
    $\omega \leftarrow \omega \cdot \omega_i$;
  end
  $\mathbf{a} \leftarrow \mathbf{A}$
end
return $\mathbf{A}$;
Algorithm 3: Cooley-Tukey NTT

Figure 4.1: Pease's (left) and Cooley-Tukey's(right) NTT(i.e. FFT over finite fields of the form $\mathbb{Z}_q[X]/\Phi_m(X)$) algorithms

Regardless of the specific NTT in $\mathbb{Z}_q[X]/\Phi_m(X)$, it is necessary to determine a primitive Nth root of unity ω such that: $\omega^N \equiv 1 \, mod \, p$, $\omega^r \not\equiv 1 \, mod \, q$; $0 < r < N$, and its powers. Both, NTT and INTT use these values, although INTT also requires the calculation of N^{-1} such that $N^{-1} \cdot N \equiv 1 \ (mod \ q)$, used at the end of INTT multiplying each of the output coefficients by N^{-1}.

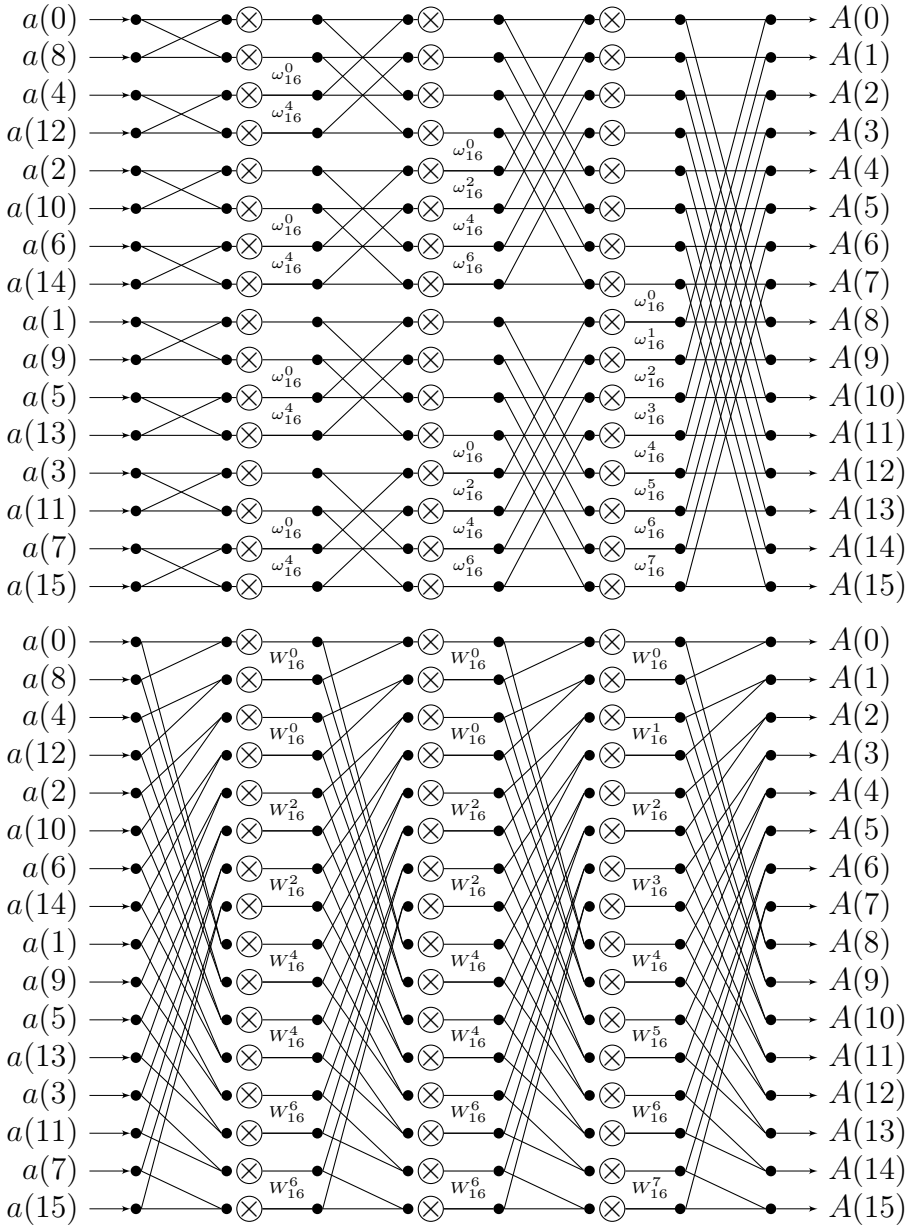


Figure 4.2: Datapath of the Cooley-Tukey(top) and the Pease(bottom) $FFT_{\omega}^{16}(a)$. The last stage of the Cooley-Tukey FFT leaves pairs of indexes together for the starting IFFT (with bit reversal order). The Pease FFT follows the same I/O pattern within all the stages.

4.2 Polynomial Multiplication

When multiplying two polynomials using the NTT, $a(x)$ and $b(x)$, of degree $N-1 = \Phi(m)$, the result is a polynomial $c(x)$ of degree $2n$. However, given the ciphertext space $\mathcal{M} = R_q[Y]$ is bounded by $\Phi_m(X) = \Phi_{2N}(X) = x^N + 1$, the maximum degree of an element $\overline{\mathbf{a}} \in R_q$ is $N-1$. Therefore, it is necessary to reduce such degree of $\overline{\mathbf{a}}$ by operating $\overline{\mathbf{a}}\, mod \Phi_m(X)$, also known as polynomial reduction.

Though the polynomial multiplication using FFT and IFFT can be computed with linearithmic complexity, it firstly needs to double the degree of the input polynomials zeropadding and perform the FFT-IFFT over the superspace of polynomials of degree $2N-2$ doubling, in turn, the number of pointwise multiplications. After that, the polynomial reduction would be performed, obtaining the result polynomial of degree $N-1$.

As we use $x^N + 1$ as generator for the quotient rings R_p and R_q, the polynomial multiplication can be simplified in comparison with equation (4.2) using $x^N \equiv -1$:

$$\sum_{i=0}^{\infty} a_i x^i \;\cdot\; \sum_{i=0}^{\infty} b_i x^i = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (-1)^{\lfloor \frac{i+j}{n} \rfloor} a_i b_j x^{i+j \bmod n} \tag{4.3}$$

To avoid doubling the operations (because of having to double the degree of the polynomials by zeropadding), it is possible to use the *Negative Wrapped Convolution (NWC)*, as illustrated by [18], based on equation 4.3 and implemented in the first and third for loops in algorithm 4. A study of the advantages of using NWC compared to the zeropadding approach (both previously detailed) has been already made by Donglong et al. [11]. When using NWC, it is necessary to find a primitive 2Nth root of unity ϕ. That is, a value ϕ such that: $\phi^{2N} \equiv 1\, mod\, p$, $\phi^r \not\equiv 1\, mod\, p$; $0 < r < 2N$. As shown in algorithm 4, the power of this root multiplies first the coefficients of the input polynomials and, after the INTT, the power of its inverse multiplies the output polynomial, computing the polynomial reduction by $x^N - 1$. For the correctness of the NWC, $\phi^2 \equiv \omega\, mod\, p$ should hold, as well as $p \equiv 1\, mod\, 2N$. Notice that, satisfying

these two last equations, $\omega^N \equiv 1 \, mod \, p$, $\omega^r \not\equiv 1 \, mod \, p$; $0 < r < N$ gets satisfied.

Finally, using the NWC, the polynomial reduction can be replaced by a pointwise multiplication by the powers of ϕ before starting the PM algorithm, and a pointwise multiplication by the powers of ϕ^{-1} at the end (see algorithm 4).

Data: $\mathbf{a}, \mathbf{b}, \omega, \omega^{-1}, \phi, \phi^{-1}, N^{-1}, p, N = 2^n$
Result: $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$
for $i = 0$ ***to*** $N - 1$ **do**
  $\overline{a_i} \leftarrow a_i \phi^i \, mod \, p$;
  $\overline{b_i} \leftarrow b_i \phi^i \, mod \, p$;
end
$\overline{\mathbf{A}} \leftarrow NTT_w^N(\overline{\mathbf{a}})$;
$\overline{\mathbf{B}} \leftarrow NTT_w^N(\overline{\mathbf{b}})$;
for $i = 0$ ***to*** $N - 1$ **do**
  $\overline{C_i} \leftarrow A_i B_i \, mod \, p$;
end
$\overline{\mathbf{c}} \leftarrow INTT_w^N(\overline{\mathbf{C}})$;
for $i = 0$ ***to*** $N - 1$ **do**
  $c_i \leftarrow \overline{c_i} \phi^{-i} \, mod \, p$;
end
return $\mathbf{c}$;

Algorithm 4: Polynomial Multiplication algorithm

4.3 Chinese Remainder Theorem

As detailed during section 4.1, the NTT limits the ciphertext message R_q to use a prime q. However, the CRT, allows q to be a product of different primes $q = \prod p_i$, perform the PM over each of these p_i and then reconstruct the resulted polynomial. This approach enables new levels of parallelism for the FPGA implementation, as well as it enables the possibility of using multiple FPGAs to work in parallel with multiple pipes. Algorithm 5 shows how the previous algorithm 4 can be called several times with small parameters (the call to PM(...) in algorithm 5) instead of once with

big parameters, obtaining the same result. The CRT approach encapsulates the PM with NTT-INTT, using it.

One of the forms of enunciating the CRT is as follows: let q be a product of different coprimes, that is: $q = \prod_{i=0}^{r} p_i\,,\ p_i \neq p_j\ \forall i \neq j,\ i,j \geq 0, r > 0$, then using CRT the quotient ring $R_q = \mathbb{Z}/q\mathbb{Z}[X]/\Phi_m[X] = \mathbb{Z}_q[X]/\Phi_m[X]$ is isomorphic to $\mathbb{Z}_{p_0}[X]/\Phi_m[X] \times ... \times \mathbb{Z}_{p_r}[X]/\Phi_m[X]$. That is:

$$R_q \cong \mathbb{Z}_{p_0}[X]/\Phi_m[X] \times ... \times \mathbb{Z}_{p_r}[X]/\Phi_m[X]$$

Bijection

Therefore, CRT proves that there exists a bijective application $\varphi : \mathbb{Z}_q[X]/\Phi_m[X] \rightarrow \mathbb{Z}_{p_0}[X]/\Phi_m[X] \times ... \times \mathbb{Z}_{p_r}[X]/\Phi_m[X]$. This application is necessary to be able to transform the coefficients of the polynomials to be multiplied to several polynomials with coefficients of smaller size to be multiplied, and to transform the result of those smaller coefficients into a single polynomial. However, such application is yet to be defined. The direct application consists just of the operation:

$$\varphi(\mathbf{x}) = (\mathbf{x}\, mod\, p_0, ..., \mathbf{x}\, mod\, p_r) \tag{4.4}$$

For the inverse application, φ^{-1}, it is necessary to use the *Extended Euclidean Algorithm* to calculate $a_i, b_i, 0 \leq i \leq r$ such that $a_i p_i + b_i(q/p_i) = 1$. Once we have these values, the inverse φ^{-1} is as follows:

$$\varphi^{-1}(\mathbf{x}_0, ..., \mathbf{x}_r) = b_0(q/p_0)\mathbf{x}_0 + ... + b_r(q/p_r)\mathbf{x}_r \tag{4.5}$$

Polynomial Multiplication using Chinese Remainder Theorem

Using the CRT in order to allow q to be coprime, it is possible to enable new levels of parallelism, since the PM over each p_i are independent. This has been referred to as *horizontal parallelism* by related work [27], in contrast with *vertical parallelism*,

which takes place within the same NTT-INTT. Algorithm 5 shows the PM using CRT.

Data: $\mathbf{a}, \mathbf{b}, \omega, \omega^{-1}, \phi, \phi^{-1}, N^{-1}, q = \prod_{i=0}^{r} p_i, N = 2^n, \mathbf{e}$
Result: $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$
for $i = 0$ ***to*** $r - 1$ **do**
  $\overline{\mathbf{a}_i} \leftarrow \mathbf{a} \, mod \, p_i$;
  $\overline{\mathbf{b}_i} \leftarrow \mathbf{b} \, mod \, p_i$;
  $\overline{\omega_i} \leftarrow \omega \, mod \, p_i$;
  $\overline{\omega_i^{-1}} \leftarrow \omega^{-1} \, mod \, p_i$;
  $\overline{\phi_i^{-1}} \leftarrow \phi \, mod \, p_i$;
  $\overline{\phi_i^{-1}} \leftarrow \phi^{-1} \, mod \, p_i$;
  $\overline{N_i^{-1}} \leftarrow N^{-1} \, mod \, p_i$;
end
for $i = 0$ ***to*** $r - 1$ **do**
  $\overline{c_i} \leftarrow PM(\overline{\mathbf{a}_i}, \overline{\mathbf{b}_i}, \overline{\omega_i}, \overline{\phi_i}, \overline{\omega_i^{-1}}, \overline{\phi_i^{-1}}, p_i)$;
end
$\mathbf{c} \leftarrow \sum_{i=0}^{r} \overline{\mathbf{c}_i} * e_i \, mod \, q$;
return $\mathbf{c}$;

Algorithm 5: Polynomial Multiplication using CRT. The parameters e_i represent the coefficients obtained by applying the Extended Euclidean Algorithm (i.e. the b_i values in equation (4.5)). For the rest of parameters see section 4.2.

Apart from the fact that, using CRT, q can be coprime, the possibility of performing the PM over each of the p_i allows the FPGA to use multipliers of small bitsize, which have small latency and take less resources. Also, the new levels of horizontal parallelism enabled by the CRT approach give more versatility when optimizing the algorithm in FPGAs, or even allows to split the multiplication into different FPGAs if the amount of p_i is too big for the PM to fit in one board (bigger security, as q would be bigger as well).

4.4 Modular Reduction

Within any PM algorithm performed over R_q or R_p, the polynomial coefficients are bounded. Therefore, a modular reduction must be performed after each multiplication in order to work with elements in the proper ring at all times. Modular reductions can take a significant amount of time and resources, representing a critical part within the PM. As a result, it is important to choose the proper reduction.

Some previous work implemented algorithms for specific primes, such as *Solinas primes*[26, 31] or *pseudo-Fermat primes*([11, 33]), whereas most of them chose the *Barret modular reduction* ([32, 27, 23]).

Data: a, p
Result: $b = a \; mod \, p$
$b \leftarrow a - \lfloor a/p \rfloor \cdot p$;
if $b < 0$ **then**
  $b \leftarrow b + p$;
end
return b;

Algorithm 6: Straight forward modular reduction method.

Data: a, p
Result: $b = a \; mod \, p$
$count \leftarrow size((a)_2) - size((p)_2)$;
$b \leftarrow |a|$;
$\overline{p} \leftarrow p \ll count$;
for $i = 0$ ***to*** $count - 1$ **do**
  if $b - \overline{p} \geq 0$ **then**
    $b \leftarrow b - \overline{p}$;
  end
  $\overline{p} \leftarrow \overline{p} \gg 1$;
end
if $a<0$ **then**
  $b \leftarrow p - b$;
end
return b;

Algorithm 7: Barret modular reduction.

Figure 4.3: Straight forward (left) and Barret (right) modular reduction methods. Note the algorithm is essentially the same but avoiding the division in Barret's.

Examining algorithm 8, inspired by ([31, 33], one can find that a small value of $p[k-2:0]$, that is, a small value of p minus the most significant bit, reduces the

number of loops. Moreover, the lower is the Hamming weight (that is, the less bits set to 1) of p, the less consuming the computation is.

It is easy to identify that the best prime case for algorithm 8 is a Fermat number. However, the largest known Fermat prime is $2^{16} + 1$ [21]. For this reason, primes p with low Hamming weight and small values for $p[k-2:0]$ are typically referred to as pseudo-Fermat primes.

Data: a, p
Result: $b = a \; mod \, p$
$k \leftarrow bitlength(p)$;
$m \leftarrow bitlength(a)$;
while $m > k$ **do**
  $a \leftarrow a[k-2:0] - a[m-1:k-1]p[k-2:0]$;
  $m \leftarrow m - k + l + 2$;
end
if $a<0$ **then**
  $b \leftarrow a + p$;
else
  $b \leftarrow a$;
end
$return \; b$;

Algorithm 8: Modular reduction for pseudo Fermat primes. Note that for this to work, a should be such that $bitlength(a) \leq 2k$

In section 6.2 one can find a comparison table regarding latency and FPGA resources between Barret, Fermat-like primes reduction and the straight forward approach. Solinas reduction has been discarded from the comparison and from the project itself since it requires a specific type of primes (Solinas primes), instead of allowing modular reduction by any number, and there have been already comparison tables ([31]) between these algorithm with others that prove Solinas approach is definitely a better choice when dealing with Solinas numbers (generalised Mersenne numbers).

4.5 Brakerski-Gentry-Vaikuntanathan Homomorphic Encryption

As mentioned at the beginning of section 4, the main interests is in level 1 ciphertexts (only use addition) in order to maintain the degree one of elements in $R_q[Y]$.

The following random distributions are used by BGV's **Key_Generation, Encryption** and **Decryption** functions, detailed below, in order to ensure cryptographic security:

- $D_{\mathbb{Z}^n,\sigma}$**:** Discrete Gaussian with parameter σ, rounded by the nearest integer vector in $\mathbb{Z}^n$. Referring to [10], we use $\sigma = 3.2$.
- ZO_n**:** randomly sample a vector of size n with $x_i \in \{-1, 0, 1\}$ and $Pr[x_i = -1] = 1/4;\ Pr[x_i = 0] = 1/2;\ Pr[x_i = 1] = 1/4$.

As such, the simplified BGV using using R_p as plaintext space and $R_q[Y]$ as ciphertext space consists of three main functions (notice $\xleftarrow{\$}$ means randomly sample):

- **Key_Generation**$() \rightarrow (pk, dk)$**:** Sample $pk_1 \xleftarrow{\$} R_q$, and $dk, e \xleftarrow{\$} D_{\mathbb{Z}^n,\sigma}$. Compute $pk_2 \in R_q$ as $pk_2 \leftarrow pk_1 \cdot dk + p * e$.
- **Encryption**$_{\mathbf{pk}}(m, r) \rightarrow (c_0, c_1, c_2)$**:** being $(u, v, w) \xleftarrow{\$} (ZO_n, D_{\mathbb{Z}^n,\sigma}, D_{\mathbb{Z}^n,\sigma})$, the output is $c_0 \leftarrow pk_2 \cdot u + p * w + m$ and $c_1 \leftarrow pk_1 \cdot u + p * v$.
- **Decryption**$_{\mathbf{dk}}(c) \rightarrow m$**:** $m \leftarrow (c_0 - dk \cdot c_1 - (dk)^2 \cdot c_2)\ mod\, p$.

Note that $\cdot$ represents the polynomial multiplication of elements in R_q, as explained in sections 4.2 and 4.3, whereas $*$ represents pointwise and scalar-polynomial multiplication. Additionally, $+$ represents the addition over R_q. Using previous functions, one can generate keys, encrypt as much data as wanted, compute operations over it and decrypt the results of the computations using the generated keys.

4.6 Parameter set

Given the big amount of parameters needed for the CRT,NTT,PM and BGV, and its requirements, generating valid parameters to successfully perform BGV can be tricky. Moreover, generating several representative set of parameters can take a big amount of time. As a result, it is worth dedicating some time to designing and algorithm to generate such parameters choosing the plaintext space and some performance and security requirements for the encryption.

The required parameters already mentioned throughout section 4 are calculated by an algorithm which, taking a bitsize for q and a bitsize for each of the p_i such that $q = \prod p_i$, generates each of the p_i, calculates a generator of Z_q and computes ϕ, ω and its inverses.

For the calculation of the generator of $\mathbb{Z}_q$ (hence, of each $\mathbb{Z}_{p_i}$), Shoup's approach [29] is used. Since we do not need any specific p_i, but rather generate them ourselves, it is possible to generate p_i such that the factorization of $p_i - 1$ is known, needed for algorithm 9 to obtain a generator of the group $\mathbb{Z}^{p_i-1}$.

In order to know the factorization of $q-1$, and to satisfy all the conditions needed to ensure the existence of ϕ and ω, already described in section 4.2, each p_i is generated such that $p_i = 2N\overline{p}_i + 1$, being $\overline{p}_i$ an odd number of known factorization (such as a prime).

This way, having a generator gen_i of $\mathbb{Z}_{p_i}$, we have $\phi_i = gen^{\overline{p}_i} \ mod\, p_i$ and, as always, $\omega_i = \phi_i^2 \ mod\, p_i$. Furthermore, using equation (4.5) one can obtain ϕ,ω and q. Finally, the values for φ^{-1} and N^{-1} are obtained computing the Extended Euclidean Algorithm.

Data: $p, p-1 = \prod_{i=0}^{t} q_i^{e_i}$
Result: $gen = find\_gen(p, \prod_{i=0}^{t}) \text{ such that } gen^p \equiv 1 \, mod \, p \text{ and } gen^i \not\equiv 1 \, mod \, p \forall 1 < i < t$
for $i = 0 \; to \; t$ **do**
  $\beta \leftarrow 1$;
  while $\beta \neq 1$ **do**
    $\alpha \xleftarrow{\$} \mathbb{Z}_p$;
    $\beta \leftarrow \alpha^{(p-1)/q_i}$;
  end
  $\gamma_i \leftarrow \alpha^{(p-1)/q_i^{e_i}}$;
end
$gen \leftarrow \prod_{i=0}^{t} \gamma_i$;
$return \; gen$;

Algorithm 9: algorithm for finding a generator of $\mathbb{Z}_p^*$ given a prime p and the factorization of $p-1$.

4.7 Related Work

Previous work have already addressed the implementation of FHE schemes. An example is the proposed by Pöppelmann et al. [26] describe an implementation of the Cooley-Tukey cached-FFT [7, 6] What to be used for the primitive operations of the YASHE [8] HE. However, they do not use the CRT, leading to big multipliers which makes it difficult to achieve good latency with few resources. They mention a promising avenue when using several FPGAs for parallelism, which can be achieved through CRT and proper hardware. They decided to use Solinas primes in order to use its ad hoc modular reduction [31]. For the polynomial reduction, their implementation is based on the negative wrapped convolution [11], adding just N multiplications at the beginning of each FFT/IFFT (being the length of the polynomial).

Roy et al. [27] implement a parallelized Cooley-Tukey FFT for the polynomial multiplication throughout the YASHE encryption scheme , using CRT in order to split the computation. However, that computation does not take advantage of such level of parallelism, but rather is used to work with 32bits words, reducing resources

usage and enabling parallelization within each small prime used for CRT. Although they accomplish to reduce resource usage values, they do not exploit the available space on the FPGA to perform some extra computations. Moreover, they do not report actual times of HE computation, but estimations. Similarly, Erdinç Öztürk et al. [23] use CRT for homomorphic AES evaluation. Even though they take into account the CRT parallelization, their approach forces them to iterate several times over the FPGA which, again, has an important amount of resources unused. Moreover, This iteration requires additional data transmission. They report estimated times for HE computations. Both papers decided to use Barret polynomial reduction along with Barret modular reduction, with a CRT prime bit size that ranges from 30 to 32 bits.

Donglong et al. [11] implement the PM using a constant geometry (CG) FFT. Although this might seem an optimization of the algorithm over FPGAs, this approach generates difficulties when parallelizing the algorithm, mainly because of the bit reversal performed before the IFFT. Also, the set of parameters used are not representative of homomorphic cryptosystems. They also dedicate some time at the parameter set selection, pointing out key observations that might affect critically the final reduction. For the modular reduction, they use the bitwise reduction described in [33]. They also decided to use the same polynomial reduction used by [26].

Most of the related work is implemented using VHDL or Verilog which makes the code more difficult to maintain and thus more prone to bugs. To the best of our knowledge, the most recent published related work using Maxeler tools [15] is from 2010, having this work focused solely on a FFT implementation.

Chapter 5

Implementation

In this chapter, our implementation of the BGV scheme is introduced. First, we show the chosen implementation to generate parameters depending on the requirements for the cryptographic scheme in section 5.1. Then, section 5.2 introduces our BGV implementation, using different versions of the PM implemented for CPUs. Finally, we show the different versions and design decisions made for the PM implemented for FPGAs in 5.3. Figure 5.1 shows different blocks (**PRECOMPUTED PARAMETERS, CONTEXT** and **BGV**) and the functions implemented in all of them as well as the input data they receive and how they communicate with each other. The execution of the project is as follows: first, one generates the precomputed parameters needed for the project, such as the specific plaintext ring and ciphertext ring parameters, roots of unity, length of polynomials, etc.. These parameters are referred to as *context* and are stored in a directory structure in different files. Afterwards, one can import them at the beginning of the execution of the BGV scheme in order to generate the keys. After that, one can encrypt any input data, operate over it and decrypt. All of these functions (key generation, encryption, decryption, operation over encrypted data...) use functions that depend on the specific context imported, such as the PM or the conversion into CRT format.

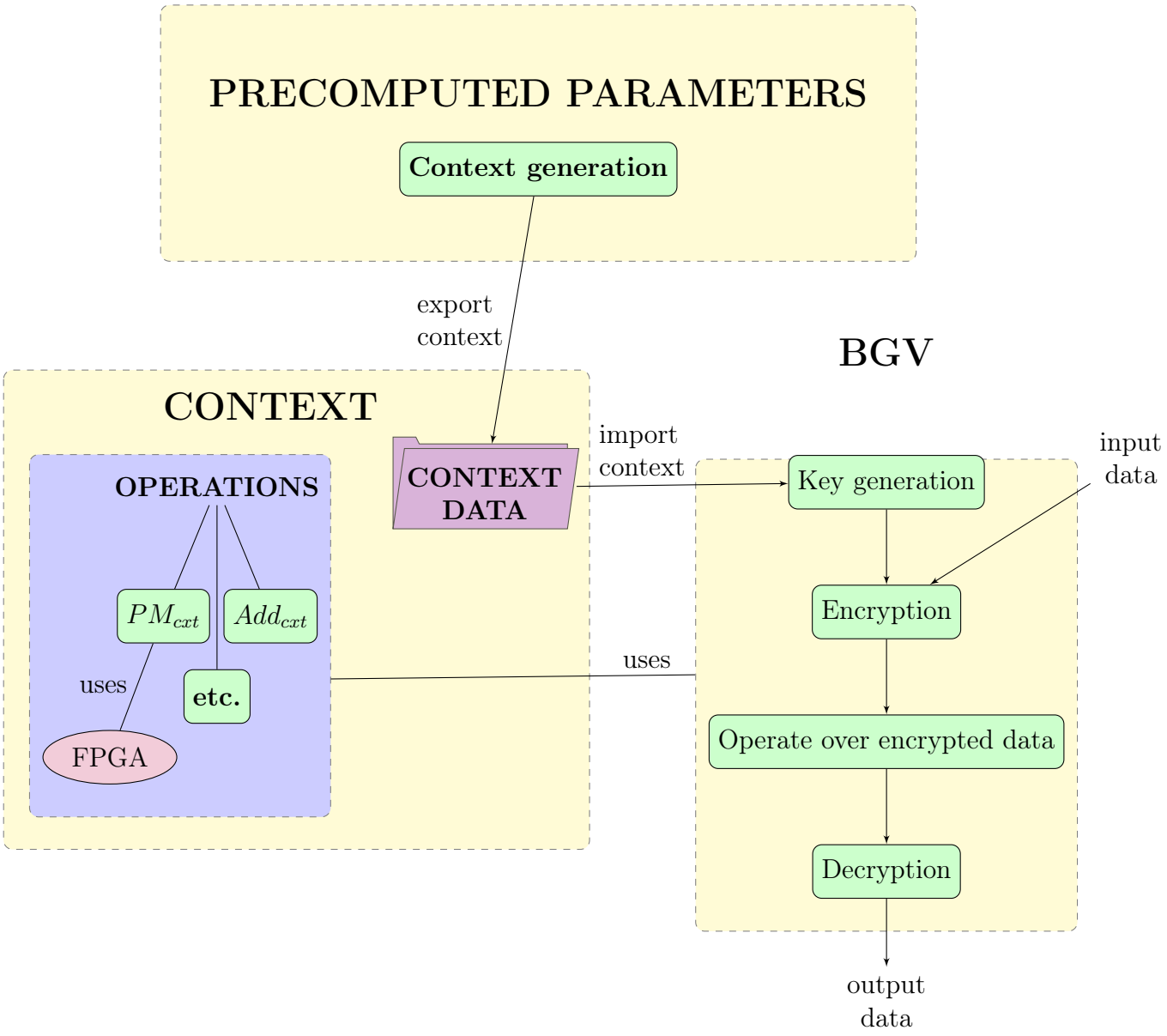


Figure 5.1: Diagram illustrating the different blocks (modules) implemented and how they are related.

5.1 Parameter set Selection

As detailed throughout section 4, some parameters used for the Polynomial Multiplication using NTT-INTT are automatically calculated in order to be able to use different sets. This is implemented by the block **PRECOMPUTER PARAMETERS** (see figure 5.1).

The block **PRECOMPUTED PARAMETERS**, detailed in figure 5.2, receives as input a minimum bitsize required for the value q (**final_bitsize**), a minimum and maximum bitsize required for each of the primes p_i such that $q = \prod p_i$ (**pi_min_bitsize** and **pi_max_bitsize**) and the length of the polynomials (that is, $\Phi(m)$ following the notation of the beginning of section 4) ($N = 2^{\mathbf{n}}$), and generates:

- the number q used for the ciphertext R_q, as well as the primes p_i such that $q = \prod p_i$ (call to **find_p_gen** in figure 5.1).
- $\overline{p}_i$ such that $p_i = 2N\overline{p}_i + 1$, following the generation algorithm explained at section 4.6 (calling **find_p_gen**).
- the bit representation and bitsize of p_i and $p_i - 1$, needed for algorithm 8.
- a generator gen_i of the field Z_{p_i} (calling **find_p_gen**).
- the powers of ω, ω^{-1}, ϕ and ϕ^{-1}, as described in sections 4.1 and 4.2 (calling **root_values** and **powers**).
- the bit-reversed indexes as needed for the $Bit\_reverse(\mathbf{a})$ used in the NTT and INTT (see figure 4.1 and 4.2).
- the value of N^{-1} in R_q, first mentioned in section 4.1 (calling **find_inverse**).
- the parameters b_i needed for the bijection $\mathbb{Z}_q[X]/\Phi_m[X] = \mathbb{Z}_{p_0}[X]/\Phi_m[X] \times ... \times \mathbb{Z}_{p_r}[X]/\Phi_m[X]$, as detailed in section 4.3 (referred to as **euclidean_coeffs** when calling **extended_euclidean_algorithm**).

These parameters are referred to as the *context* of the ciphertext.

The project's CPU code uses the mpz_t C types from the library GMP [16] in order to work with numbers bigger than one machine word. Finally, the algorithm performs some checks to verify that the exported data fulfills the so-called requirements. The execution is illustrated in figure 5.2.

This project uses different subblocks: **extended_euclidean_algorithm** (which implements the algorithm detailed in section 4.3), **bijection_CRT_Q** (with a set of functions to convert data between isomorphic rings, such as the bijective application from section 4.3) and **gmp_extra_functions** (with useful auxiliary functions, such as printing multidimensional mpz arrays or initializing them).

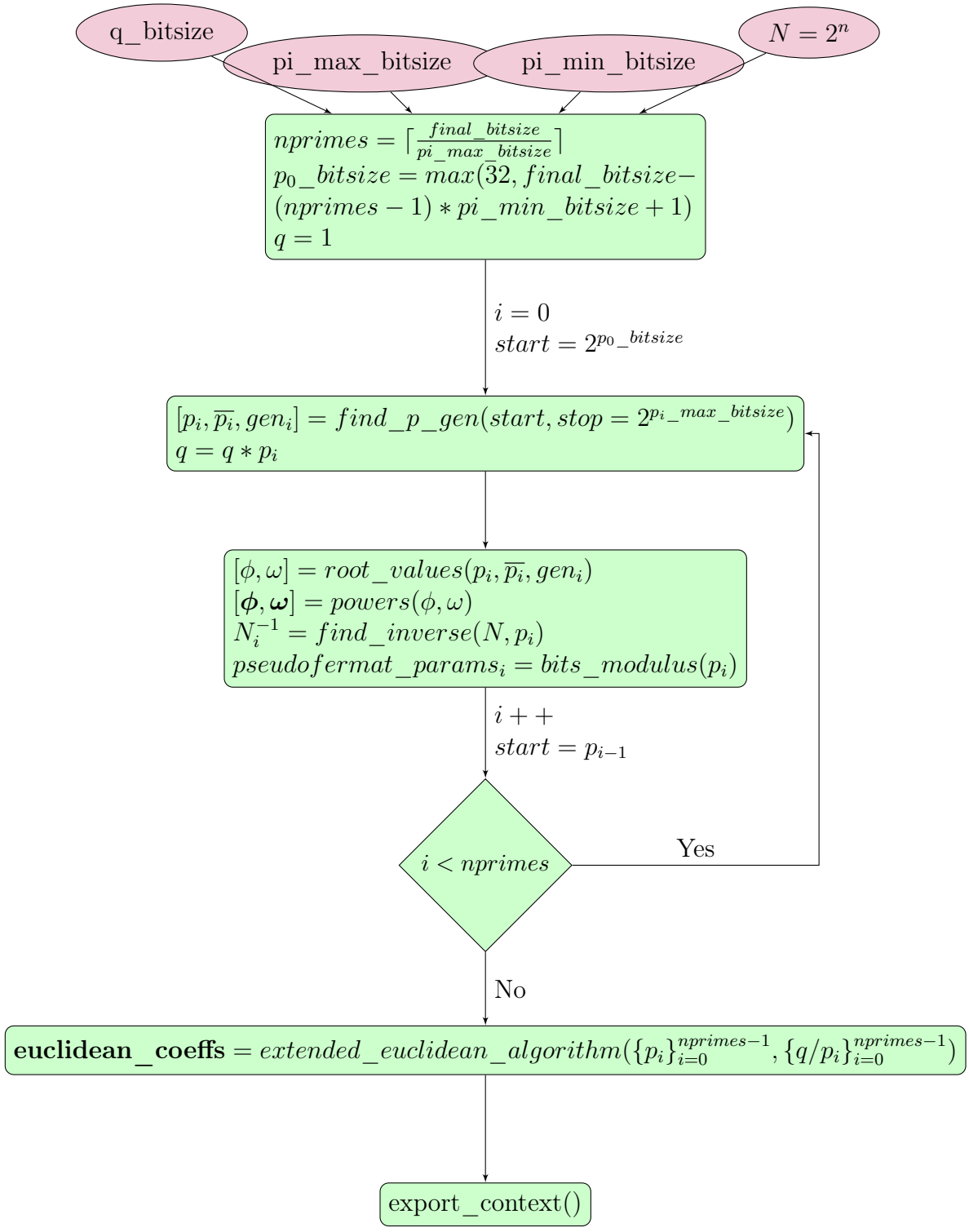


Figure 5.2: Diagram illustrating the execution of **PRECOMPUTED PARAMETERS**. Notice that $\overline{p}_i$ follows the same notation as in section 4.6. **export_params** writes the parameters inside the "data" subdirectory.

5.2 BGV & Polynomial Multiplication (CPU)

The block **BGV** (see figure 5.1 hosts the main execution of the project (key generation, encryption, operate over encrypted data and decryption). It uses the blocks **CONTEXT**, **bijection_CRT_Q**, **gmp_extra_functions** and the library **libherandom.a**. As one can observe from the documentation of GMP, "raw output" of their random functions "is unsuitable for cryptographic applications without further hashing or the like". For this reason, the randomness implemented uses the functions and types from *Number Theory Library* (NTL) [30], which implements "high quality, cryptographically strong pseudo-random numbers". However, given that NTL is a C++ library, the library **libherandom.a** was implemented encapsulating only the needed methods from NTL into a C library. These methods include a bijective conversion between NTL *ZZ* and GMP *mpz_t* types, as well as a wrapper for returning int64_t types.

The program execution is illustrated in figure 5.3. First, one should import a context in the format exported by **PRECOMPUTED PARAMETERS**. Then, one can start the keygeneration, generating a decipher key **dk** and a public key $pk \in R_q[Y],\ pk = pk_0 + pk_1 y$. Afterwards, the encryption of the input messages **a** and **b** takes place, generating $\overline{\mathbf{a}} = \overline{\mathbf{a}}_0 + \overline{\mathbf{a}}_1 Y$ and $\overline{\mathbf{b}} = \overline{\mathbf{b}}_0 + \overline{\mathbf{b}}_1 Y$. Then, one can operate with these two messages (in this case we add $a+b$ as an example), generating encrypted output ($\overline{\mathbf{c}}$). Finally, the generated output is decrypted, producing plaintext output **c**.

NTL has several methods to generate cryptographically strong random numbers within an uniform distribution. Nevertheless, as detailed throughout section 4.5, a Discrete Gaussian Distribution is also needed for the BGV encryption scheme. As a result, we implement the *Box-Muller method* (BM) [20] to generate a Discrete Gaussian Distribution out of two Uniform Distributions, proposed by George Edward Pelham Box and Mervin Edgar Muller in 1958, BM algorithm is a transformation of two Uniformly distributed random numbers to generate a pair of independent

Gaussian distributed random numbers.

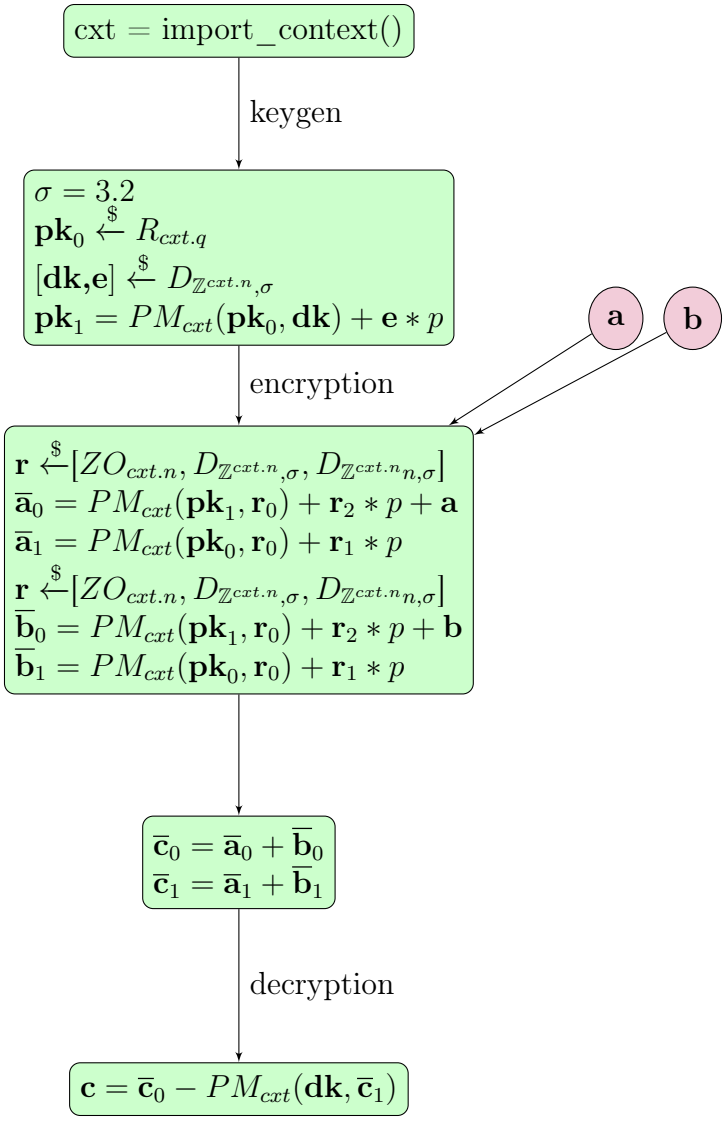


Figure 5.3: Diagram illustrating the execution of **BGV**.

5.3 Polynomial Multiplication (FPGA)

the FPGA base implementation of the PM consists basically of two inputs (each of the polynomials to be multiplied), and two outputs (one representing the first half $x^0, ..., x^{N/2-1}$ of the output polynomial and the second one the other half $x^{N/2}, ..., x^N$), instead of two inputs and one output as shown in algorithm 4. The reason for this is that the output polynomial has to be split between first and second half since, as shown in algorithm 4, two outputs in equivalent indexes relative to the first and second half are generated at the same time and, as explained in section 3 FPGAs do not allow random access to positions in output streams. Three counters help controlling the execution of the FPGA, given the pipelined architecture of the FPGA (detailed throughout section 3). One of these counters, i, keeps track of the number of ticks since the execution started. The second and third counter k and j, respectively, are chained counters in such a way that j increments in each tick from 0 to $N/2-1$, when it wraps again to 0 and k increments its value by 1. That is, k represents each "stage" (block of N/2 ticks), and j represents each tick relative to the stage, the same way two nested for loops work in CPU programming. This nested loops are the ones already mentioned in section 4.1 (i for stages and j for ticks within stage in algorithm 2).

Given the programming paradigm of FPGAs, random access to different indexes of an array is simply not possible for streams (such as the input and output polynomials). As such, the initial bitreversal operation of the NTT has to be performed explicitly reordering the indexes. It can be either performed in CPU or at the beginning of the FPGA, reading in natural order the input and storing in the proper position in RAMs (by adding another stage at the beginning of the FPGA execution, delaying the execution of the actual NTT stages). Additionally, as mentioned in section 3, BRAMs can not have more than two ports, and both ports can not point at the same position in memory, which forces the usage of a Ping-Pong approach as in [11]. This approach consists simply of having two RAMs for the same data, but depending on the tick, one of the RAMs is for reading and the other for writing data,

or the other way around, illustrated in figure 5.6.

One can notice that the stages are of N/2 ticks whereas the input polynomials are of degree N-1. This is because the butterfly operation (operations in the innermost loop of both algorithms in figure 4.1) uses two coefficients of the polynomial. Again, regardless of the specific chosen NTT (Pease's or Cooley-Tukey's), this restriction along with the access patterns for both algorithms are incompatible with having just two Ping-Pong RAMs per polynomial, as it would imply having more than four ports per RAM, two writing ports or two reading ports. Thus, at least 4 RAMs are needed: 2 'Ping' and 2 'Pong' equivalents. For Pease's algorithm, a feasible approach is to store even indexes (coefficients) (such as $x^0, x^2, x^4, ...$) in one of the RAMs; whereas odd indexes in the other one. Notice that when speaking of odd and even indexes, it is relative to the stage, regardless of the absolute coefficient that the index represents. That is, after the bitreversal, the coefficient of $x^{N/2}$ is in the index 1 and for such reason it is an odd coefficient within the first stage (it would be the first position of the RAM for odd coefficients).

Furthermore, the butterfly operation returns indexes of the same parity (both odd or even numbers) and therefore they cannot be written during the same tick as they are generated, given the two-port limitation in BRAMs (one is for reading). As a solution, it is possible to access to the data in a stream (a variable along ticks in FPGA) with a tick *offset* (relative to the current tick). A tick offset allows accessing future/previous values of a stream in different ticks relative to current tick. Figure 3.1 shows both types of offsets (negative and positive), represented in the shape of a diamond. Offsets should not be used for random access to array positions, since they slow down execution (if the offset is positive the entire execution path needs to increase latency in order to wait for the specific value to arrive) and consume LUTs (if the offset is negative it needs to store previous values for when useful). In this case, however, an offset of +1 (that is, accessing the value of the same variable in the following tick) for even ticks and of -1 (that is, accessing the value of the same variable in the previous tick) solves the parity problem, since the parity of both

output values of butterflies changes within consecutive ticks (see figure 5.4).

Moreover, using 4 RAMs with Pease's gives a reduced latency in the most critical part of the algorithm: linking the output patterns of both FFTs with the bitreversal at the beginning of the IFFT (see algorithm 4). As illustrated by figure 5.5, when reading in bitreversal during the last FFT stage, the output is properly paired (when there is no parallelism) for the IFFT which allows an execution path with no extra latency derived from waiting for data to be ready; other than the ticks needed to ensure that the bitreversal indexes of the data to be read has already been stored. This approach is used when reading in bit reversal order in Figure 5.6.

It is worth noticing that there is a number of ticks at the beginning of the algorithm itself which do not write into RAM just because the data is not ready to be written. This is referred to as the initial latency. For this to be solved it is necessary to work with an offset over the ticks since the beginning, since the compiler does not calculate the offset automatically. This offset negative value varies significantly depending of the bitsize of the specific project, as well as the modulus implemented, because the latency of the pipe changes significantly according to these parameters (see section 6.2). Also, as both FFT of the input polynomials are completely independent, they can be performed simultaneously in parallel pipes that converge into one at the end of both FFTs, to perform the element-wise multiplication for the finishing IFFT.

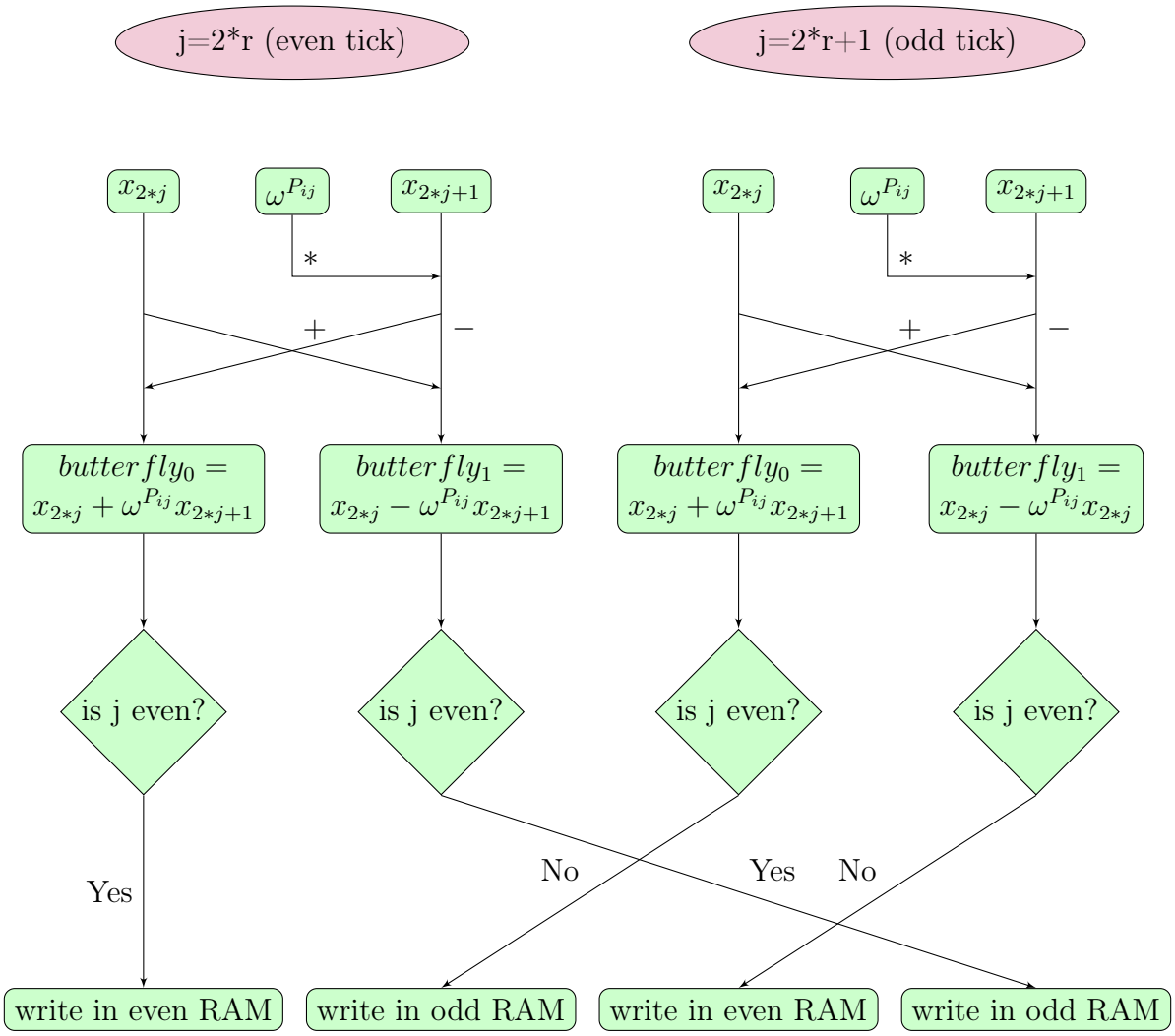


Figure 5.4: Butterfly FPGA execution graph. Note the different circuit depending on the parity of the tick in order to write in even and odd RAMs in each tick.

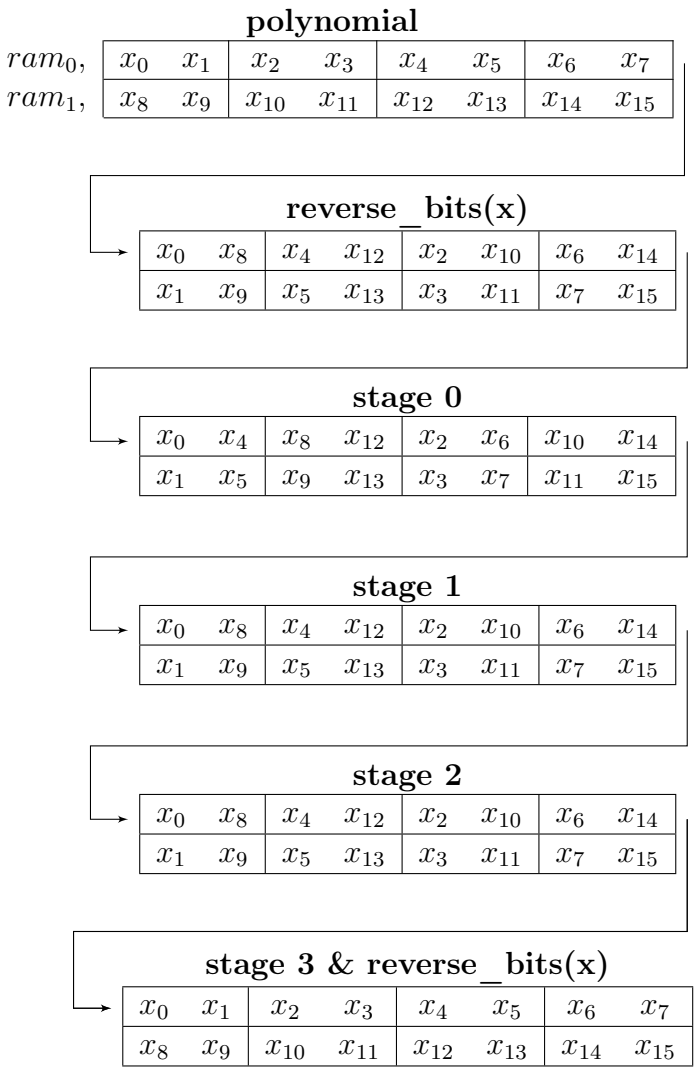


Figure 5.5: Pease's FFT and bit reversal (for IFFT) I/O patterns for case N=16. Note that the pair of coefficients (coefficients in the same cell) remain intact between **stage 2** and **stage 3 & reverse_bits(x)**. Only the position of the pairs is changed. Pairs of coefficient represent coefficients that are read at the same time to perform the butterfly.

Optimizations

Regarding data transfer between CPU and FPGA, the precomputed parameters can be directly stored in the FPGA or transferred from CPU via PCIe bus at the beginning of each execution. Transferring from CPU increments the overhead time spent in transferring data, whereas directly storing into the FPGA causes the FPGA to reload when the context changes. Therefore, the proper approach should involve transferring the context from CPU, via PCIe bus trying to optimize the data. A quick look at the precomputed parameters evidences some optimizations. Given that $\phi^2 = \omega \ mod\, q$ and $\phi^i = (\phi^{-1})^{N-i} \ mod\, p$ (ϕ is a primitive Nth root of unity), we can simply transfer $\phi^i, i = 0, ..., N-1$. This optimizes the transfer time and the BRAMs used in FPGA, but it uses extra LUTs. The resource usage feedback by MaxCompiler shown in section 6 helps tell the proper option depending on the context.

Whether the context is finally transferred from CPU or stored in ROMs (read-only BRAMs), there are at least two streams of data from CPU (the two vectors) and another two to CPU (the output vector). In this case, MaxCompiler gives the possibility of using the already mentioned LMem as part of their DFE engine. Using LMem one can define an asynchronous write to the FPGA while performing different tasks, invoke the FPGA for execution (after having waited for the write asynchronous task to finish), and then read from FPGA when needed. Another option is to use CPUStreams, which are transferred from and to CPU (to and from FPGA, respectively) within the same invocation to the actual Kernel. Again, there is no clear winner for all cases, since using LMem implies there has to be three calls to the DFE with different SLiC, which has an overhead compared to different times. Besides, the actual invocation of the Kernel can be implemented to be asynchronous too. Furthermore, LMem requires data to be accessed burst aligned, being burst a fixed number of bits that vary depending on the specific DFE (192 bits for our targeted Board), which causes the data to be zero-padded and the read/write operations to take more time than needed.

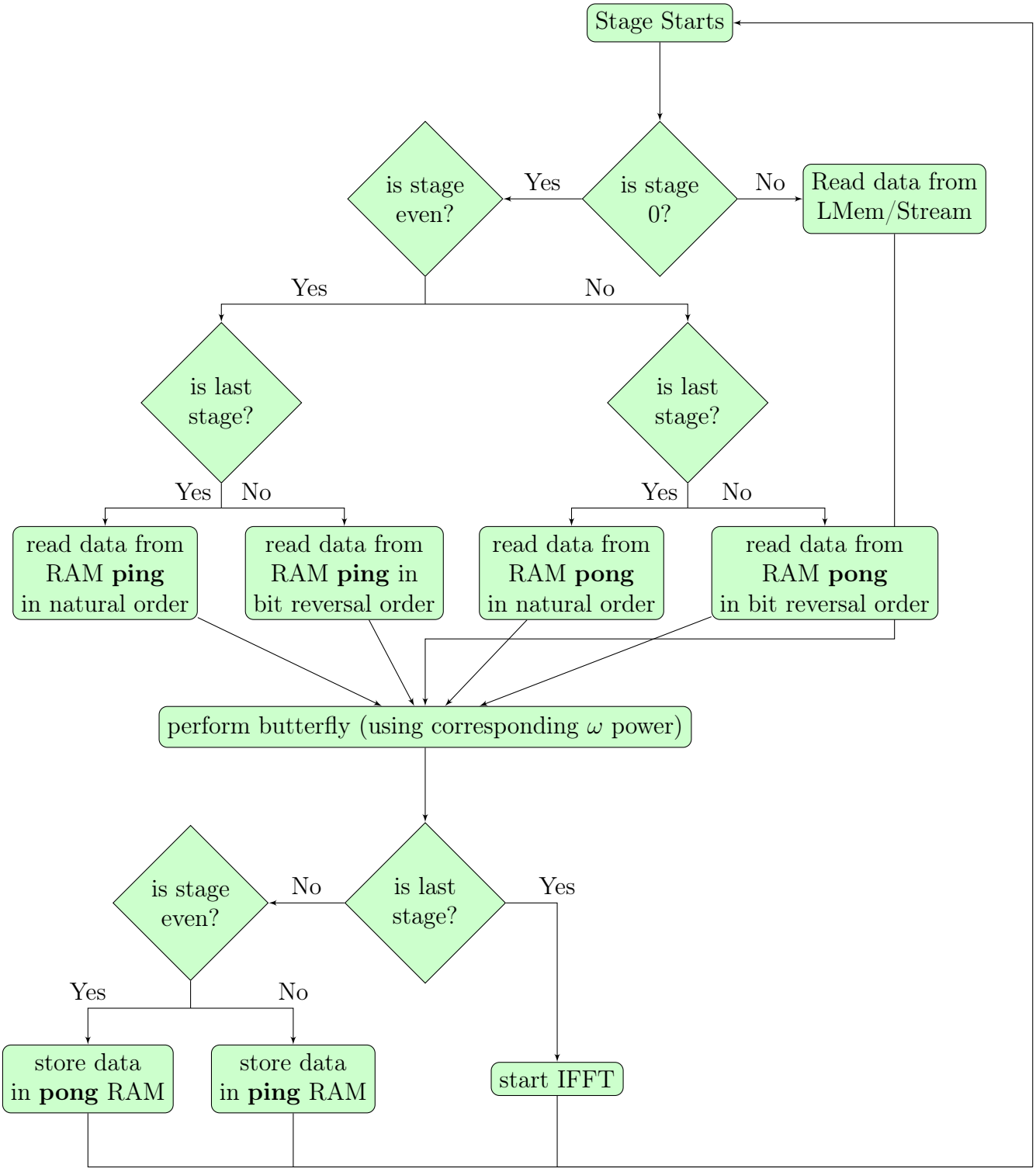


Figure 5.6: FPGA simplified Kernel graph of the FFT.

Other important optimizations lie in the use of the same resources for the IFFT and for (one of) the FFT. Though the butterfly operations, RAMs, modulus operations,... take a significant amount of resources, they can be reused for the IFFT by using more logic cells that distinguish between FFT stages and IFFT stages. This way, one can use the same DSPs and BRAMs of one of the FFTs for the following IFFT by increasing the logic utilization resulting from selecting the FFT or IFFT read/write pattern and ω or ω^{-1} depending on which one (FFT or IFFT) is being performed in the specific tick. Therefore, We perform two parallel FFTs and one IFFT without almost any extra resources compared with two parallel FFTs.

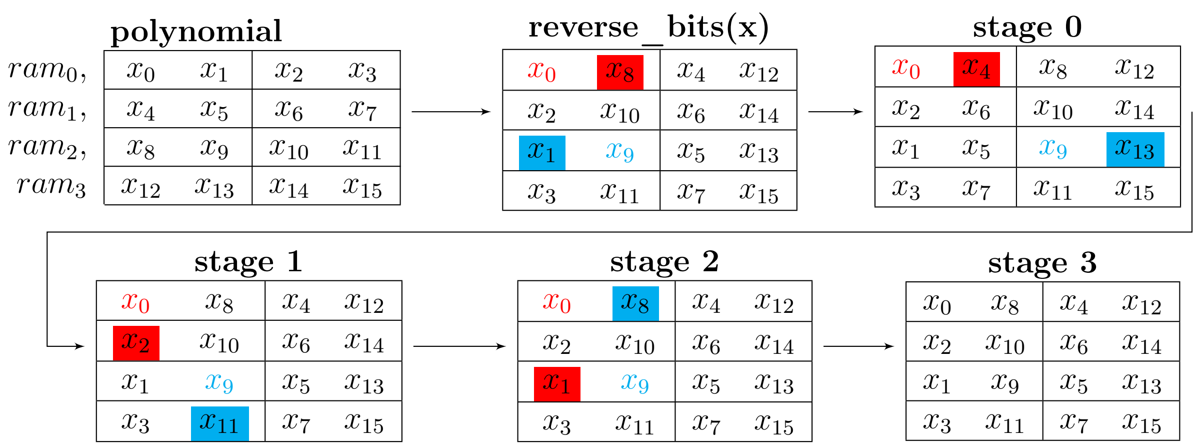


Figure 5.7: Cooley-Tukey's RAMs I/O patters for case N=16, numpipes=4

FPGA Parallelism

Apart from the already mentioned parallelization of both FFTs of the input polynomials, there are three main levels of parallelization when using CRT in order to divide the PM over R_q into many PMs (PM_{p_i}) over rings of smaller size, as explained in section 4.3: using multiple FPGAs for different PM_{p_i} (horizontal parallelism), using different PM_{p_i} (horizontal parallelism) in the same FPGA and using multiple pipes within the same PM_{p_i} (vertical parallelism). The three of them can be combined limited only by the amount of resources available in the FPGAs. Figure 5.8

shows the two horizontal parallelism cases compared to no parallelism for a case with 3 p_i. Notice that any of the three parallelism requires a high amount of resources or even several FPGAs. The parallelism that takes the least amount of resources would be the vertical one, since there is no need to increase BRAM usage at all, given that the precomputed parameters remains the same for all the pipes. Regarding the vertical parallelism, it should be noticed that the multiplicity of pipes only works for numbers of pipes which are power of two. Because of this latter level of parallelization, Cooley-Tukey's approach comes up easier for FPGAs, given that, despite of not having a uniform geometry (constant I/O pattern), the inputs and outputs have the same patterns in their positions. The other two levels of parallelism come for free since they have no dependency whatsoever.

As for the parallelization within the same FFT_{p_i}, the main issue is to design a suitable, resource-efficient I/O pattern for storing the coefficients into RAMs. It is obvious that to have **numpipes**=4 (following example shown in figure 5.7) parallel pipes it is necessary to read 8 different coefficients to perform 4 butterflies. Therefore, it is necessary to distribute the coefficients into 4 (8 Ping-Pong) RAMs. Moreover, this RAMs store different coefficients in different stages, as the I/O patterns vary. Finally, it is also necessary to be able to store the coefficients again in a feasible way for the next stage to be performed.

Figure 5.7 shows a possible solution, consisting of splitting the polynomial into **numpipes**, having the ith partition the coefficients $x^{i*N/numpipes}, ..., x^{(i+1)*N/numpipes-1}$, showing the approach for the case numpipes=4, N=16. With this approach, represented as a matrix in figure 5.7, all the necessary coefficients are available at each stage. Notice that the coefficient in red background represents the coefficient to read in order to operate the butterfly with the coefficient x_0, in red font, while the coefficient in blue represents the same for coefficient x_9, in blue font. At each stage, the coefficient in red/blue to be read is either in the same pair of the row as the corresponding red/blue font coefficient, or in the same column. Given that each of the rows represent a different RAM of a vector of size 2 (that is, always read and write

two consecutive coefficients), it can be ensure that all the dependent coefficients are to be read and written during the same ticks. Notice that figure 5.7 shows how the coefficients change the position within the row, in order to maintain the read (and not the write) pattern somewhat constant within the same RAM. Note also that in each tick different RAMs allow reading the same position (two values) in each row in figure 5.7.

However, one of the drawbacks of using Cooley-Tukey is related to the bitreversal operation at the beginning of the IFFT, which can not be perform simultaneously (that is, during the same read tick) with the last stage of the FFT. Nevertheless, the amount of ticks that were needed to wait for the data to be ready where big enough (around N ticks) to reduce the importance of such drawback, as in the end the amount of ticks needed is roughly the same. Again, the first arrow of figure 5.7 shows another stage dedicated to perform such bitreversal, when is possible to simultaneously compute the first stage of the IFFT.

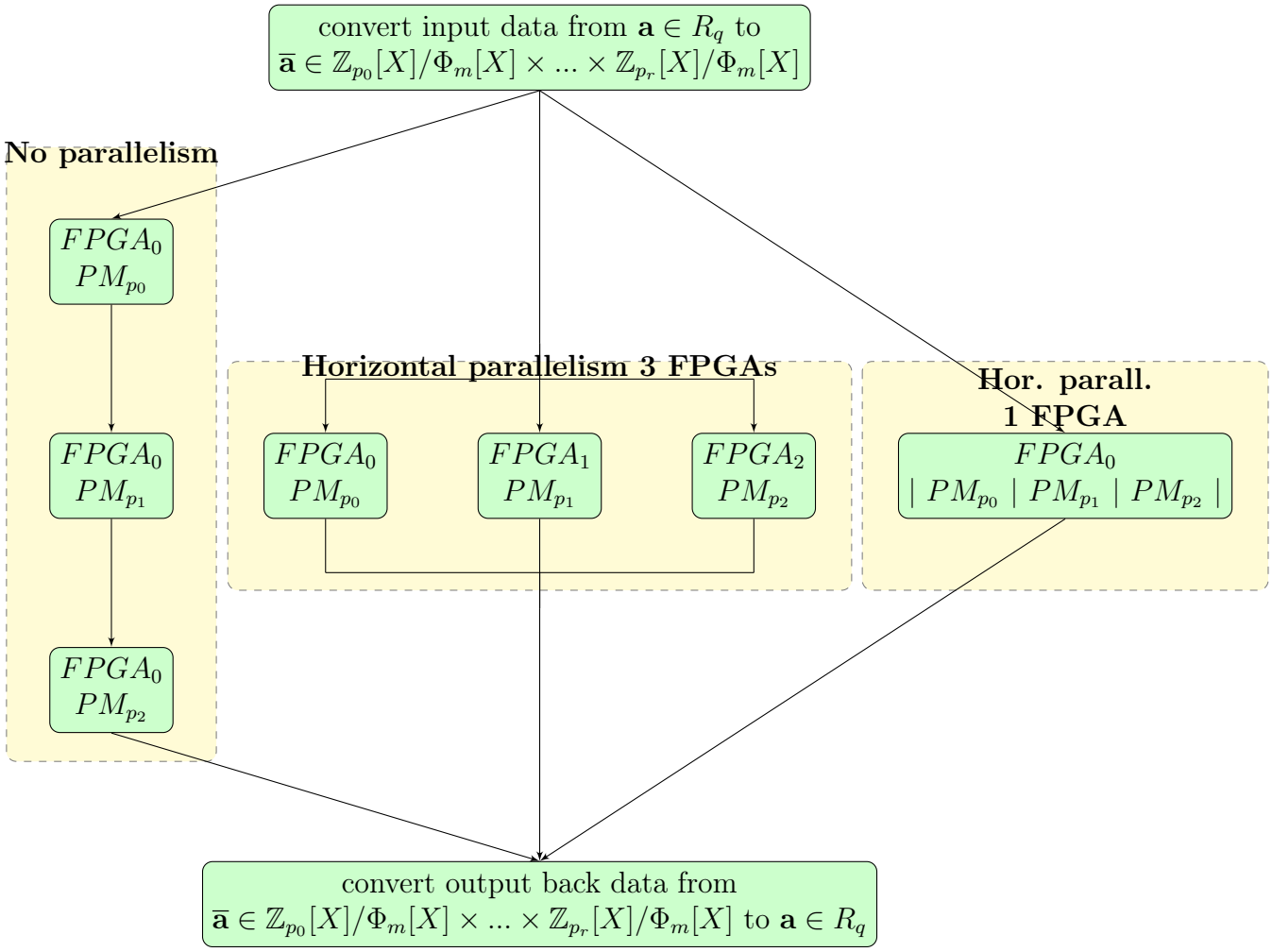


Figure 5.8: Different types of horizontal parallelism. All of them can be combined to maximize the available resources.

Chapter 6

Evaluation

Section 6.2 analyses the resources usage and latency of each of the modular reductions for different configurations. Different CPU versions of the PM seen in chapter 4 are compared in section 6.3. Some of those CPU versions are compared with different FPGA versions of the PM, along with some times regarding the whole BGV encryption scheme in sections 6.4 and 6.5.

6.1 Targeted Board

As the target FPGA board, we use Altera Stratix V 5SGSD5, within the MaxWorkstation kit. It comes with a Intel Core i7-3770S CPU @ 3.10GHz, 16GB RAM CPU, which is the CPU used for the CPU times in chapter 6. DFEs connect to CPU via PCI Express gen2 x8.

6.2 Modular Reductions

As mentioned in section 4.4, there are three main possible modular reductions, illustrated by algorithms 6, 7 and 8. As explained in section 4.4, algorithm 8 depends greatly on the chosen 'prime' p used for the modular reduction. Specifically, the Hamming weight and the value of p minus its most significant bit (with respect to

the value of p) increases significantly the computation of the algorithm. For this reason, there are several tables evaluating resources and latency depending of the type of prime used: pseudo-Mersenne primes (primes with big Hamming weight and value of p minus its most significant bit), pseudo-Fermat primes (primes with small Hamming weight and value of p minus its most significant bit), and standard primes (primes that have neither big nor small Hamming weight and value of p minus its significant bit). The following tables study the resources usage and latency (number of clocks) needed per modular reduction. It should be noted that in order to perform a successful modular reduction within the FPGA, it is necessary (at least) to input two values (from two input streams), multiply them and generate an output. These additional operations add latency and take resources to complete. As a result, all of the shown resources are with respect to a basic resource usage and latency shown in table 6.1. In other words, all the tables in this section have the amounts of resources shown in table 6.1 (first or second row depending of the bits) plus the amount of resources of each modular reduction.

Configuration	LUT	FFs	BRAMs	DSPs	Latency
64bit	60	410	0	8	5
128bit	506	1934	0	15	7

Table 6.1: Resource usage for simple 64bit and 128bit multiplication of two values in FPGA.

Table 6.2 shows the latency and the amount and percentage of LUTs,FFs,BRAMs and DSPs needed when operating modulo over two different pseudo-Mersenne 'primes' for algorithms 6, 7 and 8, named Modulus, Barret and Fermat-like in table 6.2, respectively. The chosen 'primes' are $p = 0x439f0001$ (I), which has a high Hamming weight and value of p minus its most significant bit close to p (0x039f0001), and p=0x439f000000000001 (II), with the same characteristic as the previous value but extended to operate with numbers of 128 bits.

As can be seen in table 6.2, when operating with pseudo-Mersenne numbers the Fermat-like reduction uses a great amount of DSPs compared to modulus and Barret

Reduction	LUT	FFs	BRAMs	DSPs	latency
Barret(I)	3176(0.92%)	4729(0.68%)	0	8(0.5%)	108
Fermat-like(I)	15956(4.62%)	41363(5.99%)	0	345(21.7%)	119
Modulus(I)	5701(1.65%)	6226(0.90%)	50(2.48%)	16(1.01%)	83
Barret(II)	11570(3.35%)	18843(2.73%)	0	15(0.94%)	206
Fermat-like(II)	31324(9.07%)	80905(11.72%)	0	645(40.57%)	219
Modulus(II)	20534(5.95%)	21647(3.14%)	114(5.66%)	15(0.94%)	146

Table 6.2: Resource usage and latency for each of the reductions using pseudo-Mersenne numbers. (I) is for 64bits words using p=0x439f0001, whereas (II) for 128bits words using p=0x439f000000000001.

reductions and therefore it should not be the chosen scheme. The straight forward approach (Modulus in table 6.2), however, has the lowest latency, not using much more resources than Barret's, apart from BRAMs.

Table 6.3, shows the same resources and latency as table 6.2 but for pseudo-Fermat primes. As such, 0x40000003 (I) represents the first three rows of the table, whereas 0x4000000000000031(II) the second three rows, both with low Hamming weight and low values without the most significant bit.

Reduction	LUT	FFs	BRAMs	DSPs	Latency
Barret(I)	3176(0.92%)	4729(0.68%)	0	8(0.5%)	108
Fermat-like(I)	946(0.27%)	2327(0.34%)	0	15(0.94%)	17
Modulus(I)	5701(1.65%)	6226(0.90%)	50(2.48%)	16(1.01%)	83
Barret(II)	11570(3.35%)	18843(2.73%)	0	15(0.94%)	206
Fermat-like(II)	1426(0.41%)	2815(0.41%)	0	15(0.94%)	17
Modulus(II)	20534(5.95%)	21647(3.14%)	114(5.66%)	15(0.94%)	146

Table 6.3: Resources and latency for each of the reductions using pseudo-Fermat numbers. (I) is for 64bits words using p=0x40000003. (II) for 128bits words using p=0x4000000000000031.

Table 6.3 illustrates a significant optimization for the Fermat-like approach, specially in the latency. For this reason, there should be no doubt that the Fermat-like prime modular reduction is the best option when using pseudo-Fermat primes.

Finally, table 6.4 follows the same structure as the two previous tables but using 'standard primes', that is, primes that are as close to being pseudo-Fermat as they are of being pseudo-Mersenne. In this case, the chosen values are 0x40000000039f0001(III),0x40000000809f0 and 0x40008901(I).

Reduction	LUT	FFs	BRAMs	DSPs	Latency
Barret(I)	3176(0.92%)	4729(0.68%)	0	8(0.5%)	108
Fermat-like(I)	5000(1.45%)	13016(1.89%)	0	105(6.60%)	39
Modulus(I)	5701(1.65%)	6226(0.90%)	50(2.48%)	16(1.01%)	83
Barret(II)	11570(3.35%)	18843(2.73%)	0	15(0.94%)	206
Fermat-like(II)	5192(1.5%)	13461(1.95%)	0	105(6.6%)	39
Modulus(II)	20534(5.95%)	21647(3.14%)	114(5.66%)	15(0.94%)	146
Barret(III)	11570(3.35%)	18843(2.73%)	0	15(0.94%)	206
Fermat-like(III)	3719(1.08%)	9800(1.42%)	0	75(4.72%)	29
Modulus(III)	20534(5.95%)	21647(3.14%)	114(5.66%)	15(0.94%)	146

Table 6.4: Resources and latency for each of the reductions using numbers that are neither Fermat-like nor Mersenne-like. (I) uses p=0x40008901, whereas (II) p=0x40000000809f0001 and (III) p=0x40000000039f0001.

Again, Fermat-like modular reduction requires a much lower latency, without a significant difference in the resource usage, since Fermat-like uses around 5% more DSPs but around 5% less of the rest altogether for 128bit. Consequently, it is clearly a better approach using the Fermat-like algorithm. The reason for having case (II) and (III) in table 6.4 is that all the primes we generate and use (see sections 4.6 and 5.1) have a Hamming weight similar to cases (II) and (III), and its second most significant bit is between the 26th (case (III)) and 32nd (case (II)) bits.

6.3 Polynomial Multiplication (CPU)

The best configuration of the PM for the FPGA might not be the best one for CPU. For this reason, several CPU versions are considered. This way, one can compare equivalent versions in CPU and FPGA as well as the best case for each of them.

Table 6.5 shows different times of different versions for the PM for a polynomial length $N = 2^n = 16384$. Note that, for each bit length of the final coprime q, different approaches are possible. PM_{CRT} shows times for an approach in which, because of using p_i of 32 bits, GMP is not needed, but the PM must be invoked $\lceil bits(q)/32 \rceil$ times. PM_{GMP}, on the other hand, uses only GMP, without any CRT at all. In this case, the Polynomial Multiplication is performed once but with big integers (mpz_t type of GMP). Finally, $\text{PM}_{\text{CRT,GMP}}$ uses a combination of the two previous, with different cases depending on the bit length desired for q and for each of the p_i. It should be note that the values the columns bitsize(q),num_{p_i}, and bitsize(p_i) are directly related, because using CRT means $bitsize(q) \leq num_{p_i} * bitsize(p_i)$. This means that modifying one of them in table 6.5 changes the value of the other two.

One can see that, as expected, since using GMP means not being able to perform multiplications using primitive types, the best option is to avoid GMP at all times. Moreover, increasing the number of p_i in case $PM_{CRT,GMP}$ increases computation time in all cases.

case	bitsize(q)	num_{p_i}	bitsize(p_i)	CRT	GMP	Microseconds
PM_{CRT}	64	2	32	YES	NO	25227
PM_{GMP}	64	1	64	NO	YES	69875
$\text{PM}_{\text{CRT,GMP}}$	64	2	32	YES	YES	141238
PM_{CRT}	128	4	32	YES	NO	49472
PM_{GMP}	128	1	128	NO	YES	86577
$\text{PM}_{\text{CRT,GMP}}$	128	4	32	YES	YES	281200
$\text{PM}_{\text{CRT,GMP}}$	128	2	64	YES	YES	147866
PM_{CRT}	256	8	32	YES	NO	100787
PM_{GMP}	256	1	256	NO	YES	129443
$\text{PM}_{\text{CRT,GMP}}$	256	8	32	YES	YES	570764
$\text{PM}_{\text{CRT,GMP}}$	256	4	64	YES	YES	299480
$\text{PM}_{\text{CRT,GMP}}$	256	2	128	YES	YES	185223
PM_{CRT}	512	16	32	YES	NO	204035
PM_{GMP}	512	1	512	NO	YES	230858
$\text{PM}_{\text{CRT,GMP}}$	512	16	32	YES	YES	1124334
$\text{PM}_{\text{CRT,GMP}}$	512	8	64	YES	YES	600584
$\text{PM}_{\text{CRT,GMP}}$	512	4	128	YES	YES	370839
$\text{PM}_{\text{CRT,GMP}}$	512	2	256	YES	YES	273417
PM_{CRT}	1024	32	32	YES	NO	423224
PM_{GMP}	1024	1	1024	NO	YES	535700
$\text{PM}_{\text{CRT,GMP}}$	1024	32	32	YES	YES	2318944
$\text{PM}_{\text{CRT,GMP}}$	1024	16	64	YES	YES	1205310
$\text{PM}_{\text{CRT,GMP}}$	1024	8	128	YES	YES	665287
$\text{PM}_{\text{CRT,GMP}}$	1024	4	256	YES	YES	667428
$\text{PM}_{\text{CRT,GMP}}$	1024	2	512	YES	YES	492946

Table 6.5: CPU implementations of the PM given $N = 2^n = 16384$. The rows within the same row delimiters perform the same PM (are comparable).

6.4 Pease's Polynomial Multiplication (FPGA)

As explained in section 5, Pease's algorithm was firstly used because of its constant geometry. Our version of Pease's algorithm uses Streams instead of Large Memory since the amount of data to transfer can be transferred in less than 300 microseconds (for the largest case N=32768) via PCIe bus, not being worth calling two more times (read and write) if using LMem. Regarding resource usage, table 6.6 shows percentage of them and latency for a polynomial length of $N = 16384 = 2^{13}$, and $bits(p_i) = 64$. Using CRT and calling several times to this DFE's algorithm one can generate a size of q as big as desired.

Version	LUT	FFs	BRAMs	DSPs	latency
Pease's	62588(18.13%)	160352(23.23%)	1466(72.79%)	1090(68.55%)	178

Table 6.6: Pease's algorithm final version resources and latency.

Note that the latency is just the number of clocks needed to have the first output, assuming the i-th output is generated in the i-th tick. Given that for this algorithm we need to calculate several stages within the FPGA, the number of ticks is significantly bigger than the length of the polynomials, which means that throughout the first ticks the output is not correct, but just undefined values. Specifically, the number of ticks is $(2*n+1)*N/2$; being $N-1 = 2^n-1$ the maximum degree of the polynomials. Also, the operating frequency of the DFE for this algorithm is set to 180MHz. As studied before, depending on the specific second most significant bit of each p_i, the resulting resources may vary (because of the Fermat-like modular reduction). Nevertheless, in order to achieve a generic algorithm (that is, being able to call the same algorithm with different precomputed parameters), the implementation performs for the worst case possible. Taking into account that, as previously mentioned, the worst case for the primes generated by the block **PRECOMPUTED PARAMETERS** has its second most significant in the 32nd position, whereas the first (and best) prime generated has it in the 26th, each time algorithm 8 is called it computes with value

$m = 32$, iterating more times than needed for some cases where m can be smaller than 32. These differences do not affect latency (hence execution time) significantly, having to increase in 0.89% the execution time for the best case, whereas it does affect resources usage (see tables from section 6.2). However, table 6.6 shows a large amount of resources still unused, although not enough to enable parallelization within the same DFE via CRT or multiple pipes (since the amount of BRAMs and DSPs use is already bigger than 50% and having two PMs would require at least twice the amount of resources). Still, as seen in chapter 5, the parallelization is performed with Cooley's approach, and not with Pease's. Later in this chapter an optimised version of the same Pease's implementation but using Cooley's I/O patterns will be shown.

Table 6.7 shows a comparison between the fastest CPU version (which, as seen in table 6.5 is always case $PM_{CPU_{CRT}}$) , the equivalent CPU version to the one in the FPGA ($PM_{CPU_{CRT,GMP}}$) and the FPGA Pease's version ($PM_{FPGA_{Pease's}}$). Note that, for each call to the DFE (each p_i), data must be converted from GMP into an FPGA-compatible format. Table 6.7 provides times including such conversion. One can see that DFE's implementation ($PM_{CRT,FPGA}$) is much faster even though these times include the computation (in CPU) of converting data into CRT format and back (which is performed in CPU for all the three cases). For a table without such times see tables 6.8. Also, figure 6.1 avoids such conversion, showing faster times for the FPGA relative to CPU times.

One can see in figure 6.1 and table 6.8 that the FPGA is significantly faster compared to the equivalent algorithm in CPU (PM_CPU_CRT_GMP), obtaining a speedup of more than 11 when comparing the case bits(q)=1088. Also, when compared with the best case for CPU, the speed up is of more than 3.

case	bits(q)	num_{p_i}	bits(p_i)	CRT	GMP	Microseconds
$\text{PM}_{CPU_{CRT}}$	1088	34	32	YES	NO	394091
$\text{PM}_{CPU_{CRT,GMP}}$	1088	17	64	YES	YES	1253793
$\text{PM}_{FPGA_{Pease's}}$	1088	17	64	YES		181320
$\text{PM}_{CPU_{CRT}}$	576	18	32	YES	NO	203004
$\text{PM}_{CPU_{CRT,GMP}}$	576	9	64	YES	YES	656712
$\text{PM}_{FPGA_{Pease's}}$	576	9	64	YES		84988
$\text{PM}_{CPU_{CRT}}$	320	10	32	YES	NO	124500
$\text{PM}_{CPU_{CRT,GMP}}$	320	5	64	YES	YES	364278
$\text{PM}_{FPGA_{Pease's}}$	320	5	64	YES		50670

Table 6.7: FPGA Pease's version, its equivalent CPU implementations and fastest CPU implementation, taking into account conversion into CRT, for a polynomial length N=16384

case	bits(q)	num_{p_i}	bits(p_i)	CRT	GMP	Microseconds
$\text{PM}_{CPU_{CRT}}$	1088	34	32	YES	NO	300931
$\text{PM}_{CPU_{CRT,GMP}}$	1088	17	64	YES	YES	1160633
$\text{PM}_{FPGA_{Pease's}}$	1088	17	64	YES	-	88160
$\text{PM}_{CPU_{CRT}}$	576	18	32	YES	NO	153684
$\text{PM}_{CPU_{CRT,GMP}}$	576	9	64	YES	YES	627392
$\text{PM}_{FPGA_{Pease's}}$	576	9	64	YES	-	55668
$\text{PM}_{CPU_{CRT}}$	320	10	32	YES	NO	97100
$\text{PM}_{CPU_{CRT,GMP}}$	320	5	64	YES	YES	336878
$\text{PM}_{FPGA_{Pease's}}$	320	5	64	YES	-	23270

Table 6.8: FPGA Pease's version, its equivalent CPU implementations and fastest CPU implementation, without taking into account conversion into CRT, for a polynomial length N=16384

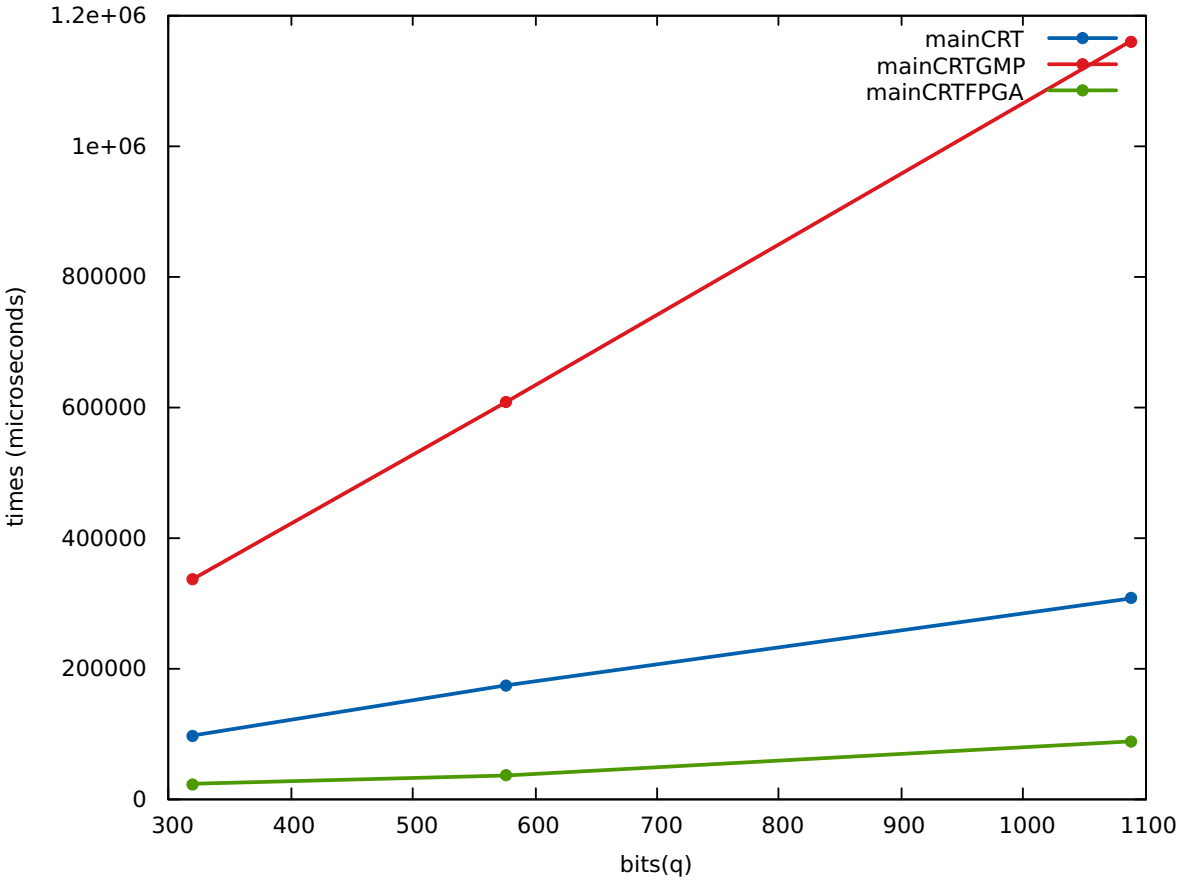


Figure 6.1: FPGA Pease's version, its equivalent CPU implementations and fastest CPU implementation, without taking into account conversion into CRT, for a polynomial length N=16384

6.4.1 Generic Polynomial Multiplication

One of the drawbacks of using CRT for the PM is that it requires a specific coprime q of known factorization $q = \prod p_i$. However, given any two polynomials over $Z[x]$, by taking the maximum degree of them, and the maximum coefficient $coeff$, one can use the DFE as long as $q > degree * coeff^2$ and $N = 2^n \geq degree$. One can see that doing this requires more operations over bigger data than simply multiplying them using PM without CRT (case $PM_{CPU_{GMP}}$, but the advantage of doing it is the possibility of using FPGA's implementation, since without using CRT one can not split the computation of big integers into different calls to the FPGA and therefore one can not use the proposed FPGA algorithm (since the coefficients are too big to fit with the available modules). Table 6.9 shows CPU times and FPGA times using this approach. In this case, times are shown including CRT conversions when needed, since in this case neither input nor output data can be in CRT format. Again, all the times shown are again for case $N = 2^n = 16384$ being $PM_{FPGA_{CRT}}$ being the fastest of them. Also, remember that changing the value of any of $bitsize(q), num_{primes}$ and $bitsize(p_i)$ modifies the rest of them. Notice that, unlike in the previous tables, in table 6.9 makes sense comparing $PM_{CPU_{GMP}}$ with smaller bitsize(q) than the rest of the cases, since such case does not need CRT and thus it can perform the PM over $\overline{q}$, $\overline{q}$ being the number (not necessarily coprime as q) that defines the ring $R_{\overline{q}}$ of the elements that are multiplied. Finally, when having $bitsize(p_i) \geq 64$ (case $PM_{CPU_{CRT,GMP}}$) the first of the p_i is actually of size 64, (shown as $p_0 \leftarrow 64$ in table 6.9), in order to obtain a small value of $bitsize(q)$. Similarly, when $bitsize(p_i) = 32$, the value of $bitsize(q)$ can be smaller than for cases with $bitsize(p_i) \geq 64$, since $bitsize(q)$ has to be a power of 32 instead of a power of 64.

case	bitsize(q)	num_{p_i}	bitsize(p_i)	CRT	GMP	Microseconds
$\text{PM}_{CPU_{CRT}}$	1056(≥1038)	33	32	YES	NO	380391
$\text{PM}_{CPU_{GMP}}$	512	1	512	NO	YES	230858
$\text{PM}_{CPU_{CRT,GMP}}$	1056(≥1038)	33	32	YES	YES	2296859
$\text{PM}_{CPU_{CRT,GMP}}$	1088(≥1038)	17	64	YES	YES	1253793
$\text{PM}_{CPU_{CRT,GMP}}$	1088(≥1038)	9	128($p_0 \leftarrow$64)	YES	YES	794472
$\text{PM}_{CPU_{CRT,GMP}}$	1088(≥1038)	5	256($p_0 \leftarrow$64)	YES	YES	597683
$\text{PM}_{CPU_{CRT,GMP}}$	1088(≥1038)	3	512($p_0 \leftarrow$64)	YES	YES	529330
$\text{PM}_{FPPGA_{Pease's}}$	1088(≥1038)	17	64	YES	-	181320
$\text{PM}_{CPU_{CRT}}$	544(≥526)	17	32	YES	NO	193134
$\text{PM}_{CPU_{GMP}}$	256	1	256	NO	YES	129443
$\text{PM}_{CPU_{CRT,GMP}}$	544(≥526)	17	32	YES	YES	1177621
$\text{PM}_{CPU_{CRT,GMP}}$	576(≥526)	9	64	YES	YES	656712
$\text{PM}_{CPU_{CRT,GMP}}$	576(≥526)	5	128($p_0 \leftarrow$64)	YES	YES	425017
$\text{PM}_{CPU_{CRT,GMP}}$	576(≥526)	3	256($p_0 \leftarrow$64)	YES	YES	332477
$\text{PM}_{FPGA_{Pease's}}$	576(≥526)	9	64	YES	-	84988
$\text{PM}_{CPU_{CRT}}$	288(≥270)	9	32	YES	NO	103647
$\text{PM}_{CPU_{GMP}}$	128	1	128	NO	YES	86577
$\text{PM}_{CPU_{CRT,GMP}}$	288(≥270)	9	32	YES	YES	616060
$\text{PM}_{CPU_{CRT,GMP}}$	320(≥270)	5	64	YES	YES	364278
$\text{PM}_{CPU_{CRT,GMP}}$	320(≥270)	3	128($p_0 \leftarrow$64)	YES	YES	246213
$\text{PM}_{FPGA_{Pease's}}$	320(≥270)	5	64	YES	-	50670
$\text{PM}_{CPU_{CRT}}$	160(≥134)	5	32	YES	NO	58575
$\text{PM}_{CPU_{GMP}}$	64	1	64	NO	YES	69875
$\text{PM}_{CPU_{CRT},CPU_{GMP}}$	160(≥134)	5	32	YES	YES	342770
$\text{PM}_{CPU_{CRT,GMP}}$	192(≥134)	3	64	YES	YES	218448
$\text{PM}_{FPGA_{Pease's}}$	192(≥134)	3	64	YES	-	23581

Table 6.9: Times for using CRT for multiplicative groups that are not suitable for them by extending the group to a bigger one. PM_{GMP} represents the multiplication not using CRT. Rows in the same block perform exactly the same multiplication.

6.5 Cooley's Polynomial Multiplication (FPGA)

In order to achieve a parallel version of the DFE algorithm, several optimizations where made (including reducing the amount of precomputed data stored to reduce BRAM usage and increasing the multiplication paths increasing ticks in order to reduce DSPs). Also, the I/O patterns changed from Pease's to Cooley-Tukey's, since it supports better parallelization within the FPGA. As explained in chapter 5, there are several levels of parallelization, that can be divided into horizontal (using CRT, independent PMs) and vertical (within the same PM multiple pipes).

Table 6.10 shows resources, latency and execution time of different configurations of Cooley's approach, including parallel cases (num_{crt} represents horizontal parallelism and num_{pipes} vertical parallelism). Note that there is no case with $num_{crt} \neq 1$ in table 6.10, given that, even though the algorithm supports horizontal parallelism within the FPGA, the implementation exceeds BRAMs when increasing its value. It also should be remarked that, when focusing on the extended resources of the case $num_{pipes} = 4$, the resource usage is maximize, having all of them more than 77% usage, apart from secondary FFs and LUTs. This means that we are using most of the available resources in our specific FPGA. notice how, when having $num_{pipes} > 1$, the number of ticks increases by $N/2$ (one stage), since, instead of executing in the first same stage the multiplication ϕ_i and the butterfly, we split that into another stage, reusing the same DSPs for both multiplications, in order to save some DSPs.

With table 6.10, and with some of the times already shown in table 6.7, one can generate the times shown in table 6.11. Notice that, in this case, the times follow the same structure as in table 6.11, but removing the common parts executed in CPU (that is, the conversions for and back CRT), in order to show the speedup of using FPGA. Besides, one can execute the whole BGV without executing such conversion.

Nevertheless, one should note all this times include a conversion into FPGA-compatible types, as well as many of them also include a conversion using CRT (prior to the conversion into FPGA-compatible types). Again, although for the

num_{crt}	1	1	1	1
num_{pipes}	1	2	2	4
Clock Frequency	180MHz	100MHz	180MHz	100MHz
Ticks	(2*n+1)*N/2	((n+2)*N)/num_{pipes}	((n+2)*N)/num_{pipes}	(n+2)*N)/num_{pipes}
N=2^n	32768	32768	32768	32768
LUTs	35644(10.33%)	80757(23.39%)	80757(23.39%)	145381(42.12%)
FFs	92648(13.42%)	203129(29.42%)	203129(29.42%)	343836(49.80%)
BRAMs	1226(60.87%)	1041(51.69%)	1041(51.69%)	1402(69.61%)
DSPs	645(40.57%)	1261(79.31%)	1261(79.31%)	1494(93.96%)
Latency	106	93	93	98
Microseconds	4794	3303	2533	2029

Extended Resources	case num_{pipes}=4
Logic utilization:	134108 / 172600 (77.70%)
Primary FFs:	291147 / 345200 (84.34%)
Secondary FFs:	93265 / 345200 (27.02%)
Multipliers (18x18):	2988 / 3180 (93.96%)
DSP blocks:	1494 / 1590 (93.96%)
Block memory (M20K):	1769 / 2014 (87.84%)

Table 6.10: Results and configuration of Cooley's different implementations, and extended resource usage of Cooley-Tukey's parallel implementation ($num_{pipes} = 4$) within the FPGA. num_{CRT} represents the number of parallel multiplication over different p_i (horizontal parallelism). num_{pipes} represents the number of pipes used within each multiplication over $\mathbb{Z}_{p_i}$ (vertical parallelism).

specific state of the algorithm it is necessary to perform them, it is possible to design an algorithm that does not require any of such conversions. For such reason, one can reasonably compare times avoiding both of them. For instance, in table 6.11, avoiding conversions into FPGA-compatible format, the actual times for the FPGA correspond to 114920 and 34493 microseconds for Pease's and Cooley's final versions, respectively. One should notice a speedup of 4.84 when comparing our best FPGA case ($FPGA_{Cooley's}$) with our best CPU case CPU_{CRT}. Moreover, our fastest FPGA case is 2.22 times faster than our best Pease's implementation and 18 times

case	bits(q)	num_{p_i}	bits(p_i)	CRT	GMP	Microseconds
CPU_{CRT}	1088	34	32	YES	NO	317243
$\text{CPU}_{\text{CRT,GMP}}$	1088	17	64	YES	YES	1176945
$\text{FPGA}_{Pease's}$	1088	17	64	YES	-	145972
$\text{FPGA}_{Cooley's}$	1088	17	64	YES	-	65545

Table 6.11: Execution times of the final versions of each of the algorithms, compared to its equivalent in CPU and with the fastest CPU algorithm for $N = 32768 = 2^n$

faster than the equivalent algorithm in CPU. Furthermore, if avoiding the previously mentioned conversions, and taking into account a time of 34493 microseconds instead of 65545, the speedup increases to 9.19 and 34.12 when compared with CPU_{CRT} and $CPU_{CRT,GMP}$, respectively.

6.5.1 BGV (CPU vs FPGA)

Having shown PM times, table 6.12 represents the different times for Encryption and Decryption of the BGV algorithm. Notice that this times actually include conversions into FPGA-compatible types, assuming a great part of the computation time.

Case	Encryption Time	Decryption Time
CPU_{CRT}	669687	324449
$\text{CPU}_{\text{CRTGMP}}$	2389091	1184151
$\text{FPGA}_{Pease's}$	327145	153178
$\text{FPGA}_{Cooley's}$	166291	72751

Table 6.12: Encryption and decryption times depending on the chosen multiplication.

Table 6.12 shows a speedup of $FPGA_{Cooley's}$ of 4.02 when encrypting and 4.459 decrypting compared with our fastest CPU implementation. Moreover, compared to $FPGA_{Pease's}$, the speedup is of roughly 2 for both cases. Compared with the equivalent CPU implementation, $FPGA_{Pease's}$ is 14.36 times faster encrypting and 16.27 times decrypting.

Apart from our CPU implementation, compared to NTL ZZ_pX types, the execution of the PM for N=32768 and bits(q)=1024 is 404573 microseconds, which gives a speedup for the DFE of 12. Moreover, for the case N=32768, bits(q)=64; that is, calling the DFE once, the speedup is 16, avoiding conversion time.

6.6 Comparison with Other Work

Comparing performance and resources usage with related work is not easy, specially in cases where the targeted board uses a complete different architecture as a result of being designed by a different company, such as [23, 27]. [27] does not report actual times of a polynomial multiplication, but rather clocks of implemented algorithms within the FPGA, excluding interfacing with CPU. As any of the algorithms in the FPGA is the actual polynomial multiplication, but rather some parts of it, it is not possible to compare our implementation with theirs in a fair way. Also, their FPGA (Xilinx Virtex-7 XC7V1140T-2) is the largest device of the Virtex-7 FPGA family, but their resources are underused (with 53% usage of DSP48 as the biggest usage percentage). Our implementation, though using a smaller device in terms of resources, maximize the resources at our disposal. Nevertheless, as their device disposes more resources, we appear to use less LUTs and DSPs, whereas the same amount of FFs (Registers) and more BRAMs (which is necessary when avoiding several calls to the FPGA so as to implement the polynomial multiplication, as they would need).

Öztürk et al. [23] seem to use about half the area (resources) compared to our presented implementation. Again, given their targeted board (Xilinx Virtex 7 XC7VX690T), it is difficult to compared with accuracy. Their design allows them to execute using 250MHz, which is not possible for our targeted board, with 200MHz as highest frequency to ensure the correctness of the output data. Also, as they implement the polynomial reduction outside of the FPGA, they need to use less resources and have a lower latency. Nevertheless, they obtain a total time for the execution of a full polynomial multiplication (as explained in this project) of 28.51

milliseconds, compared to 40 milliseconds that we obtain with the same parameters (executing at 100MHz). However, most of this time takes place in CPU (because they compute the polynomial reduction in CPU) meaning that, though they have more resources in the FPGA to parallelize more the (I)NTTs, their project scales worse with the power of the FPGA. Furthermore, their resource usage is at about 50% (with a bigger board compared to ours), meaning that their approach spends more energy than ours. In any case, the comparison seems odd, given the disparity of both FPGAs and designs.

Pöppelmann et al. [26] use a fairly comparable Altera board (Stratix V 5GSND5H). Their modular reduction (Solinas) allows them to reduce their DSP usage to 577. However, this reduction is designed for a subset of all the primes. Besides, they do not use CRT, having to lead with big word sizes and limiting the possibility of R_q having q to be a Solinas primes, instead of a product of Solinas primes. There is no significant difference with the rest of the resources. Furthermore, our execution would complete a full polynomial multiplication in less than 16.37ms, compared to the 27.88ms they need.

Finally, any of the mentioned projects report actual complete times of encryption or decryption, though some of them report times of the primitives of a FHE Cryptosystem. We presented, to the best of our knowledge, the first execution times of a complete Cryptosystem in an FPGA-based system. Also, thanks to using Maxeler Tools, our design would be easily parallelized by using more than FPGA, something already described as 'promising' by Pöppelmann et al. [26].

Chapter 7

Conclusions and Future Work

We showed that modern FPGAs have sufficient resources for efficient computation of primitives required for evaluating functions on FHE encrypted data. Despite this, maximizing the resources of its FPGA is critical, and interfacing with CPU should be minimized. For this purpose, a proper approach would be an implementation of the algorithm that admits several sequential PMs with one call, avoiding overhead derived from calling several times. This approach is easy to implement. Also, for the sake of saving time, the FPGA code can be executed asynchronously the computer.

We presented several designs of modular reduction, showing the benefits of using the Fermat-like modular reduction when the number to reduce by is not Mersenne-like. In other cases, Barret Reduction proof to make a better job. Apart from this, Fermat-like modular reduction should always be used when having plenty of resources left. Also, we studied different versions of the PM in the FPGA and in CPU.

To the best of our knowledge, we present here the first FPGA implementation used by an encryption system, with actual times of such implementation as well as the encryption and decryption algorithm. We saw the benefits of using CRT for splitting the PM of big integers into several PMs of smaller ones. Moreover, we showed advantages of parallelizing using Cooley-Tukey's NTT algorithm rather than Pease's. In order to accelerate the execution, we consider using several FPGAs

in the future working together to enable parallel calls to several PMs over 64bits. For example, given the case $bits(q) = 1088,\ N = 32768$, one could use 17 FPGAs to parallelize all 17 of the PMs over 64bits. One execution of the FPGA algorithm takes 2029 microseconds, of which 310 microseconds are needed for data transfer. Since the 17 FPGAs would theoretically use the same PCIe bus (thanks to the DFEs designs by Maxeler), the final computation of all the 17 calls wold take 6989 microseconds, compared with 34494 microseconds of a sequential call. Finally, we are studying ways of increasing the frequency of the case $num_{pipes} = 4$ since that would allow a decrease of over 500 microseconds in each of the calls to FPGA.

As changes in the implementation itself, we are studying the use of multiple CRTs within the same FPGA by reducing the bits of each p_i to 32bits, avoiding the use of GMP and also the conversion into FPGA-compatible format. Furthermore, comparing different bit lengths implementation's resources, latencies and execution times to conclude which one is more recommendable for the FPGA implementation would be an interesting study, the same way it was done for the CPU one. Using more than 64bits would require declaring variables in the FPGA of more than 128bits feature Maxeler is working on in order to provide MaxJ with that and bigger bit sizes for variables at the moment. Finally, we also considered introducing our different PMs into HELib, replacing the built-in PM in HELib, in order to study the speedup of both algorithms in a reputable HE library.

Bibliography

[1] *Bambu: A Free Framework for the High-Level Synthesis of Complex Applications.*

[2] *FpgaC compiler.*

[3] Maxeler technologies.

[4] *Nios-II C-to-Hardware Acceleration Compiler.*

[5] *ROCCC: Riverside Optimizing Compiler for Configurable Computing.*

[6] B.M. Baas. A generalized cached-fft algorithm. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, volume 5, pages v/89–v/92 Vol. 5, March 2005.

[7] M. Baas. An approach to low-power, high performance, fast fourier transform processor design. Technical report, 1999.

[8] JoppeW. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *Cryptography and Coding*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer Berlin Heidelberg, 2013.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[10] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Advances in Cryptology–CRYPTO 2011*, pages 505–524. Springer, 2011.

[11] D.D. Chen, N. Mentens, F. Vercauteren, S.S. Roy, R.C.C. Cheung, D. Pao, and I. Verbauwhede. High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 62(1):157–166, Jan 2015.

[12] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19, 1965.

[13] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. Cryptology ePrint Archive, Report 2014/202, 2014.

[14] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 47–56, New York, NY, USA, 2012. ACM.

[15] Haohuan Fu, Robert G. Clapp, and Olav Lindtjorn. *Revisiting Convolution and FFT on Parallel Computation Platforms*, chapter 602, pages 3071–3075. 2015.

[16] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.0 edition, 2015.

[17] David Harvey. Faster arithmetic for number-theoretic transforms. *J. Symb. Comput.*, 60:113–119, 2014.

[18] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. *SWIFFT: A Modest Proposal for FFT Hashing*, volume 5086 of *Lecture Notes in Computer Science*, pages 54–72. Springer Berlin / Heidelberg, 2008.

[19] Sparsh Mittal and Jeffrey S. Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Comput. Surv.*, 47(2):19:1–19:23, August 2014.

[20] Mervin E. Muller. A comparison of methods for generating normal deviates on digital computers. *J. ACM*, 6(3):376–383, July 1959.

[21] Henri J Nussbaumer. *Fast Fourier transform and convolution algorithms*, volume 2. Springer Science & Business Media, 2012.

[22] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[23] Erdinç Öztürk, Yarkın Doröz, Berk Sunar, and Erkay Savaş. Accelerating somewhat homomorphic evaluation using fpgas. Cryptology ePrint Archive, Report 2015/294, 2015.

[24] Marshall C. Pease. An adaptation of the fast fourier transform for parallel processing. *J. ACM*, 15(2):252–264, 1968.

[25] John M Pollard. The fast fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971.

[26] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrián Macías. Accelerating homomorphic evaluation on reconfigurable hardware. *IACR Cryptology ePrint Archive*, 2015:631, 2015.

[27] Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil Dimitrov, and Ingrid Verbauwhede. Modular hardware architecture for somewhat homomorphic function evaluation. Cryptology ePrint Archive, Report 2015/337, 2015.

[28] Victor Shoup Shai Halevi. *HELib: Homomorphic-Encryptio Library*, 2015.

[29] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge university press, 2009.

[30] Victor Shoup. *NTL: A Library for doing Number Theory*, 9.6.2 edition, 2015.

[31] Jerome A. Solinas. Generalized mersenne numbers. Technical report, 1999.

[32] Joachim von zur Gathen. J.gerhard: Modern computer algebra, 1999.

[33] Reto Zimmermann. Efficient vlsi implementation of modulo (2n plusmn;1) addition and multiplication. In *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, pages 158–167, 1999.