INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Parallel Execution of RSA Encryption on the GPU of the RPi

Emma Munisamy

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Parallel Execution of RSA Encryption on the GPU of the RPi

Emma Munisamy

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Jan Schmidt Tobias Guggemos
Abgabetermin:	5. Dezember 2019

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 5. Dezember 2019

. . . .

(Emma Munisamy)

Abstract

The Raspberry Pi is a 2012 published single-board computer purposed for private usage. It includes a VideoCore IV graphics card. In the case that it is used headless the potential performance of the GPU remains unused. This work uses this capacity for parallel execution of the cryptographic algorithm RSA. RSA is an asymmetric encryption system based on the mathematical difficulty in factoring very big integers in finite time. The most important part of the algorithm is an efficient implementation of modular exponentiation. During this work, different possibilities are developed how modular exponentiation for RSA could be performed on the VideoCore IV. One of these possibilities was implemented using modular exponentiation partly processed parallel on the GPU of the Raspberry Pi B+, using the open source c++ library QPULib for programming the VideoCore IV processing units, the QPUs. Although the GPU implementation doesn't accelerate RSA, this work is a first step of how the RSA execution can be executed on the VideoCore IV, and what difficulties complicate the realization. Additionally the developed GPU executed RSA was integrated to OpenSSL.

Contents

1 Introduction

The Raspberry Pi is a single board computer, which has a relatively weak performance, but is still very often used because of its low price and power consumption. Mostly it is used for pure computing power without a screen. All calculations are done by the CPU, while the GPU only consumes power and is not used. Thus the resources of the Raspberry Pi are anything but evenly utilized. At the same time, the security of digital data is becoming increasingly important. Many calculated results are cryptographically encrypted before they are transferred to another system. This means an additional load for the CPU, on top of the actual calculation. These two considerations led to the idea to try to offload cryptographic calculations on the Raspberry to its GPU. Since RSA is still used very often even though it is a complex computation and because it requires a lot of CPU computing capacity due to this complex computation, in the course of this work an attempt is made to execute parts of the RSA encryption on the GPU. This way the usage of the Raspberry Pi resources should become better distributed.

Furthermore the GPU of the Raspberry Pi, the VideoCore IV by Broadcom, has a really large computational power. Due to its Single-Instruction-Multiple-Data architecture, it can process more operations in parallel which leads to a theoretically better throughput than the CPU possesses. Because of this maybe an even shorter execution time of RSA can be achieved by executing it on the GPU. To ensure the security of the RSA encryption, increasingly larger numbers are used for it. Since these numbers are sometimes a hundred times larger than the machine word of a personal computer, multiprecision operations are required. They lead to really long execution time of the RSA cryposystem compared to other encrypting algorithms. That's why it is even more desirable to shorten the execution time by using the GPU.

In this work it is investigated whether it is possible to offload parts of the RSA encryption to the GPU. Besides, it might be possible to draw a conclusion whether the execution of other cryptographic algorithms is also possible, and if so, for which one the greatest advantages could be expected. Possible advantages, that are also intended for RSA, would be a relief of the CPU and perhaps even an acceleration of the algorithm's computing process.

For this purpose several approaches of GPU executed RSA are developed. They are differing in the parts of RSA that are executed on the GPU and additionally how the SIMD property of the GPU is effectively used. One of these approaches is successfully implemented on the VideoCore IV and tested in comparison to a pure CPU executed RSA algorithm. Although the results speak against an advantage of the implementation, the analysis of the implementation serves for determining the advantages and disadvantages of the VideoCore IV. The result does not speak against general purpose programming of the GPU in general, but shows that some algorithms for explainable reasons are not well suited for execution on the VideoCore IV.

In this way, this work serves as a fundament to either implement another of the developed GPU executed possibilities of RSA, whereby the found bottlenecks can now be avoided. Or to use the results for a good choice, where other cryptographic algorithms could be offloaded to the VideoCore IV with better prospects, and to implement them successfully.

Structure

Now that the motivation and contents of this work have been explained, a detailed introduction into the mechanism of RSA and the principles used to calculate the required modular exponentiation follows. This is followed by an overview of the Raspberry Pi and its GPU, the VideoCore IV. In addition, the QPULib, the library used to program the VideoCore IV with C++ code is introduced.

The next part of the thesis is the design of an RSA encryption algorithm, which can be partly calculated on the GPU and uses the possibility of processing multiple data in parallel.

Once an algorithm was found that can be implemented on the Raspberry Pi, it was implemented using the QPULib. This is described in chapter 4 with all interesting implementation details and VideoCore IV specific features.

Finally, the partial GPU-executed RSA implementation was tested and compared to the full CPU-executed RSA implementation of OpenSSL. Therefore an OpenSSL engine was built, which enables RSA encryption on the Raspberry Pi's GPU through OpenSSL. The results are presented and discussed in chapter 5.

2 Background

In this chapter the background and basic principles used in the developed RSA implementation are explained.

At the beginning the cryptosystem RSA and its functionality is introduced. The basis of RSA encryption and decryption is the modular exponentiation on very large numbers. These will be 1024 bits large for our implementation, since RSA is only considered safe from attacks starting at this number size.

Therefore a short insight into the computer calculation of numbers is given, which are too large to fit into a machine word, so-called multiprecision integers.

This is followed by an introduction on how modular exponentiation can be implemented efficiently and applied to multiprecision integers.

In addition to the RSA Cryptosystems and its implementation approaches, also the hardware of the Raspberry Pis belongs to the background of this work. Therefore the Raspberry Pi with its GPU, the VideoCore IV in focus, is introduced in this chapter as well. The tool used for the GPU's programming, the QPULib, is also explained.

And finally some papers are presented, which are closely related to the topic of this work and have also served as a background for this work and could be helpful for similar works.

2.1 The RSA Cryptosystem

RSA is a public cryptosystem invented by R. L. Rivest, A. Shamir and L. Adleman in 1978 [RSA78]. It's benefit is that it can be used for encryption without requiring secure connections for key exchange. That's the reason why, although it takes a relatively long execution time, it is still used widely, often to transmit a symmetric key between two communication partners over an insecure channel to enable symmetric encryption after that. In addition, it is used for authentication with signatures.

The security of the RSA procedure is based on the factorization problem. But due to the increasingly powerful computers, larger and larger numbers have to be used for the procedure to be still secure. A public key of the length 768 bits has already been cracked [KAF$^+$10]. This is why this work will develop a RSA 1024-bit implementation, which is still considered as secure.

In the following, first the general mechanism of public cryptosystems is explained, then the RSA cryptosystem is explained in detail.

2.1.1 Public Cryptosystems

The concept of public (or asymmetric) encryption was published in 1976 by W. Diffie and M. Hellman [BSW15]. It is based on the principle, that different keys are used for encryption and decryption. By this way one key can be published without increasing insecurity. During public cryptosystems the sender and the receiver do not have the same (symmetric) key,

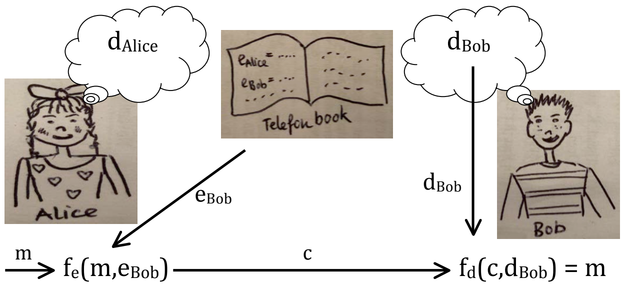


Figure 2.1: Encryption with a public cryptosystem

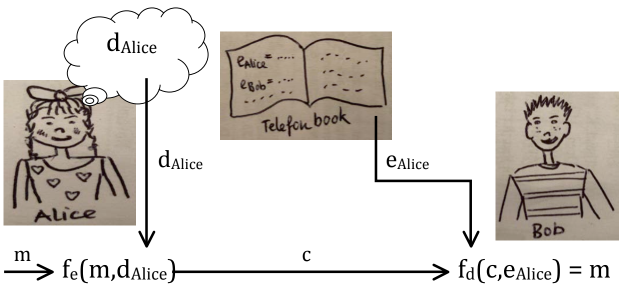


Figure 2.2: Using a public cryptosystem for signatures

which would have to be exchanged via a secure channel, but different keys that complement each other[BSW15].

In the following, the procedure of asymmetric encryption is described with Alice and Bob, who want to communicate over a public channel safely with each other using a secure public cryptosystem (see figure 2.1). For this all communication participants have a matching key pair consisting from a private key d and a public key e. The public key e is public to all communication participants, e.g. in something like a telephone book, while the private (or secret) key d is known only by the owner. Now if Alice wants to send a secret message to Bob, she searches something like a public telephone book for Bob's public key e_{Bob}. She encrypts her message m with $f_{\text{e}}(m, e_{\text{Bob}}) = c$ and sends the resulting cipher c to Bob. The channel used for this does not have to be secure because c is encrypted and can only we decrypted with the matching key d_{Bob}. As soon as Bob receives the cipher c, he can decrypt and read it with the function $f_{\text{d}}(c, d_{\text{Bob}}) = m$. If Bob wants to send something to Alice, he does the same, but instead of e_{Bob} he chooses the public key e_{Alice} from the phone book and Alice uses her own private key d_{Alice} to decrypt it [BSW15].

In addition to sending messages securely an asymmetric encryption method can be used for authentication with signatures (see figure 2.2) [Buc01]. If Alice wants to show that a message originates from her, in other words she wants to sign this message, she proceeds as follows. She encrypts the message with her private key d_{Alice}. $f_{\text{e}}(m, d_{\text{Alice}}) = c$. Each communication participant can now check the message by decrypting c with Alice public key. $f_{\text{d}}(c, e_{\text{Alice}}) = m$. If m makes sense, it has been encrypted with Alice's secret key and cannot come from anyone else but her [Buc01].

By using both introduced methods a message can be signed and send securely (see figure 2.3).

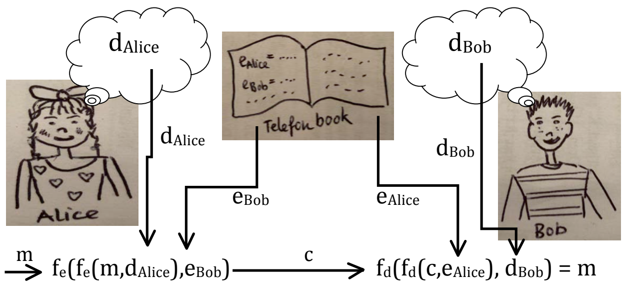


Figure 2.3: Using a public cryptosystem for signing and encrypting a message

2.1.2 The RSA Algorithm

RSA is a public cryptosystem working as explained in the former chapter. Now it will be explained how the public and private keys used for RSA look exactly like. And how the encryption and decryption functions are defined for RSA. Therefore this section is divided into 3 parts:

- RSA key generation
- RSA encryption f_{e} and RSA decryption f_{d}

Further it shall be mentioned that here only schoolbook RSA is explained. This means that the basic mathematical functions of RSA are explained without padding the message before encryption. Consequently, the resulting cipher will not be safe from attacks, because every encryption of the same message will result in exactly the same cipher. To prevent this, the message to be encrypted would have to be padded before real use, so that the same messages will result in different ciphers. The default padding method is RSAES-OAEP [rfc]. However, since this is a separate procedure, it does not belong to the RSA algorithm, although it must be used to make RSA encryption really secure. At this point it is already mentioned beforehand that this work implements textbook RSA without padding. The preceding padding of the message can then be done manually by the user before the encryption.

Key Generation

The public and private keys for the public RSA cryptosystem are consisting of two numbers. The public key is defined as (e, n) and the private key is a tuple of (d, n), where N is the same in both keys in a matching key pair. Now e will not stand for the whole RSA pubic key anymore but for one part of the public key tuple, the encryption exponent. The same for the number d that now only stands for the decryption exponent as part of the private key. n will further also be called module. During the key generation the matching key pair is extracted. The two keys are generated depending on two prime numbers p, q. This is done in these two steps:

- Calculating n and $\phi(n)$
- Calculating the exponents e and d

They are explained in the next two paragraphs.

Calculating n and $\phi(n)$: To calculate n select p and q, two prime numbers [NW15]. These must be kept secret, since the key pair, including the private key d, can be reconstructed from them. The primes p and q must be of large size to ensure the security of the RSA procedure. They are selected via a random number generator in combination with a prime number test for example the Miller Rabin procedure.

From p and q n is calculated: $n = p * q$. n is already one part of the public and private keys. For 1024-bit RSA n has to be 1024 bits long. Due to the factorization problem, it is not possible to deduce p and q from n if n is that large.

As a last step $\phi(n)$ is calculated. $\phi(n) = (p - 1) * (q - 1)$

Calculating the exponents e and d: Now from $\phi(n)$ the numbers e and d are calculated [Tit00]. First, the encryption exponent e is selected. e must satisfy $1 < e < \phi(n)$. Usually, e is chosen so that it's binary representation has a low number of the digit 1. In general, the number 65537 is recommended. The public key (e, n) includes the encryption exponent e and the module n. An OpenSSL RSA public key looks like shown in listing 2.1.

```
RSAPublicKey ::= SEQUENCE {
          modulus           INTEGER,  -- n
          publicExponent    INTEGER   -- e
}
```

Listing 2.1: Components of a public key [rfc]

Then the decryption exponent d is calculated. The following must be considered: $d \in N$, $d > 1$, $gcd(d, \phi(n)) = 1$ and $e*d \equiv 1 \pmod{\phi(n)}$ what in other words means $e*d+k*\phi(n) = 1$ for some k. At this point it is to be pointed out, that beside n also d can reach a length of up to 1024 bits.

With the advanced Euclidean algorithm d can be calculated from e and $\phi(n)$. Why this works exactly is explained in section 2.1.3. Listing 2.2 shows what is included in the private key of OpenSSL. If any one of the values, except n and e, gets published, the private key is not secure anymore.

```
RSAPrivateKey ::= SEQUENCE {
          version           Version,
          modulus           INTEGER,  -- n
          publicExponent    INTEGER,  -- e
          privateExponent   INTEGER,  -- d
          prime1            INTEGER,  -- p
          prime2            INTEGER,  -- q
          exponent1         INTEGER,  -- d mod (p-1)
          exponent2         INTEGER,  -- d mod (q-1)
          coefficient       INTEGER,  -- (inverse of q) mod p
          otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

Listing 2.2: Components of a private key [rfc]

Encryption and Decryption

Now the RSA Encryption and Decryption functions are explained. They will be implemented during this work using OpenSSL standard RSA keys, so no key generation has to be implemented.

The Encryption is performed calculating the function $f_{\mathrm{e}}(m) = m^e \mod n = c$ [ZG15].

In order to encrypt a message, it must first be converted into a number. For example the ASCII encoding or base64 encoding can be used for this or a 1024 bits long message is just read as a 1024 bits long number from its binary representation. As a more basic example several characters can be combined before they are sent. Instead of sending 'HI' as $m1 = 72$ and $m2 = 73$, the 7273 can be sent as m at one time. In this case, encryption and decryption must be coordinated. Please note that the number m must be smaller than the module n.

Decryption is performed using the function $f_{\mathrm{d}}(c) = c^d \mod n = (m^e \mod n)^d \mod n = m$ [NW15]. Because c and d are inverse in the multiplicative group modulo n, $(m^e)^d \mod n = m^1 \mod n = m$ as explained in the mathematical background in section 2.1.3. Here as well c is smaller than n

2.1.3 Mathematical Background

As can be seen in the encryption and decryption functions RSA encryption works entirely through modular exponentiation and as already mentioned it is based on the factorization problem. For this reason this chapter tries to give a short overview over the mathematical principles that RSA is based on. This chapter is not necessary for the implementation of RSA, but explains why RSA encryption works. The exact procedure of key generation and encryption of RSA in chapter 2.1.2 is enough to understand the GPU RSA implementation.

First a short explanation of the Factorization Problem is given that ensures the security of RSA. The encryption and decryption of RSA with different keys is only possible because e and d are both inverse elements of the multiplicative group of number with the module n. Why their modular exponentiation on a number after each other results again in the original number is explained afterwards.

Integer Factorization Problem

Factoring a number means splitting the number into its prime factors. Example: $56 = 2 * 2 * 2 * 7$

Compared to multiplying the prime factors, the decomposition takes a very long time [Wag13]. This is a so-called one-way function, since one direction, the multiplication, can be performed very quickly, but the other one, the decomposition into it's prime factors, takes a very long time.

For the use of RSA only the product $n = p * q$ is published. Even with the randomly selected public exponent e, it is not possible to calculate the prime numbers p and q, which could be used to calculate the private key d and decode the cipher, from n. The factorization of n into p and q cannot be solved in finite time for large[Wag13]. Until today it isn't proven, that there isn't a more efficient algorithm for factoring numbers in polynomial time. If it would be found, the factorization problem would be solved and RSA would not be secure anymore.

The security of RSA thus depend on whether the modulo n is so large that it cannot be broken down into its prime factors in finite time. Even though increasing calculation power of modern computers this is currently still true for modulo of size 1024 bits.

Multiplicative group of integers modulo n

The principle which RSA is based on uses multiplicative groups modulo n $(\mathbb{Z}_{\mathrm{n}}, *)$, in particular the inverse elements.

This set $(\mathbb{Z}_{\mathrm{n}}, *)$ consists of all elements that are coprime to n. Example: $(\mathbb{Z}_8, *) = \{1, 3, 5, 7\}$.

Their number can be determined with the Euler's Totient Function ϕ [Sch06]. For a prime number the coprimes are the numbers $\{0, .., n-1\}$. Therefore for a prime number p follows $\phi(p) = p - 1$. Example: $(\mathbb{Z}_{13}, *) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ For a product n of two prime numbers p and q, $\phi(n) = (p-1) * (q-1)$.

With the multiplication modulo n $({}_{\mathrm{n}}*)$, the set $(\mathbb{Z}_{\mathrm{n}}, *)$ forms an abelian group. This means that the group operation ${}_{\mathrm{n}}*$ is commutative and associative, there is a neutral element e and for each element a there is an inverse element a^{-1}. $a * a^{-1} \mod n = a^{-1} * a \mod n = e$.

For the inverse to an element a in any group $(\mathbb{Z}_{\mathrm{n}}, *)$ 1 is the neutral element. It applies $a * a^{-1} \mod n = a^{-1} * a \mod n = 1$. For every element of the group an inverse element is uniquely determined.

Example: Group $(\mathbb{Z}_{13}, *)$, $3 * 9 \equiv 27 \equiv 1 \pmod{13}$.

To find two inverse elements of such a group the Extended Euclidean Algorithm can be used [Sch06]. So it is for the RSA key generation. Why this inverse elements cancel each other out on exponentiation $(m^e)^d \mod n = m$, as it is used for RSA encryption, is proven with Euler's theorem. Both are explained in the following paragraphs.

Extended Euclidean Algorithm The inverse a^{-1} to an element a of a $(\mathbb{Z}_{\mathrm{n}}, *)$ can be calculated using the Extended Euclidean Algorithm. That's why it is used to calculate the decryption exponent d from the encryption exponent e.

In general for two numbers x, y the Extended Euclidean Algorithm returns the greatest common divisor *gcd*, as well as the linear combination $gcd(x, y) = s * x + t * y$, with s,$t \in \mathbb{Z}$ [Sch06].

Since all elements in $(\mathbb{Z}_{\mathrm{n}}, *)$ are coprime to n (which means $gcd(x \in (\mathbb{Z}_{\mathrm{n}}, *), n) = 1$), $1 = a * s + n * t$ applies to all $a \in (\mathbb{Z}_{\mathrm{n}}, *)$. Since the group is the multiplicative group mod n, it follows: $1 \equiv a * s + n * t \equiv a * s \pmod{n}$ That's how we get to: $a^{-1} = s$. So the inverse element to $a \in (\mathbb{Z}_{\mathrm{n}}, *)$ is value s from the Extended Euclidean Algorithm.

Euler's theorem Euler's theorem explains why the RSA cryptosystem is working at all. More in detail it explains why $f_{(\mathrm{f_e(m)})} = m$ [Sch06]. Euler's theorem says:

$$(I.) \quad x^{k*\phi(n)+1} \mod n = x$$

In addition for any element a and its inverse a^{-1} from the group $(\mathbb{Z}_{\phi(n)}, *)$ and $k \in \mathbb{Z}$ can be said:

$$(II.) \quad (x^a)^{a^{-1}} = x^{a*a^{-1}} \equiv x^1 \equiv x^{k*\phi(n)+1} \pmod{\phi(n)}$$

So let a and a^{-1} be elements of the group $(\mathbb{Z}_{\phi(n)}, *)$, together with Euler's theorem then this equation is valid:

$$\begin{aligned}(x^a)^{a^{-1}} \mod n &= x^{a*a^{-1}} \mod n \\ &\overset{(II.)}{=} x^{k*\phi(n)+1} \mod n \\ &\overset{(I.)}{=} x\end{aligned}$$

$$\begin{aligned}(x^a)^{a^{-1}} \mod n &= x^{a*a^{-1}} \mod n \\ &\overset{a*a^{-1}\equiv 1 \pmod \phi)(n)}{=} x^{k*\phi(n)+1} \mod n \\ &\overset{Euler's Theorem}{=} x\end{aligned}$$

Because for RSA also two elements inverse to each other e and $d = e^{-1}$ of the group $(\mathbb{Z}_{\phi(n)}, *)$ are chosen, the encryption and decryption behave like proofed in Euler's theorem. If a message m is encrypted and decrypted, the calculation $(m^e)^{e^{-1}} = m^{\phi(n)+1} = m$ is performed and it results at m again. That is why the RSA cryptosystem works.

2.2 Multiprecision Arithmetic

If calculations are to be made with values that no longer fit into the machine word of a computer, it is called multiprecision arithmetic. This is usually reached with personal computers with a length of more than 32 or 64 bits.

Since the module n needed during safe RSA encryption alone is at least 1024 bits long, and the message m can be almost as large as n, for the modular exponentiation $m^e \mod n$ multiprecision arithmetic is required. In general, multiprecision operations work just like primary school methods, where the large number is processed step by step (usually in machine word size) and the transfer is passed on to the next block in its own variable. Depending on the operation, the length of the result may vary and get much larger than the operands.

How the multiprecision operations were handled in this work can be seen in the Implementation chapter under the GPU implementation 4.2.

A good introduction to multiprecision arithmetic, that also was used for this implementation, can be found in chapter 14 of the "Handbook of applied cryptography" [KMVOV96].

2.3 Modular Exponentiation

For both Encryption and Decryption in RSA algorithm the mathematical operation modular exponentiation is used. There are three different mechanisms or algorithms that are used to enable a more efficient calculation of modular exponentiation in this implementation regarding thousands of bits long numbers as n and d are. These are:

- Residue Multiplication
- Exponentiation by Squaring
- Montgomery Algorithm

How they work and why they are needed is explained in this section.

2.3.1 Residue Multiplication

Residue multiplication is used to keep the length of the numbers of the calculation to be calculated down during modular exponentiation. This reduces the calculation time and the memory space needed.

To calculate $m^e \mod n$ the most obvious way would be to calculate $m^e = x$ and in the next step to calculate $x \mod n$.

In the following section the size of a number refers to the number of bits it needs to be represented and not to the value of the number. That means that the numbers 6 and 7 would have the same length because they both need three bits to be represented.

Considering that m has to be smaller than n, and n is 1024 or even 2048 bits long concludes that m can be that long as well. By raising m to the power of e , x can get $e * 1024$ bits long. That would require a huge part of the memory.

For modular multiplications the following equation can be used [Sha13]:

$$(a * b) \mod n = (a \mod n * b \mod n) \mod n \tag{2.1}$$

That leads to:

$$\begin{aligned} m^e \mod n &= (m * m^{e-1}) \mod n \\ &= (m \mod n * m^{e-1} \mod n) \mod n \\ &= ((((m \mod n * m \mod n) \mod n * (m \mod n * m \mod n) \mod n) \\ &\quad \mod n * ...) * (m \mod n * m \mod n) \mod n) \mod n \end{aligned} \tag{2.2}$$

By using equation 2.2 for the modular exponentiation it can be guaranteed that the numbers of the calculation won't get larger than $2 * 1024$ bits. The reason therefore is that the result of $m \mod n$ is an element of $[0, ..., n-1]$ and wont cost more bits than n. When calculating $m \mod n * m \mod n$ the result can not get bigger than two times the size of $m \mod n$, which is at most the same as $2 * n$. Considering $(m \mod n * m \mod n) \mod n$, this result will again be maximal the same size as n. Therefore no number will get larger than $2 * 1024$ bits.

2.3.2 Exponentiation by Squaring

Exponentiation by squaring is used to reduce the number of multiplications during the calculation of any exponentiation [Flea]. This shortens the computation time.

The calculation from chapter 2.3.1 of $m^e \mod n = [(m \mod n * m \mod n) \mod n * ... * (m \mod n * m \mod n) \mod n] \mod n$ costs e multiplications. This number can be reduced by using a method of exponentiation by squaring. There are three different methods. The first works recursive, the second and third work iterative and use the binary form of the exponent. All three methods reduce the number of multiplication to $2{*}log(e)$ multiplications.

Here the iterative version from least to the most significant bit is used. For $m^e \mod n$ we choose $e = 13$ as example. The modulo can be discarded because it is not relevant for the number of multiplications. So we remain with the formular m^{13}.

13 can be written in binary form as 1101. That representation corresponds to

$$2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 1 = 8 * 1 + 4 * 1 + 2 * 0 + 1 * 1 = 13. \tag{2.3}$$

That leads to

$$\begin{aligned} m^{13} &= m^{2^3*1+2^2*1+2^1*0+2^0*1} \\ &= m^{8*1+4*1+2*0+1*1} \\ &= m^{8^1} * m^{4^1} * m^{2^0} * m^{1^1} \end{aligned} \tag{2.4}$$

Based on the presentation described above the general $power = m^e$ can be calculated with the following algorithm 1.

Algorithm 1 Iterative Exponentiation by Squaring

```
Input: base $m > 0$, exponent $e \geq 0$
Output: $m^e = res$
  $res \leftarrow 1$
  while $e \neq 0$ do
    if $e \mod 2 == 1$ then
      $res \leftarrow res \times m$
    end if
    $m \leftarrow m \times m$
    $e \leftarrow \lfloor e \div 2 \rfloor$
  end while
  return $res$
```

That way the multiplications can be reduced to two times the number of bits needed to represent e, which is $2 * \log_2 e$. As it will be shown in chapter 2.3.3 the division and the modulo operation can be done with a right bit shift and logical AND, so that they won't need many operations relatively to the multiple precision multiplications.

2.3.3 Montgomery Algorithm

For optimizing the modulus operation during a multiplication the Montgomery algorithm was chosen [Fleb]. Among others it uses the technique introduced in chapter 2.3.1. Instead of performing the exponentiation at once and then performing the modulo operation it calculates every modular multiplication after each other. Through this the calculation time and memory space needed is reduced, because the resulting numbers won't get bigger than two times the module. These special modular multiplications performed during the Montgomery algorithm are called Montgomery multiplications. How the Montgomery Algorithm is used exactly will be explained in the following.

Alternatively the Barret Algorithm is often used for modular exponentiation. But for calculations with very large integers Montgomery algorithm has the best performance [BGV93].

At first, the two ideas on which the Montgomery algorithm is based are presented. One is, that the execution of a modulo operation with a power of two 2^x as a module can be done in a very short time by a bitwise AND operation. Second, that a division with a power of two 2^x as divisor can be performed with a right bit shift.

Idea

The Montgomery algorithm uses the fact that the modulo operation with a power of 2 as module and the division with a power of 2 as divisor is easier to calculate than for other numbers. Before the Montgomery algorithm itself is explained, the two mechanisms of how the Modulo and Division operation can be replaced during Montgomery algorithm are shown.

Modulo Operation Normally for calculating $a \mod b$ following algorithm is used:

Algorithm 2 Modulo operation

```
if $a < b$ then
  return a
end if
while $a > b$ do
  $a \leftarrow a - b$
end while
return $a$
```

One modulo operation costs many comparisons and divisions. For multiple precision integers as used in RSA this takes relatively long. Therefore Montgomery uses the following principle for modulo operations with module $n = 2^x$, $x > 0$.

In binary number system, where the base $b = 2$, modulus can easily be calculated for module $n = 2^x = b^x$. All digits that are more significant than n are multiples of n. These can be ignored for the modulus. Example with $a = 217$ and $n = 2^3 = 8$ in binary representation:

$$(217)2 = 1101\ 1001$$
$$(8)2 = 0000\ 1000$$

All more significant digits than 8 are a multiple of 8. $16 = 8 * 2; 32 = 8 * 4$ etc.

Also the digit with the same value as n is divisible by n and not relevant for the result of the modulo. That's how $8 \mod 8 = 0$.

The less significant digits, the digits 0 to x, where $n = 2^x$ are enough to calculate $a \mod n$. Those digits can represent the values $[0, ...n - 1]$. The same values as the modulo operation with module n can result.

That's why we can replace the algorithm from above by only looking at the n least significant digits. Therefore the bitwise logical AND operation for a and $n-1$ can be used. That way the result represents only the least significant digits.

For example $217 \mod 8$ can be calculated that way:

$$\begin{aligned}(217)2 &= 1101\ 1001\\ (8\text{-}1 = 7)2 &= 0000\ 0111\\ \& &\text{---------}\\ (1)2 &= 0000\ 0001\end{aligned}$$

With this principle the normal modulo operation can be replaced by one logical bitwise AND, if the module is a power of 2.

Division Operation The division $a \div d$ with a divisor $d = 2^x$ can be realized by shifting a x times to the right.

For example $217 \div 8$ can be calculated that way:

$$\begin{aligned}(217)2 &= 1101\ 1001\\ (217 \gg 3)2 &= 0001\ 1011\end{aligned}$$

Thus also the division can be realized much more efficient for powers of 2 than for other numbers.

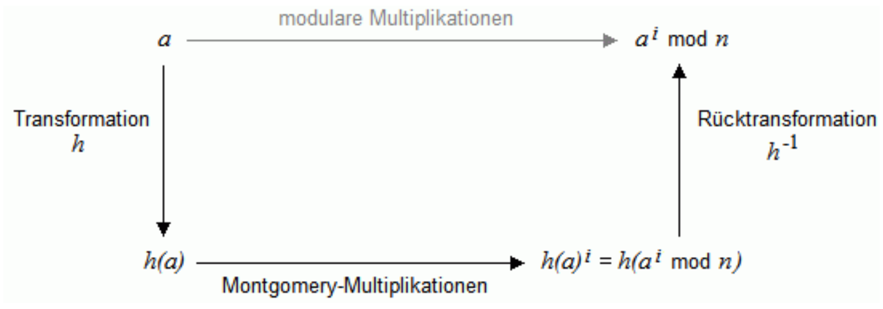


Figure 2.4: Concept of the Montgomery Exponentiation [Fleb]

Montgomery Exponentiation

The Montgomery Exponentiation uses this advantages of calculations with powers of 2 by reducing the m of the original calculation $m^e \mod n$ into the Montgomery representation, where instead of $m^e \mod n$, $h(m)^e \mod r$ can be calculated with $r = 2^x$. Montgomery algorithm itself only provides an efficient technique for modular multiplication. By using the in chapter 2.3.2 introduced technique of exponentiation by squaring we only have to transform m one time into Montgomery representation $h(m)$. After that we can square it by applying $\log(e)$ times the Montgomery multiplication resulting in $h(m^e \mod n)$. This number has to be re-transformed with h^{-1} into the primal representation. That gives us the final result $h^{-1}(h(m^e \mod n)) = m^e \mod n$. Graphic 2.4 schematically shows the difference between the Montgomery algorithm and the general procedure. Even with the additional transforming into and out of the Montgomery representation, the faster modulo and division operations lead to a better overall run time.

The Montgomery multiplication itself only calculates a modular multiplication not an exponentiation. That's why the following parts of the Montgomery Algorithm are explained for the calculation of $m * m \mod n$ and not for $m^e \mod n$. b is the base of the numerical system. Best chosen is the machine word size, or an other power of 2, the smallest $b = 2$. The algorithm only works, if n is odd, which is always the case if n is the product of two (odd) primes, as it is the case in RSA algorithm. The next steps will be explained with $7 * 7 \mod 13$ as an example.

First some parameters needed for the Montgomery algorithm have to be calculated. This step will be called *preprocessing* during this work. Therefore the numbers r, r^{-1} and n' have to be calculated from n [Fleb]. r is the next biggest number to n, with $r = b^x$. It can be calculated as $r = b^{\lceil log_b(n) \rceil}$. With $b = 2$ and $n = 13$ and r would be 16.

r^{-1} and n' have to satisfy $0 < r^{-1} < n$, $0 < n' < r$ and $r * r^{-1} - n * n' \equiv 1 \pmod{n}$. They can be calculated with the extended Euclidean algorithm r*s+n*t=1 [Sch06]. After this $r^{-1} = s \mod n$ and $n' = -t \mod r$.

For example calculating r^{-1} and n' for $n = 13$:

$$16 * (-4) + 13 * 5 = 1$$

$$r^{-1} = -44 \mod 13 = 9$$

$$n' = -5 \mod 16 = 11$$

Montgomery Representation

The Montgomery representation is a number transformed into Montgomery form by applying the function $h : N \to N$, $h(x) = (x*r) \mod n$ [Fleb]. For example to calculate $7*7 \mod 13$, the Montgomery representation of 7 would be $h(7) = (7*16) \mod 13 = 8$.

To get a number out of the Montgomery form the function h^{-1} is needed. $h^{-1}(y) = (y*r^{-1}) \mod n$. For example $h^{-1}(8) = (8*9) \mod 13 = 7$.

Montgomery Multiplication

The Montgomery multiplication or Montgomery reduction works as in algorithm 3 shown [Mon85]. Given n and former calculated n' and r are written in capital letters.

Algorithm 3 Montgomery Multiplication (REDC())

```
Input: $T = h(x) * h(y)$, $N$, $N'$, $R$
Output: $h(x * y \mod n)$
  $m \leftarrow (T \mod R) \times N' \mod R$(So m ¡ R)
  $t \leftarrow (T + m \times N) \div R$
  if $t > N$ then
    return $t - N$
  end if
  return $t$
```

First two numbers in Montgomery representation are multiplied. The product is $T < 2*N-2$. For the operation $7*7 \mod 13$ it would be:
$T = h(7)*h(7) = 8*8 = 64$
$m = (64 \mod 16)*11 \mod 16 = 0$
$t = (64 + 0*13)/R = 4$
Because $4 < N$ the return value is 4.

The return value is still in Montgomery form. To get the real result of $7*7 \mod 13$ we have to apply $h^{-1}(4) = (4*9) \mod 13 = 10$.

The advantage of this algorithm compared to the general method is that the modulo operation can be replaced by a bitwise logical AND and the division by bitshifts (see chapter 2.3.3 - Idea). This leads to better efficiency with increasing sizes of numbers.

Range of the Algorithm

Considering that a machine word has a size of 32 bits, the Montgomery algorithm introduced in Chapter 2.3.3 does not work for big numbers as used in RSA. For the function $m^e \mod n$ the parameters have the following restrictions.

The exponent e has to fit in 32 bits, so it can't be bigger than $2^32 - 1$. The calculation only works with a and n having together a maximal length of 30 bits. Additionally $n < 2^16$.

2.3.4 Montgomery Algorithm for Multiprecision Integers

In order for the Montgomery algorithm to be applicable to small machine words and large numbers (i.e. 32 bit machine words and 1024 bit integers), there is a variation of the algorithm described above for multiprecision integers [Mon85].

The following variant algorithm decomposes numbers of all possible number systems into their digits and calculates the modular exponentiation separately for each digit one after the other. For easier understanding, the algorithm is explained in the following example using decimal numbers. When using the algorithm for the calculation of multiprecision integers, it can be imagined that the multiprecision integers are split into small integers that are equivalent to single digits of a numeral system with the base fitting into an integer. With these integers a computer can calculate using single precision operations. After processing the algorithm a multiprecision integer results, which consists of many resulting integers. In this case, the basis of the algorithm is not, as in the examples, the 10, but at most the size of an integer, so that the algorithm can be calculated with single operations.

In parts the following Transformation and Multiplication of the variant Montgomery Algorithm work similar to algorithm 2.3.3 only with multiprecision operations. For more details the implementation in chapter 4.2 can be seen.

Montgomery Exponentiation for Multiprecision Integers

Same as done in chapter 2.3.3, the Algorithm only does a modular multiplication. For the modular exponentiation the algorithm can be combined with exponentiation by squaring (see chapter 2.3.2).

Montgomery Representation for Multiprecision Integers

Also here r and n' have to be calculated from the module n represented in the numeral system with base b [Mon85].

Therefore first the base b of the number system is chosen in which the calculation is performed. Due to the binary computer architecture a power of two, at best a multiple of the machine word should be chosen. As before it has to be $gcd(n, b) = 1$. This is fulfilled for RSA, because n is always odd and as base a power of two greater than one is chosen, which is always even. Here also an r is chosen which is larger than the multiprecision integer n and at the same time a power of the base b. It can be calculated by the formula $r = b^{x \geq \lceil log_b(n) \rceil}$.

For example we could chose base $b = 10$, the decimal numeral system. For a given module $n = 23$ we would pick $r = 10^{x \geq \lceil log_b(23) \rceil} = 10^3 = 30$.
The transformation into the Montgomery form works exactly like before. For the calculation $a * a \mod n$ with a, n multiprecision integers, $h(a) = a * r \mod n$. For all operations multiprecision operations are used.

To calculate $18 * 18 \mod 23$, for example, $h(18) = 18 * 30 \mod 23 = 540 \mod 23 = 11$. If this numbers are too large for a single precision operation they have to be performed with multiprecision arithmetic.

During Montgomery algorithm for multiprecision integers n' is calculated a bit different. It must be $n * n' \equiv -1 \pmod b$ and $n' \in [0, b-1]$. It will always remain smaller than the base, which means that it will always fit in one digit. It can be calculated with the extended euclidean algorithm $b * s + n * t = 1$ and b as the module. $n' = -t \mod b$.

For $n = 23$ and $b = 10$ the extended euclidean algorithm gives:

$$1 = 7 * 10 + 23 * (-3)$$
$$n' = 3 \mod 10 = 3$$

The value of r^{-1} can be calculated as before with the extended euclidean algorithm $r * s + n * t = 1$ and $r^{-1} = s \mod n$.

$$1 = 30 * 10 + 23 * (-13)$$
$$r^{-1} = 10 \mod 23 = 10$$

The re-transformation out of Montgomery form is performed by applying the function $h^{-1}(h(a)) = h(a) * r^{-1} \mod n = a$. For example $h^{-1}(11) = 11 * 10 \mod 23 = 110 \mod 23 = 18$.

Montgomery Multiplication for Multiprecision Integers

The input T of the new algorithm is same as before the product of $h(a)$ and $h(a)$, except that it is now a multiprecision integer. In the decimal example every digit can be seen as a single precision. T would be $h(108) * h(108) = 20 * 20 = 400$. T is represented by 3 digits, so it is 'multi precise'.

The output S of the algorithm represents the result of the operation in Montgomery form. $S = h(a*a \mod n)$. To get the result of $a*a \mod n$ the re-transformation out of the Montgomery form has to be performed. Therefore again apply $h^{-1}(S) = S * r^{-1} \mod n = a * a \mod n$.

The Montgomery algorithm for modular multiplication of two multiprecision integers and a multiprecision module n follows as algorithm 4 [Mon85].

In the algorithm, the variables b, n, n' and r are represented with capital letters. $X[i]$ refers to the $i - 1$th digit of the huge number X represented in the numeral system of base b. The multiprecision comparison $\geq$ is represented by $\geq$*.

For the implementation of the algorithm every digit can be stored as a value of an array. For example for an array of unsigned int the base 2^{32} would be the most efficient. Every value of the array would be used maximal, because digits from 0 to $2^{32} - 1$ are used, and numbers from 0 to $2^{32} - 1$ can be stored in one unsigned integer. The carry c is always 0 or 1 and could be stored as a flag instead of a bigger data type to safe storage. For numbers with 2n digits, their modular multiplication can be performed in n single precision operations.

2.4 The Raspberry Pi

The Raspberry Pi is an affordable miniature single-board computer designed to develop an understanding of computer architecture through testing, without fear of damaging much.

It is developed by the Cambridge-based nonprofit Raspberry Pi Foundation since 2012 and 100 percent manufactured in South Wales [Sev13]. The price of a Raspberry Pi varies according to model and features, but averages 30€ [Ras].

The computer originally developed for children has exceeded the expected interest even of experienced computer scientists due to the low price and the few restrictions. Already in 2017, after 5 years, the Raspberry Pi is regarded as the "world's third best-selling general purpose computer"[Mag17].

Because the developed implementation of RSA will run on the GPU there will be a really short introduction to the general architecture of the Raspberry Pi followed by a whole section only about the VideoCore IV GPU. Last the QPULib, the C++ library that is used to address the GPU is explained in detail for a better understanding of the implementation in chapter 4.2.

Algorithm 4 Multiprecision Montgomery Multiplication (Multiprecision REDC())

```
Input: Base $B = 2^x$ with x $\in N$
  $N$ with $gcd(b, N) = 1$ and represented in $p$ digits,
  $R = b^r$ with $r > p$ and represented in $r + 1$ digits,
  $N'$ with $0 \leq N' < B$,
  $T$ with $0 \leq T < R * N$ represented with $r + p$ digits
Output: $S$ with $0 \leq S < N$ and represented in $p$ digits
  $T[r + p] \leftarrow 0$
  for $0 \leq i < r$ do
    $c \leftarrow 0$
    $m \leftarrow T[i] * N' \mod B$
    for $0 \leq j < p$ do
      $x \leftarrow T[i + j] + m * N[j] + c$
      $T[i + j] \leftarrow x \mod B$
      $c \leftarrow \lfloor x/B \rfloor$
    end for
    for $p \leq j \leq r + p - i$ do
      $x \leftarrow T[i + j] + c$
      $T[i + j] \leftarrow x \mod B$
      $c \leftarrow \lfloor x/B \rfloor$
    end for
  end for
  for $0 \leq i \leq p$ do
    $S[i] \leftarrow T[i + r]$
  end for
  if $S \geq^* N$ then
    return $S - N$
  else
    return $S$
  end if
```

2.4.1 General Architecture

A nice overview about the hardware of all Raspberry Pi models can be found at [Ras19a], [Ras19b] or [Wik19]. Those are also used for the following information. Because for this work a Raspberry Pi 1 Model B+ was used, the following descriptions might only fit to this model.

Raspberry Pi 1 Model B+ includes the System on a chip (SoC) BCM2835 from Broadcom [Bro19a]. It's for this work important components are the following.

RAM

The Synchronous Dynamic Random Access Memory (SDRAM) has 512 MB and is shared by the CPU and the GPU[Ras19b]. The ratio is by default evenly distributed, but can be changed.

CPU

The CPU is a one core ARM processor, the ARM1176JZF-S with 700MHz. It is "the highest-performance single-core processor in the Classic Arm family"[Dev19]. For more details see [ARM19].

GPU

The GPU used is the VideoCore IV by Broadcom running with 250 MHz. Since the characteristics of the GPU are most important for the implementation, they are discussed in chapter 2.4.2 in detail.

2.4.2 The VideoCore IV GPU

The VideoCore IV graphics processing unit (GPU) by Broadcom is built in all versions of the Raspberry Pi so far[Wik19]. This has the nice effect that the implementation of RSA achieved during this thesis works on all Raspberry Pi models.

Available is a public Documentation of the VideoCore IV [Bro19b] and an unofficial one that was created before the official one was released [Her19].

In this chapter first the components important for the work are roughly explained, since they are not addressed directly, but indirectly with the help of the QPULib library.

Then follows a description of the actual arithmetic units, the Quad Processing Units (QPUs).

Components

As shown in Figure 2.5, the GPU consists of many components. However, only the QPUs (Quad Processing Unit) are freely programmable [Sta17]. These are addressed in this work with the used library QPULib.

The VideoCore IV can contain up to 4 slices with 4 QPUs each. The chip built into the Raspberry Pi consists of three slices, resulting in 12 QPUs [Sta17]. Each slice has several caches which are shared by all 4 QPUs. Each slice also has a Special Function Unit (SFU), which can be used to approximate functions as the square root ($sqrt(x)$), the binary logarithm ($log_2(x)$) as well as the exponential function based on two (2^x) and the

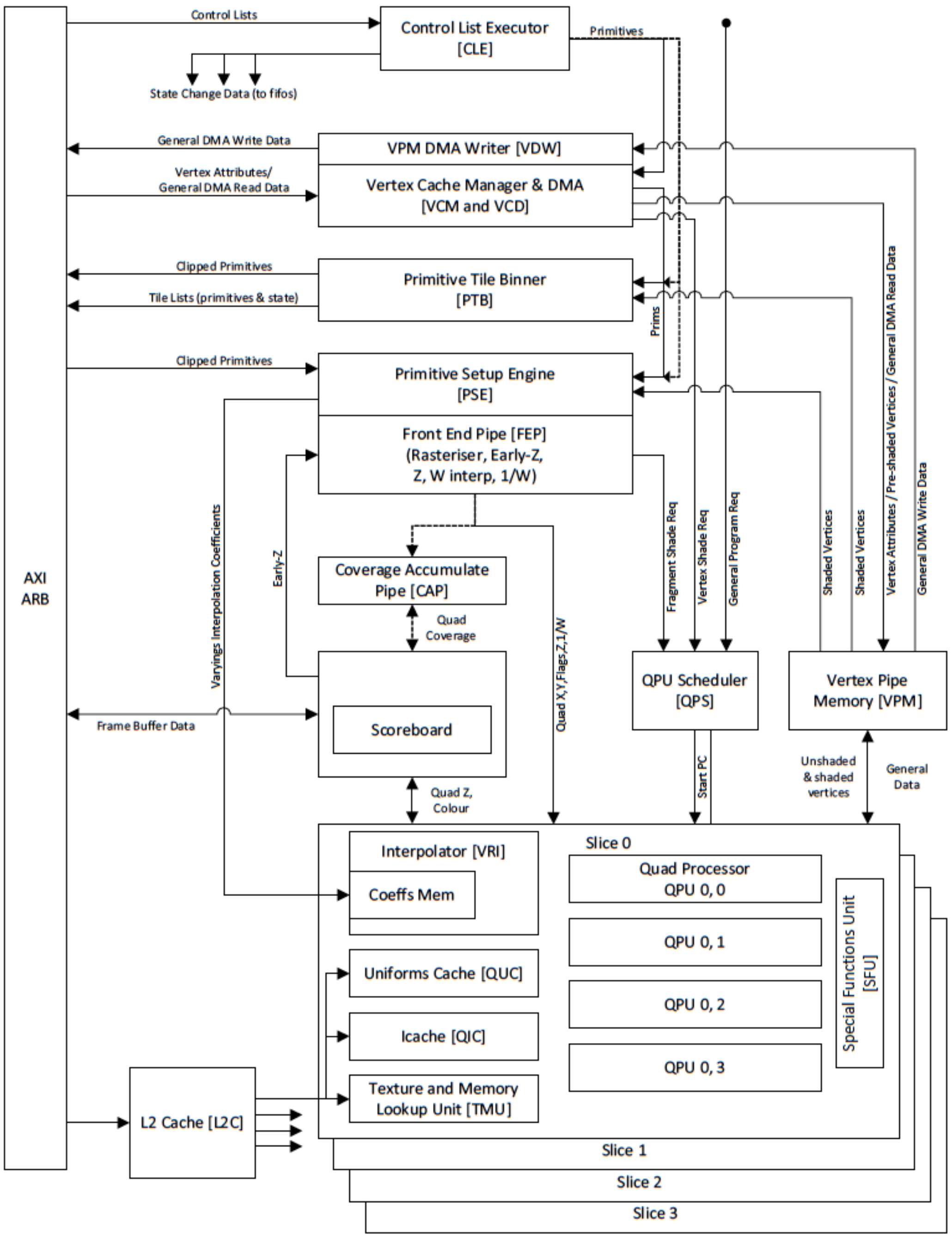


Figure 2.5: Architecture overview of the VideoCore IV[Bro19b]

multiplicative inverse ($1/x$) for floating point numbers [Bro19b]. They are not needed for RSA implementation. All slices have shared access to one L2 cache and one vertex pipe memory (VPM), which can be used to access the main memory, as the GPU does not have its own memory, but uses the same memory as the CPU. The percentage of memory that is made available to the GPU can be changed by the user [Sta17].

The Quad Processing Unit

The Quad Processing Units (QPU) are Vector processors and the programmable arithmetic units of the VideoCore IV. The vectors consist of 16 elements, of which four are always combined into a quad, a vector with four elements. One clock is needed to process a quad ("Four-way physical parallelism"), so four clocks are needed to process a vector ("16-way virtual parallelism"), by processing four Quads after each other [Bro19b]. The individual vector elements are either 32-bit integer or 32-bit floating point values [Nay16].

Each QPU has two register files A and B, each with 32 registers, of which 5 are accumulators that can be used as temporary memory, as they can be read and written directly one after the other. The registers again consist of 16 32-bit elements to enable SIMD on all 16 vector values [Sta17].

During execution, each element is processed by the SIMD processor, which has the same index as the element. In addition, the 16 vectors can be rotated so that each element can also be moved to all other indexes, always keeping its neighbors [Sta17].

The QPU has two asymmetric ALUs, one for additions and one for multiplications [Sta17]. The ALU for additions has 24 different operations, also including logical operations and shifting. The ALU for multiplications has only 8 different operations. Among other things there is no modulo operation and only a 24-bit multiplication [Bro19b]. The existing operations can be found in the VideoCore IV documentation, and chapter 2.4.3 lists which operations can be addressed with the QPULib. Although the entire GPU has a 32-bit architecture, the instruction set is 64 bits long [Bro19b].

Each QPU has a frequency of 250 MHz. Since 16 elements can be calculated on one QPU in four clock cycles and a total of 12 QPUs can be used, theoretically 4*12 values can be processed in one clock cycle. With a frequency of 250MHz this results in a theoretical throughput of 250,000*4*12 = 12,000,000 operations a second [Nay16].

2.4.3 The QPULib

The QPULib is a C++ library that runs on the CPU of the Raspberry Pi and addresses the GPU during runtime. It is published 2016 under the MIT License and is protected by the copyright of Mathew Naylor. [Nay16].

As described in chapter 2.4.2 all QPUs are working with 16-vector elements. This means that every instruction done on a QPU is executed on every vector element, which means on all 16 elements. If you add two of these 16-vectors, you get another 16-vector, where each vector element is the sum of the two summand elements at the same index (see figure 2.6).

How the QPULib basically is used is explained in the following sections. Everything is cited from the README.md of the QPULib [Nay16] or concluded from the library code.

Types and Pointers

Currently, the QPULib contains the data types `Int` and `Float`, each representing a 16-vector of 32-bit integers and a 16-vector of 32-bit floats. There is also a type `Bool`, that consists of a 16-vector of boolean values. The types `Ptr<Int>` and `Ptr<Float>` are each 16-vector filled with the addresses of each vector element of an Int or Float data type.

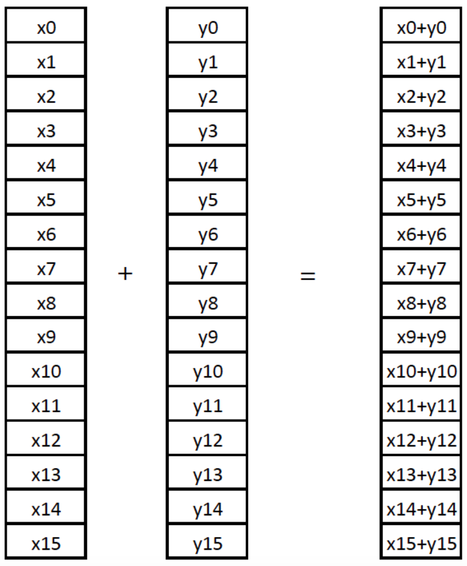


Figure 2.6: One operation applied on all 16 vector operands

In the following, all properties are explained only for the data type `Int`, since floats are not used in this work. In fact `Int` and `Float` work similar.

The value stored at the addresses in `Ptr<Int>` can be accessed using the `*` operator. If p is a `Ptr<Int>`, the `*p` command would load all 16 32-bit integers stored at the addresses in `p`, starting at the address in the first vector element of `p`. The command `*(p+1)` would also load 16 32-bit integers, where the first value would be the value stored at the address in the second vector element of `p`. The 16 value could be outside the allocated memory.

Operations

The QPU contains the following operations on datatype `Int`:

- `+` : Addition
- `-` : Subtraction
- `*` : 24-bit Multiplication
- `<<` : Left Shift
- `>>` : Right Shift
- `++` : Increment
- `min` : Min
- `max` : Max
- `&` : Bitwise AND
- `|` : Bitwise OR
- `^` : Bitwise XOR
- `~` : Bitwise NOT
- `shr` : Unsigned shift-right
- `ror` : Bitwise rotate-right

All those operations are binary with Int as operands, except Increment and Bitwise NOT, they have only one operand. It can be seen that neither the division nor the modulo operation is implemented.

Also the Int comparisons `==`, `!=`, `<`, `<=`, `>`, `>=` can be done. The resulting 16-vector of all operations is of type `Bool`.

On type `Boolean` there are following single and binary operations, that are useful for conditionals:

- `!` : logical NOT
- `&&` : logical AND
- `||` : logical OR
- `any` : returns true if any of the 16-vector elements is true
- `all` : returns true if all of the 16-vector elements are true

Only the logical AND and the logical OR are binary operations. All others have only one `Boolean` as parameter.

Further the QPULib provides the functions `me()`, `index()` and `numQPUs()`.

`me()` that returns a 16-vector with every vector element containing the unique id of the respective QPU. Those ids start it 0 and go up to the number of active QPUs - 1. This is useful to ensure that each QPU processes different data in one array.

The call `index()` returns a 16-vector whose elements contain the indices of the particular vector elements. So this vector always contains all numbers from 0 to 15. This function is needed for precise memory accesses (see 2.4.3).

The function `numQPUs()` returns a 16-vector containing the number of active QPUs in every element. These can have a value from 0 up to 12. This function is also useful in iterating over one Array shared by all QPUs, to to ensure that each QPU processes different data.

Invoking the QPUs

To execute the functions implemented with the QPULib, e.g. `foo(Ptr<Int>, Ptr<Int>)`, on the GPU, `auto kernel = compile(foo);` must be called. With the resulting kernel of type `Kernel<Ptr<Int>, Ptr<Int>>` and matching parameters, the function can be executed.

The data to be passed as parameters to the QPU must be stored by the CPU in so-called `SharedArray<int>` or `SharedArray<float>`. These pointers can then be passed to the kernel as parameters. The QPULib then converts the `SharedArray<int>` into the GPU data types `Ptr<Int>`.

Before calling the kernel and the actual calculation on the GPU you can set the number of used QPUs to a number from 1 to 12 with the call `kernel.setNumQPUs(n)`. Otherwise only one QPU is used by default.

To finally execute the function `foo()` with `SharedArray<int> x` and `SharedArray<int> y` as parameter, `kernel(&x, &y);` must be called on the CPU.

Conditionals and Control Flow Instructions

The QPULib has an implementation of `For` and `While`. The `For` loop is structured as follows:

```
For (Int i = 0, i < n, i = i+inc)
    ...
End
```

The difference to usual c++ for loops is, that `i` is not a integer but a `Int` datatype with all elements containing the same value at every iteration.

The `While` is structured as follows with the variable `b` of type `Boolean`:

```
While (b)
    ...
End
```

Instead of an if statement the QPULib provides the Statement `Where`. It is structured as follows, again `b` has to be of type `Boolean`:

```
Where (b)
    op1
    op2
    ...
End
```

This statement means that the operations `op1`, `op2`, ... are performed on all vector elements with the same index as where b is true and all other elements stay the same.

Attention to the fact that `While` and `For` loops can be nested inside each other. These `While` and `For` loops may also contain `Where` conditions. But it is not allowed that loops are executed within a `Where` statement. This is not possible due to the single instruction multiple data principle of the GPU, because otherwise only selected data get special instructions and not all.

Memory Access

Since conservative loading and storing data with pointers takes a long time, the QPULib has additionally implemented non-blocking loads and stores. These work via the 4-element FIFO of each QPU where data can be stored and retrieved later in less time. The command `gather(p)` is used to load data into the queue. `p` is a `Ptr<Int>`, but it must point to the exact address of the desired integer and not only to the start address `a` of `a` 16-vector. To do this `Ptr<Int> p = x + index();` can be used so that the exact address for each vector element is calculated.

To load the data from the queue, `receive(x);` is called, where `x` must be of type `Int`. This `receive(x)` loads the data in the first position of the queue into the variable `x`.

Since the queue is 4 elements long, a maximum of 4 gathers can be executed until a receive must be called. Also the Queue has to be emptied before the QPULib function terminates.

The function `store(p)` allows non-blocking stores to the address `p`. Where `p` is again a `Ptr<Int>` which contains the exact address for each vector element, and e.g. can be defined with `Ptr<Int> p = x + index();`.

Function Calls

As in common programming languages, a QPULib function can call subfunctions if it passes the appropriate parameters. It should be noted, however, that Single Instruction, Multiple data principle applies. For this reason, no function call can be made within a Where statement.

Emulation Mode and Debug Mode

The QPULib can be used in emulation mode on other devices, where parallel processing is only simulated on the CPU. This emulation mode can be executed by not running the program with the flag QPU=1. It is possible to set the number not equal to 1 or to drop the flag.

Debug mode can only be active in combination with emulation mode. If debug is true all operations, including NoOps because of blocking operations, that are executed on the QPUs are printed.

2.5 Related work

Due to the frequent and sometimes very CPU-intensive use of cryptographic algorithms, many attempts already exist to execute cryptography on the GPU although it is actually not designed for this purpose. Because RSA is so widely used, many of these attempts deal with the execution of RSA on the GPU.

For example, H. Fadhil and M. Younis have compared a sequential RSA calculation with a multithread RSA calculation on the CPU and an execution on a GPU [FY14]. They also use the Montgomery algorithm for the modular exponentiation and come to the conclusion that starting from a key length of 1024 bits the GPU implementation always enables the fastest calculation.

Since RSA is very slow as an asymmetric encryption method, but is still used very often, many people are still researching new approaches to speed up RSA. Besides such practical work on the parallelization of RSA, S. Saxena and B. Kapoor's paper presents various approaches to RSA parallelization at a glance, which do not all use the Montgomery algorithm, but explain different variations of software and hardware implementations for an acceleration of the execution time of RSA [SK15].

In addition to the many parallel RSA implementations on different multicore CPUs and GPUs there are also papers dealing with the usage of the VideoCore IV. None of them deals with the computation of RSA, but with the difficulties arising by programming the VideoCore IV which is not intended for executing normal code.

For example, P. Pauls has offloaded AES to VideoCore IV by programming it with assembler. It is therefore possible to use the GPU of the Raspberry Pi to perform some encryption procedures. It does not achieve any speed advantage compared to the sequential AES implementation of OpenSSL yet, but gives an insight into the structure of the VideoCore IV and its difficulties [Pau17].

Based on this work, Y. Rixen uses the QPULib presented in chapter 2.4.3 to program the VideoCore IV for AES and achieve a better performance [Rix19]. Therefore he extends the open source QPULib with the data type `char` and, with an invocation of 12 QPUs in parallel, he reaches a nearly as fast result as the OpenSSL AES algorithm, which is executed on the CPU. Thus not only the possibility, but also an timely advantage of the execution of some cryptographic algorithms on the VideoCore IV has prospect for success.

As an alternative to programming the GPU using assembler or the QPULib, D. Stadelmann has developed an Open-CL standard implementation that allows the execution of Open-CL C code on the VideoCore IV [Sta17]. The library is not yet complete, but already offers some features. Also a deep insight into the structure of the GPUs as well as the difficulties and bottlenecks to avoid is given.

2.6 Summary

In summary, this chapter first describes how the RSA cryptosystem works. In this thesis, the encryption and decryption functionality will be implemented on the GPU, while the key generation will not be programmed, but the RSA encryption will read standard keys generated by OpenSSL.

In addition, it has been shown that RSA encryption in both directions, encryption and decryption, is based on modular exponentiation. The GPU implementation must therefore perform a modular exponentiation. Because the RSA keys consist of numbers up to 1024 bits in length, the modular exponentiation must be executable on multiprecision integers.

In order to carry out this very costly and time-consuming modular exponentiation, three techniques were presented to simplify the calculation.

One is the modular multiplication which proves that not first the whole exponentiation and then the modulo calculation has to be done. The calculation can instead be performed by many multiplications followed directly by modulo operations. Thus the numbers do not exceed at any time of the calculation a size of 2 times the size of the module m minus 2. This saves memory space as well as calculation time of the multiprecision multiplications, because all numbers are much shorter.

Second, it was shown that the required exponentiation can be shortened with exponentiation by squaring to logarithmically many multiplications. This reduces the calculation time immensely.

Third, it was shown that the single modular multiplications can be accelerated by using the Montgomery algorithm. Thereby the runtime expensive division and modulo operation are replaced by fast bitwise AND and bitshifts operations. Again, the calculation time is accelerated by this. The Montgomery algorithm can also be executed with multiprecision integers, which makes it perfect for the GPU implementation of RSA.

In the last part of this chapter the Raspberry Pi was introduced, whereby the functionality of its GPU, the VideoCore IV, is in the foreground. Together with the following introduction into the programming of the GPU using the QPULib, a solid basis for the implementation of RSA encryption on the GPU of the Raspberry Pis has been created.

3 Parallel RSA Algorithm

In the previous chapter the basics for an RSA implementation on the GPU of the Raspberry Pi were laid. The functionality of RSA encryption and methods for simplifying and accelerating modular exponentiation were explained. By combining these methods at the beginning of this chapter, a sequential structure of an RSA implementation is developed. After this sequential design of an RSA encryption has been found, different possibilities are considered where and how the RSA encryption could be offloaded to the GPU in parallel. In the previous chapter the properties and limitations of both, the VideoCore IV and the QPULib were shown. The developed approaches for parallelization of RSA will be compared with these and finally one parallel algorithm will be chosen, which will be partly implemented on the GPU with the help of the QPULib.

3.1 Sequential Algorithm

As a first step to find an efficient parallel algorithm it will be designed sequentially. Since RSA is only based on modular exponentiation, the concepts presented in Chapter 2.3 were combined to create an efficient sequential algorithm.

The exponentiation by squaring from chapter 2.3.2 is performed as in algorithm 1, with the variables b and res as multiprecision integers and the multiplications are replaced by modular multiplications. Because of the residue multiplications from chapter 2.3.1 the execution of modular multiplications is allowed, and the result of the exponentiation by squaring will be $m^e \mod n$. For the modular multiplication the Montgomery multiplication for multiprecision integers introduced in chapter 2.3.4 is used. The resulting modular exponentiation by squaring is shown in algorithm 5. The operations on multiprecision integers are all multiprecision operations. The Montgomery multiplication for multiprecision integers is written as $mpMontMul(a, a, n)$ for the calculation of $a * a \mod n$ and works exactly like the multiprecision Montgomery multiplication in algorithm 4.

The calculation of the parameters r, n' and r^{-1}, the transformation into Montgomery form and out of Montgomery form are performed for multiprecision integers.

It is also possible for RSA to divide a large message M into small messages m_{i} and to encrypt and decrypt these small messages. Otherwise the size of encryptable messages would be limited, because m must always be smaller than the module n. Because the same module n is used for all messages m_{i} that are encrypted or decrypted with the same key, the parameters r, n' and r^{-1} can still be calculated only one time for all of them. Because this has to be done before starting the actual RSA encryption it will be referenced at as $preprocess(n)$.

Because Exponentiation is the repeated multiplication of the same value, for each part m_{i} of the whole message M that will be encrypted or decrypted, it is enough to transform it into Montgomery form only one time, before starting the exponentiation by squaring. The Exponentiation itself only uses the message m_{i} in Montgomery form. This procedure is refered to by $transform(m, r, n)$

Algorithm 5 Montgomery Exponentiation by Squaring

```
Input: mp integer in Montgomery form $h(m) > 0$, exponent $e \geq 0$, mp integer module
  $n > 0$ where $n$ odd, mp integer r, mp integer n'
Output: $h(m)^e \mod n = res$
  $res \leftarrow 1$
  while $e \neq 0$ do
    if $e \mod 2 == 1$ then
      $res \leftarrow mpMontMul(h(m) * res, n, r, n')$
    end if
    $m \leftarrow mpMontMul(h(m) * h(m), n, r, n')$
    $e \leftarrow e >> 1$
  end while
  return $res$
```

The result of a modular exponentiation can be re-transformed out of Montgomery form after the Exponentiation is done. This is done as under the action $retransform(h(m^e \mod n), r^{-1}, n)$.

All this combined leads to the following algorithm 6. With this a full modular exponentiation on multiprecision integer can be performed, which is equal to a RSA encryption.

Algorithm 6 Sequential RSA Algorithm

```
Input: mp integer $m < n$, exponent $e \geq 0$, mp integer module $n > 0$ where $n$ odd
Output: $m^e \mod n = res$
  $r, n', r^{-1} \leftarrow preprocess(n)$
  $h(m) \leftarrow transform(m, r, n)$
  while $e \neq 0$ do
    if $e \mod 2 == 1$ then
      $hres \leftarrow mpMontMul(h(m) * hres, n, r, n')$
    end if
    $m \leftarrow mpMontMul(h(m) * h(m), n, r, n')$
    $e \leftarrow e >> 1$
  end while
  $res \leftarrow retransform(hres, r^{-1}, n)$
  return $res$
```

The following diagram 3.1 provides a more abstract overview of the process design. The function $ModExpSquare()$ refers to algorithm 5, the modular exponentiation by squaring.

3.2 Approaches For Parallelization

For this work Montgomery multiplication combined with exponentiation by squaring was chosen to perform modular exponentiation. There are four kinds of possibilities to parallelize this way of modular exponentiation [Pea96]. All possibilities will be introduced during this section. In the following section they will be discussed regarding their implementation possibilities on the GPU.

In order to simplify the referencing of the different possibilities, this section labels each

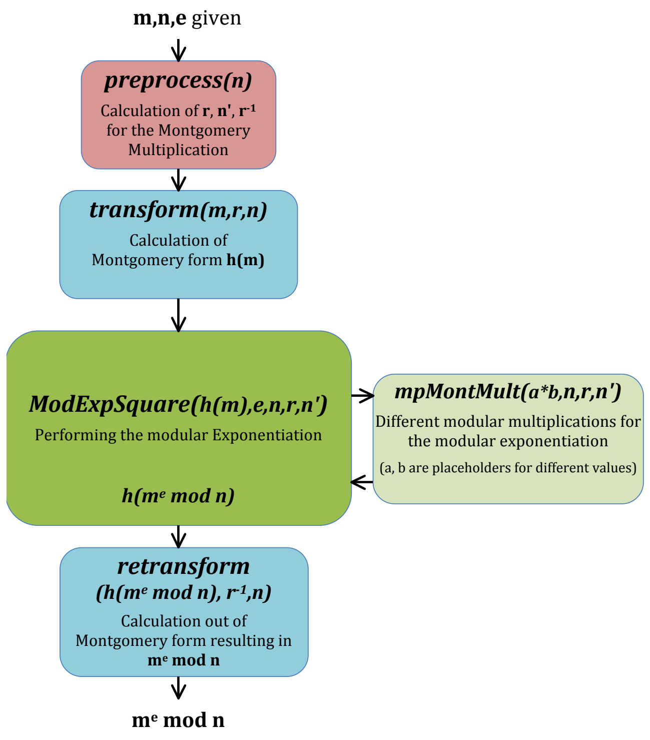

m,n,e given
preprocess(n)
Calculation of r, n', r-1 for the Montgomery Multiplication
transform(m,r,n)
Calculation of Montgomery form h(m)
ModExpSquare(h(m),e,n,r,n')
Performing the modular Exponentiation
h(me mod n)
mpMontMult(a*b,n,r,n')
Different modular multiplications for the modular exponentiation
(a, b are placeholders for different values)
retransform (h(me mod n), r-1,n)
Calculation out of Montgomery form resulting in me mod n
me mod n

Figure 3.1: Process of the full Modular Exponentiation Algorithm

parallelization possibility from P1 to P4, with P1 being divided into three versions: P1a, P1b and P1c. These labels will be referenced to in the next section as well.

3.2.1 Types of Parallelization options

In general, the possibilities of parallelizing RSA can be summarized in four groups. These are briefly introduced now. In the following section they will be explained exactly in relation to the design of the sequential algorithm.

P1 Parallel Message encryption

Normally a big message M would be partitioned into several parts $,_i$, converted into a number and then modular exponentiation would be applied on these parts sequentially. One possibility of using the parallel computational power of the GPU would be to partition the full message as before, but apply modular exponentiation parallel instead of sequential on all parts m_i. This should not decrease the time taken for one modular exponentiation significantly, but would increase the overall efficiency of encrypting a message. Since there are different possibilities of processing parallel message encryption, depending on what parts of the RSA calculation are performed parallel, they are divided in versions P1a, P1b and P1c and explained in section 3.2.2.

P2 Parallel Exponentiation by Squaring

For the modular exponentiation exponentiation by squaring is used. This kind of exponentiation can be done in parallel, by dividing the exponent in parts, performing the exponentiation with those small parts as exponents and afterwards merging all results together to a final result [LBPN12]. If the looping is performed from the least to the most significant bit, as described before, the exponentiation can be parallelized saving up to 33% [Pea96].

P3 Half-Size Modular Exponentiation for Decryption

The Decryption can be divided into two half-size modular multiplications, if p and q, and not only n and d are given in the private key. Then the modular multiplication can be performed parallel for the modules p and q instead of n. The parallel computed results can be combined with the Chinese Remainder Theorem to the full result. Those two half-size modular exponentiations can be four times faster than one full modular exponentiation.

P4 Parallel Multiprecision Multiplication

Part of the exponentiation by squaring is the Montgomery multiplication for multiprecision integers (2.3.4). This consists of some multiprecision operations, for example the complex multiprecision multiplication.

A way to parallelize the modular exponentiation in parts, is to modify the modular multiplication, so that the multiprecision multiplication is performed parallel. Normally the multiprecision multiplication has a complexity of $O(n) = n^2$. But it can be partitioned and executed parallel to reach a better performance if the communication between the processing units is faster than the multiplication time itself [Pea96]. For example S. Baktir and E. Savas reached up to 39% better performance by parallelizing the multiprecision Montgomery multiplication [BS13].

3.2.2 Parallelization Options for the sequential Algorithm

The figure 3.2 gives an overview of the in section 3.1 found sequential structure of the algorithm in combination with the introduced possibilities of parallelization. The different techniques of parallelizing the algorithm are labeled as before from P1 to P4. After the overview, each possibility is described in detail, whereby it is also discussed how the previous sequential algorithm would change exactly for this way of executing RSA on the GPU. All black lines stand for sequential processing. The colored ones for parallel computations. The different colors help to differ each parallelization approach and at which step of the algorithm they start. The purple letters are used for algorithm steps, that are only needed for P3.

Of course, some of the separately drawn possibilities could also be combined.

P1 Parallel Message encryption

P1a, P1b and P1c stand for the parallel processing of several messages simultaneously. As explained in the following, the three approaches differ at which point the algorithm is parallelized.

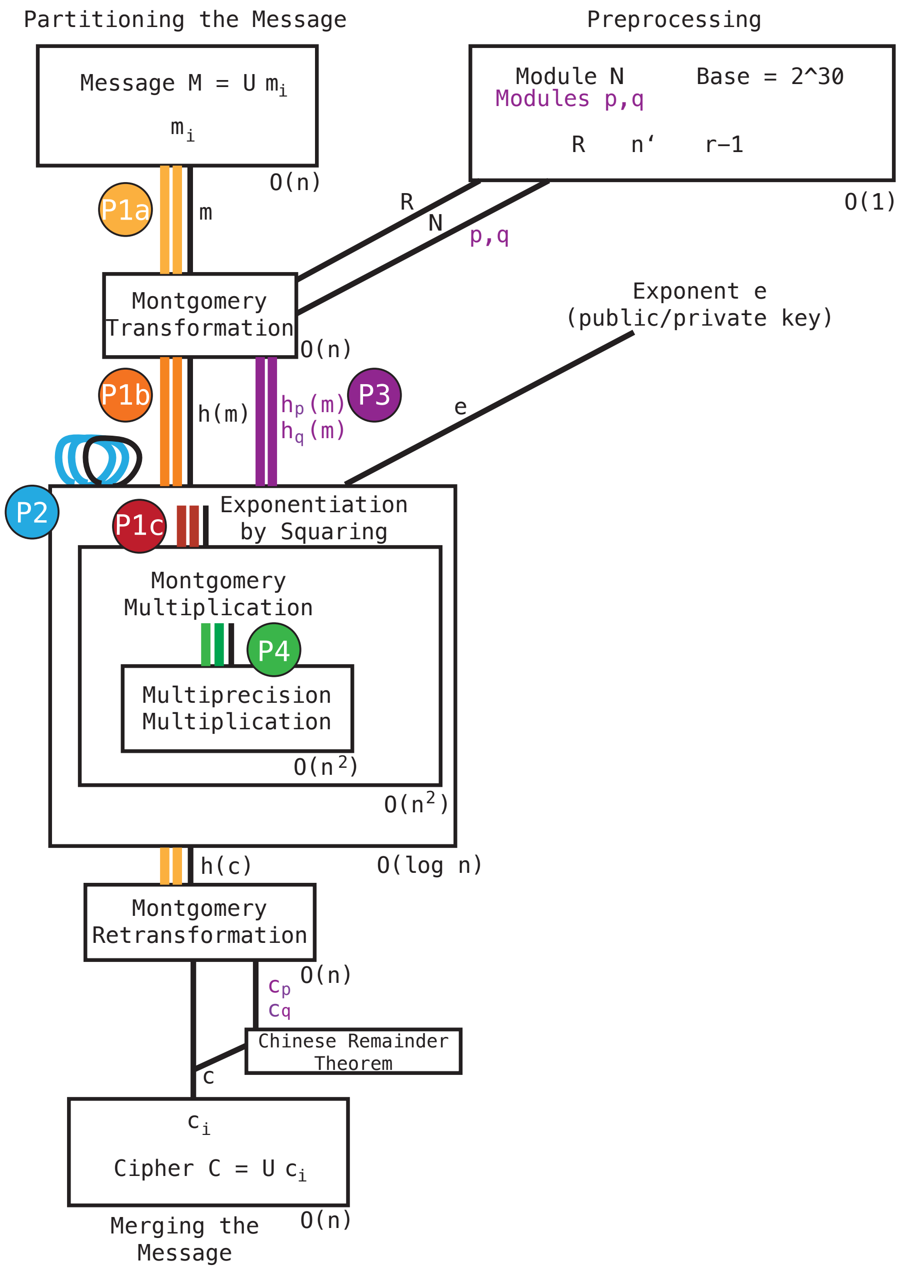


Figure 3.2: Process of the full Modular Exponentiation Algorithm with different parallelization steps

P1a: The yellow lines stand for parallel processing directly after the partition of the message M into submessages m_{i}. This means that the transformation into Montgomery form is already performed on all different submessages according to the single instruction multiple data principle on the GPU. Similarly, the entire Montgomery exponentiation including all Montgomery multiplications is executed in parallel with different data $h(m)$, as well as the re-transformation from out of Montgomery form are executed on the GPU.

Only then the results c_{i} are merged again by the CPU into a complete cipher C.

There is also the option to only offload the transformation to and out of the Montgomery form to the GPU. (Both because exactly the same operation is performed only with different values). But since these are not the most costly operations in the algorithm, this is not listed as an option.

P1b: The technique that is described by the orange lines differs from P1a by the fact that the transformation into the Montgomery form is calculated sequentially for all partial messages mi on the CPU and then the $h(m_{\mathrm{i}})$ is given to the GPU and the entire exponentiation is then executed parallel on the GPU. The re-transformation from the Montgomery form is then again executed sequentially with all data on the CPU. This means that only the Montgomery exponentiation is transferred to the GPU.

P1c: The red lines stand for the version to convert the single message pieces on the CPU sequentially into the Montgomery form, and then also to execute the exponentiation by squaring on the CPU. This is possible because the exponent for all partial messages m_{i} of a message M is the same. The individual Montgomery multiplications contained therein can then be executed on the GPU in parallel for all temporary products. The division of the exponentiation by squaring as well as the re-transformation from the Montgomery form and the composition of the partial ciphers c_{i} into the final cipher C are again all executed on the CPU.

P2 Parallel Exponentiation by Squaring

The blue lines stand for the parallelization of the exponentiation by squaring. Only one message part m_{i} of M is transformed sequentially into Montgomery form at a time on the CPU. Then the exponent is divided into different parts and the message $h(m)$ and the different exponent parts are transferred to the GPU. The GPU calculates the Montgomery exponentiation for $h(m)$ in parallel with the different exponent parts instead of the exponent e. Then the results of all on the GPU parallel performed Montgomery exponentiations are merged sequentially on the CPU again and the algorithm is completed sequentially.

P3 Half-Size Modular Exponentiation for Decryption

The purple lines represent the method to parallelize the Montgomery exponentiation by doing this real parallel with the two prime numbers p and q as module instead of the double sized n and combining the two results with the Chinese Remainder Theorem. The prime numbers p and q are the two numbers from which the private key d is calculated. This method therefore only works for decryption and not for encryption.

Therefore at first step $preprocessing()$ has to be performed for p and q instead of n. Then a message m_{i} is transformed sequentially on the CPU into Montgomery form for module p and for module q and then the Montgomery exponentiation for the messages $h_{\mathrm{p}}(m_{\mathrm{i}})$ and $h_{\mathrm{q}}(m_{\mathrm{i}})$ are executed in parallel on the GPU. The given exponent stays the same. Without combining this parallelization with others, as for example parallel exponentiation by Squaring, at this moment only two values of the possible 192 would be calculated at one time on the GPU. The re-transformation out of the Montgomery forms and the composition of the results are then executed sequentially on the CPU again.

P4 Parallel Multiprecision Multiplication

The green lines describe the method of accelerating the performance of the algorithm through parallel execution of the most expensive multiprecision operation, the multiprecision multiplication.

For this algorithm one message part m_{i} is processed at a time. Therefore all steps are executed sequential on the CPU up to the Montgomery multiplication. Only the multiprecision multiplication, that's part of it is offloaded to the GPU and executed parallel. All other parts of the Montgomery multiplication and all the following steps, like the re-transformation, are again calculated on the CPU.

3.3 Parallelization Techniques with regard on the Limitations of the GPU and QPULib

Because the parallel algorithm that is developed during this thesis is executed on the VideoCore IV with help of the QPULib (see chapter 2.4) the parallel algorithm has to be specially adapted to the possibilities and limitations of the QPULib and its GPU.

The previous section presented all possibilities how the parallel execution power of any GPU could theoretically be used to execute RSA. Now it has to be checked, which options can be executed on the VideoCore IV specially.

In the following, first the difficulties that go along with the VideoCore IV and the QPULib that were already partly mentioned in the background are shown. Afterwards, the different parallel possibilities presented in chapter 3.2.2 are considered together with these limitations, to arrive at the parallel implementations that are actually possible on the VideoCore IV. From those one algorithm will be chosen.

3.3.1 Limitations

In general, the bottleneck of the GPU is the transfer of data between the GPU and the RAM [Sta17]. Therefore it is important to reduce memory access, when developing the parallel algorithm.

An obvious but existing limitation of the QPULib is that it is addressed differently than in usual c. For this reason, no functions from existing libraries can be used to run on the GPU. Because of this all multiprecision operations that have to be developed on the GPU, have to be implemented especially for the QPULib.

In addition, the QPULib does not offer the data type `long` because it only has 32-bit ALUs. There is also no implementation of `unsigned int`. So all operations have to be executed on 31 bit maximum, to prevent an overflow.

Also the VideoCore IV nor the QPULib offer a modulo operation or division operation. These can therefore not be used in the parallel algorithm.

In the progress of this work, it turned out that the number of memory accesses in a GPU kernel invocation is limited by the QPULib. For this reason no arbitrarily long code can be executed in one kernel call on the GPU.

3.3.2 Parallelization possibilities with the QPULib

The variants presented in chapter 3.2.2 on how to parallelize modular exponentiation for RSA are not all applicable to QPULib and VideoCore IV. In the following, each variant is discussed with the former explained limitations and it is decided whether an implementation is possible and which one will be implemented.

P1 Parallel Message encryption

For this parallelization option again all three versions will be discussed separately.

P1a: To process the different variants parallel from the transformation onwards (yellow lines), a modulo operation with a module n is required for the transformation into the Montgomery form. Since n is a prime number, and therefore in any case no power of 2, a Division would be required for the modulo calculation on the GPU. The VideoCore IV provides no Division instruction and the QPULib has no division implemented neither. So the transformation cannot be implemented on the GPU. In addition, the transformation in Montgomery form and the entire exponentiation by squaring would require more memory accesses than the QPULib allows.

P1b: Processing many messages parallel during the whole exponentiation by squaring (orange lines) would also require more memory accesses than the QPULib allows.

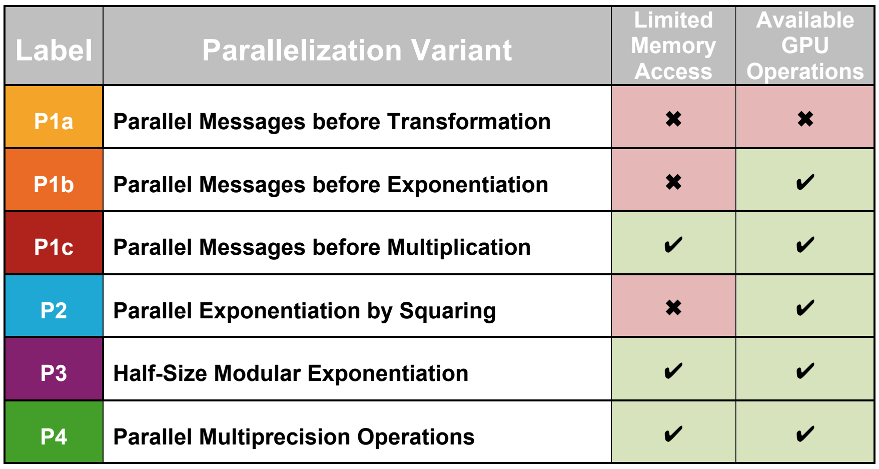
P1c: Only executing the Montgomery multiplication on the GPU (red lines) is possible considering the number of memory accesses. In addition, all required operations are available on the GPU.

P2 Parallel Exponentiation by Squaring

To parallelize the exponentiation by squaring on the GPU more memory accesses would be needed than are possible with the QPULib. That's why, although all needed operations are provided by the QPULib this technique can't be realised.

P3 Half-Size Modular Exponentiation for Decryption

To perform both Half-Size modular exponentiations with p and q instead of n as module in parallel, as it is shown in figure 3.2, again more memory accesses would be needed, than are possible. To only perform the Montgomery multiplications in parallel and the exponentiation on the CPU as described by the red line is would be possible, but this can only be implemented for the decryption, where the private key provides the primes p and q beside d.



Label	Parallelization Variant	Limited Memory Access	Available GPU Operations
P1a	Parallel Messages before Transformation	✖	✖
P1b	Parallel Messages before Exponentiation	✖	✔
P1c	Parallel Messages before Multiplication	✔	✔
P2	Parallel Exponentiation by Squaring	✖	✔
P3	Half-Size Modular Exponentiation	✔	✔
P4	Parallel Multiprecision Operations	✔	✔

Figure 3.3: Overview of Parallelization Possibilities and Limitations

P4 Parallel Multiprecision Multiplication

The parallel calculation of the multiprecision operations, especially the complex multiprecision multiplication, can be realized with the VideoCore IV using the QPULib.

Conclusion

Figure 3.3 gives an overview about the requirements of the QPULib for each Parallelization possibility and their applicability.

For the reasons given above, three of the six possibilities can be implemented with the current state of the QPULib.

Out of these three the parallel multiplication of 1024 bit integers leads to nearly no time advantages compared to the parallel multiplication of 2048 Bit or 3072 Bit integers. Sometimes they result in even worse execution times as a sequential algorithm [BS13]. But since this implementation tests only 1024 bit keys due to the small memory of the GPU, and additionally tries to achieve an acceleration of the execution time, in this thesis does not implement P4, a parallel multiprecision multiplication.

Because P3, the half-size modular exponentiation, can only be used in some cases of the decryption, that means in less than 50% of the RSA cryptosystem usage, it was favorable to implement the Parallelization option P1c, the offloading of the Montgomery multiplication with different messages onto the GPU. Because P3, the half-size modular exponentiation, is only executable on the VideoCore IV in combination with P1c, it is left as a additional project based on this thesis.

3.4 Parallel Algorithm

The different approaches for parallelizing RSA encryption were all considered for the implementation of the GPU executed RSA, depending on their advantages and possibilities to be combined with the QPULib and the VideoCore IV. As explained in chapter 3.3, there are only three approaches that can be implemented. The most promising results are expected from encrypting different messages in parallel (P1c), in comparison to performing a parallel

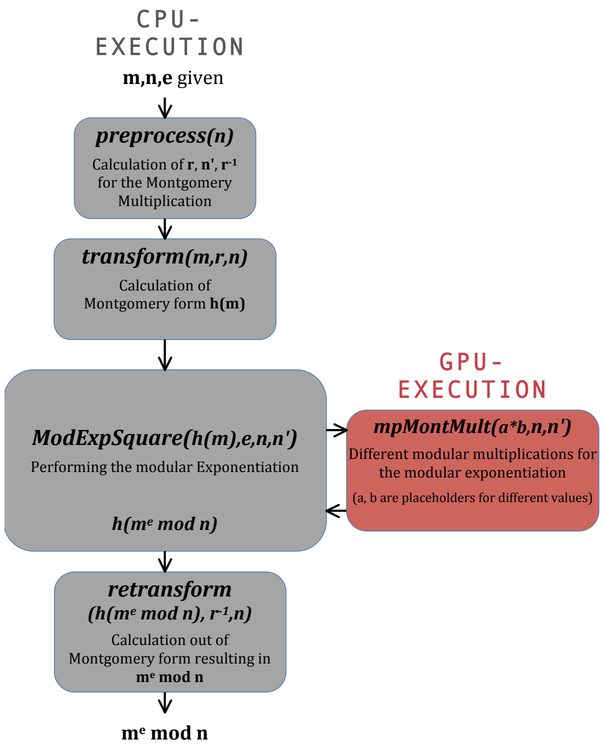

CPU-EXECUTION
m,n,e given
preprocess(n)
Calculation of r, n', r⁻¹ for the Montgomery Multiplication
transform(m,r,n)
Calculation of Montgomery form h(m)
ModExpSquare(h(m),e,n,n')
Performing the modular Exponentiation
h(mᵉ mod n)
GPU-EXECUTION
mpMontMult(a*b,n,n')
Different modular multiplications for the modular exponentiation
(a, b are placeholders for different values)
retransform (h(mᵉ mod n), r⁻¹,n)
Calculation out of Montgomery form resulting in mᵉ mod n
mᵉ mod n

Figure 3.4: Parallel Modular Exponentiation Algorithm on CPU and GPU

multiprecision multiplication on key sizes of 1024 bit or implementing only a decryption algorithm.

The final design of this chosen algorithm is now presented in this chapter.

For the reasons listed in Chapter 3.3, we have decided to implement an algorithm that processes different messages of 128 Byte at once. Because all those messages are encrypted with the same key the preprocessing for the Montgomery transformation has to be done only once at the beginning. It will be performed on the CPU, because its execution time is not important in comparison to the modular exponentiation following and was not examined for parallelization potential. After this all 128 Byte messages are transformed into Montgomery form sequentially, because the modulo operation needed can not be performed on the GPU. The exponentiation by squaring is also performed on the CPU, because its execution would need too many memory accesses than are possible with the QPULib. Because the Key for all messages is the same, they can all be encrypted inside the same exponentiation by squaring. For the Montgomery multiplications, that are done in every iteration of the exponentiation by squaring now all messages in Montgomery form $h(m_\mathrm{i})$ are performed real parallel on the GPU. With 12 QPUs this would be 192 $h(m_\mathrm{i})$ at the same time. After the exponentiation by squaring is done the re-transformation is again processed sequentially on all resulting ciphers in Montgomery form $h(c_\mathrm{i})$, because the modulo operation here neither can be accomplished with the GPU.

The figure 3.4 gives an overview of the structure of the algorithm chosen. The grey color steps are executed on the CPU and the orange one on the GPU.

4 Implementation of RSA

Previously, we developed a technique to efficiently calculate RSA and chose a kind of parallelization that will be implemented on the VideoCore IV. The selection of the implemented algorithm design was explained and presented in chapter 3.4. This chapter presents the final implementation of RSA encryption.

In the following, the on the CPU implemented part is presented first. It contains all sequential concepts that are used, especially the storage of data in the memory shared by CPU and GPU. This is followed by the implementation on the GPU, which was implemented using the QPULib. Thereby all parallel executed functions are presented. Finally, special problems during the implementation are mentioned.

4.1 Implementation on the CPU

In this implementation the CPU is used for more than only passing the given data to the GPU. It takes over a lot of the algorithms processing.

In the following the implementation of all algorithm steps that are processed on the CPU are shown chronological. Along the way the special concepts that were used to enable the calculations and achieve a even better performance are explained.

4.1.1 Input

The initial input of the implementation are the keys n and e (or n and d) as `char*` and the message M as `char[][]`. The message M is a two-dimensional array of the format x times y, containing x submessages m_{i} of y `char` length. All `char*` contain the numerical value of the key or message as decimal string. For example `char *n = ”0”;` represents the decimal number 0 and not the binary value, the number 48. Further the number of used QPUs could be changed by a `#define` but is set to 12. That means all 12 available QPUs are used for the implementation.

4.1.2 Preprocessing

During preprocessing r, n', and r^{-1} have to be calculated from n.

Since the memory accesses are technically very important for the GPU runtime, the parts of the algorithm that are executed only once are not migrated to the GPU, but executed on the CPU. This includes the part *preprocess*(). The parameters r, n', and r^{-1} are calculated from n here. These remain for the entire encryption, since the same key is used with the same module n.

It is not possible to calculate these three parameters in real parallel on the GPU, because the GPU is addressed via single instruction multiple data, and the three parameters are not calculated in the same way. Parallelizing single operations used in the calculation, such as the advanced Euclidean algorithm, were not tried. Since this operation is also only executed twice, it is assumed that the overhead of the data transfer to the GPU combined with the slower execution time of the GPU is greater than the time gain resulting from the parallel execution. However, this still could be tried.

For this implementation it was nevertheless decided to execute *preprocess*() sequential on the CPU. Since it contains multiprecision operations, they are calculated using the GNU Multiple Precision Arithmetic Library (GMP) [G+], which contains the extended Euclidean algorithm, so that n' and r^{-1} can be calculated easily.

For this at first the `char*` input is converted to the GMP type `mpz_t`:

```
mpz_init(n);
mpz_set_ui(n,0);
mpz_set_str(n, input_string, 10);
```

Listing 4.1: Initializing and loading value into multiprecision variable n as example

Unfortunately, it does not contain a logarithmic function that is actually required to calculate r. Since $r = b^{\lceil log_b(n) \rceil}$, r can also be calculated via a detour (see figure 4.1). For this n has to be converted from the GMP type `mpz_t` into the number system, in which also the exponentiation is carried out later. This has to be done anyway, so that this doesn't cause additional time. r can then be represented as a number with one digit more than n, where the most significant digit gets the value 1 and all other digits get the value 0. Then it is converted into GMP data type `mpz_t` again. This is done only once and is not expected to cost long time.

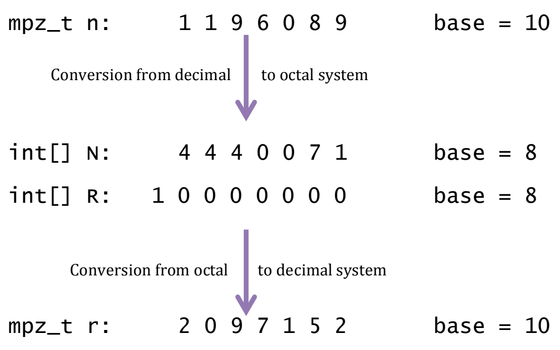

mpz_t n: 1 1 9 6 0 8 9 base = 10
Conversion from decimal to octal system
int[] N: 4 4 4 0 0 7 1 base = 8
int[] R: 1 0 0 0 0 0 0 0 base = 8
Conversion from octal to decimal system
mpz_t r: 2 0 9 7 1 5 2 base = 10

Figure 4.1: Calculation of r with $n = 1196089$ and $base = 8$

After calculating n and r in GMP datatype `mpz_t` the GNU implementation of *gcd*() can be used to calculate n' and r^{-1}:

```
// Initialisation of the multiprecision variable n'
mpz_t n_;
mpz_init(n_);
mpz_set_ui(n_,0);
// Initialisation of the multiprecision variable r^(-1)
mpz_t r_;
mpz_init(r_);
mpz_set_ui(r_,0);

```

```
mpz_gcdext(NULL, r_, n_, base, n);
mpz_neg (n_, n_);
mpz_mod (n_, n_, base);

mpz_gcdext(NULL, r_, NULL, r, n);
mpz_mod (r_, r_, n);
```

Listing 4.2: Calculating the greatest common divisor during preprocessing

At this time n, r, n' and r^{-1} are all accessible as GNU multiprecision datatype mpz_t.

4.1.3 Transformation into Montgomery form

The transformation into Montgomery form ($h(m) = m * r \mod n$) is processed for every m_i after each other. This is done by a for loop that iterates over one dimension of the array M.

Since the transformation includes a modulo operation, where the module is not a power of 2, the transformation cannot be calculated on the GPU by using a logical bitwise AND operation, as it is done during Montgomery algorithm. A modulo operation neither a division is provided by the QPULib itself. Besides, the question arises whether the slower execution time and memory accesses on the GPU would actually lead to longer execution time for this one-time calculation. For these reasons, the transformation of the partial messages m_i is calculated sequentially on the CPU and only then transferred to the GPU. Therefore also the GNU Multiple Precision Arithmetic Library (GMP) [G+] is used, that contains a modulo operation for multiprecision intergers.

The following example shows how the Transformation is calculated using the GMP library:

```
//Initialisation of the multiprecision variable h(m)
mpz_t h_m;
mpz_init(h_m);
mpz_set_ui(h_m,0);

// h(m)= (m * r) mod n
mpz_mul (h_m, m, r);
mpz_mod (h_m, h_m, n);
```

Listing 4.3: Transformation of m into Montgomery form $h(m)$

4.1.4 Preparation of the data for processing with the QPULib

All data used for RSA encryption is now prepared for the Montgomery exponentiation and accessible as GMP type mpz_t. Until the GPU can use these data for exponentiation some more steps are needed to access the data with the QPULib on the GPU. In the following three paragraphs, first the multiprecision integers that were defined newly for calculations on the GPU are explained. Later the conversion of GMP type mpz_t into these newly defined GPU multiprecision integers is explained. Finally these multiprecision integers are loaded into memory that can be accessed by the CPU and GPU.

Multiprecision numbers on the GPU Since the RSA encryption requires very large numbers, but the QPULib does not support a multiprecision library, the multiprecision integer and the operations used on it must be implemented independently for the GPU.

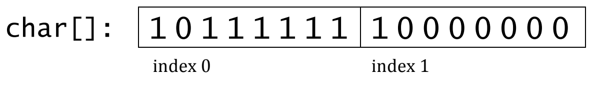


Figure 4.2: Representation of the number 509 in a multiprecision integer consisting of *char*s

These are realized by arrays of a certain data type. They are big endian, so that the index of each element corresponds to its value in the number system. They can be imagined as numbers of any number system, except that the value of each array element stands for a certain digit, and the base of the number system is one greater than the maximum possible value of the array elements.

The figure 4.2 shows how for example `char` arrays can be used to represent larger numbers than are actually possible with 8 bits.

Here the number 509, which actually can't be represented in a `char` is represented by a `char` array. Since a `char` can have a maximum value of $2^8 - 1 = 255$, the largest possible base can be $2^8 = 256$. It is pointed out that any smaller value would be possible as a base, but would lead to unused bits and resulting wastage of memory. Now the desired number 509 must be converted into the number system with the base 256 of which the digits can be numbers from 0 to 255. This way we get the number $(517)8 = 2531$, because $253 * 256^0 + 1 * 256^1 = 509$. The first element of the array contains the value 253 and the second one contains the value 1. Together they are interpreted as a multiprecision integer with the value 509.

Since the Montgomery algorithm is executed on these multiprecision integers, the number of iterations of the algorithm depends on the length of these arrays. For this reason it is necessary to keep the arrays as short as possible, which means to select the individual elements of the arrays as large as possible. The figure 4.3 shows how the performance of the Montgomery exponentiation changes depending on which base is chosen for the algorithm. The seconds relate to a sequential execution of the algorithm with only one 128 Byte message as input performed on a 2,2 GHz Intel Core i7. But also here it can be seen how the base has influence on the performance.

Because neither the QPULib nor the VideoCore IV supports the data types `long` and `unsigned int` we chose *pointer* on `int` arrays for the multiprecision integers. In this way, 31 bits can theoretically be used in each element of the array before an overflow occurs. The largest possible value is recommended because the bottleneck of the GPU is the memory access.

Conversion of GMP data to multiprecision number As explained above, the numbers n, n' and $h(m)$ are calculated with the GMP library. Since these numbers are used further on the GPU, they must be transferred into a data structure usable on the GPU. For this the multiprecision integers explained in chapter 4.1.4 were chosen.

The new multiprecision integers are filled by representing the gnu `mpz_t` number binary and then converting it to the base of the numeral system in which the Montgomery algorithm will be calculated in. Since the base for the Montgomery algorithm should be a power of two anyway, it is easy to read $log_b(base)$ many digits from the binary form and stored them as one element of the array.

For the base, the following applies: $base = 2^x$ and the larger it is, the better, because

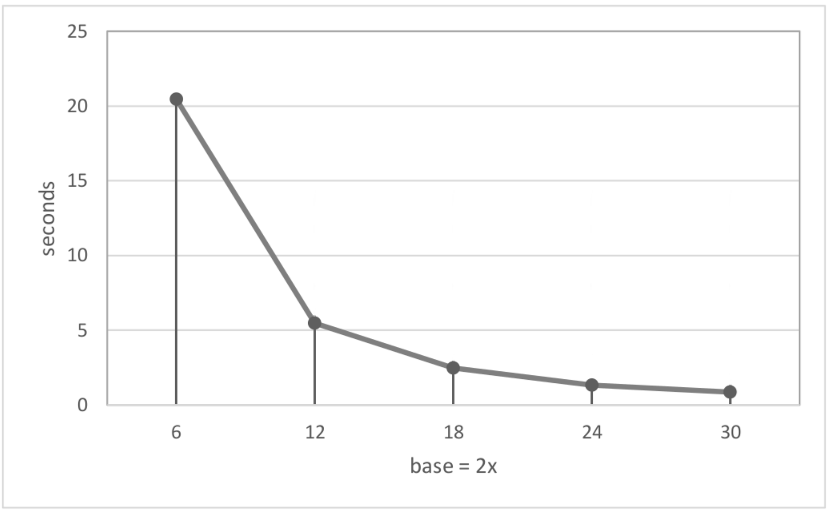


Figure 4.3: Performance of the sequential modular exponentiation calculated in different number systems

then the resulting multiprecision integer is shorter. Still it must not be so large that the operations applied on the multiprecision numbers lead to an overflow. A possible overflow is determined by the size of the base in combination with the way the multiprecision operations are implemented.

For this implementation the $base = 2^{30}$ was chosen. This means that each element of the integer array is filled to 30 of 32 bits with data. Since the chosen data type for the multiprecision integers is `int`, only 31 bits are generally available for positive numbers. Because two numbers of the same size in a number system can lead to a carry of maximum one digit, the sum of two numbers of 30 bits can still be stored in the 31 bits of an `int`. A normal Addition operation is therefore possible with this base. For multiplication, the result can be twice as many bits long. Therefore the base could only be 15 bits so that the result can be stored in the available 31 bits of an `int`. Since this would mean a memory utilization of only 50% during all other operations, it was decided to allow the multiplication of 30-bit numbers by a more complex multiplication operation, since the memory access which is the bottleneck of the GPU can be reduced by this. This is why the $base = 2^{30}$ is used.

In total the GMP `mpz_t` numbers n, n' are calculated only once for the whole message M and the Montgomery form $h(m_\mathrm{i})$ is calculated only once for each message part m_i on the CPU. Still on the CPU they are stored in data structures usable for the GPU. From this moment on it is possible to continue the calculation parallel on the GPU with as little memory as possible.

Allocating memory accessible by CPU and GPU To be able to access the data with the QPULib, they must be loaded into the QPULibs datatype `SharedArray<int>`. The chosen length of these `SharedArrays` is explained later. The `SharedArrays` are initialized as follows:

```
SharedArray<int>     qpu_h_m(p_len*16        * numberOfQPUs),
                     qpu_n_(16),
                     qpu_new_hm(p_len*16     * numberOfQPUs),
                     qpu_t(16*(p_len+r_len) * numberOfQPUs),
                     qpu_s(16*(p_len+1)      * numberOfQPUs),
                     qpu_n(16*p_len          * numberOfQPUs);
```

Listing 4.4: Initializing the needed `SharedArray<Int>`

Since each QPU calculates on a 16 values long vector, the used constants n' and n are set as `SharedArrays`, whose 16 vector elements are all filled equally.

Because n' is always smaller than the base, a `SharedArray` with only 16 elements containing the value of n' is passed to the GPU. This is done as shown in listing 4.5.

Since n only fits into a multiprecision integer, a `SharedArray` is created that has 16 times the length of the multiprecision integer n. These elements are filled so that the first 16 elements each contain the 1st digit of the multiprecision integer n, the next 16 elements all contain the second digit of the multiprecision integer n and so on. The figure 4.4 shows abstractly how the `SharedArray` is filled with the data of n. The implementation is shown in listing 4.5.

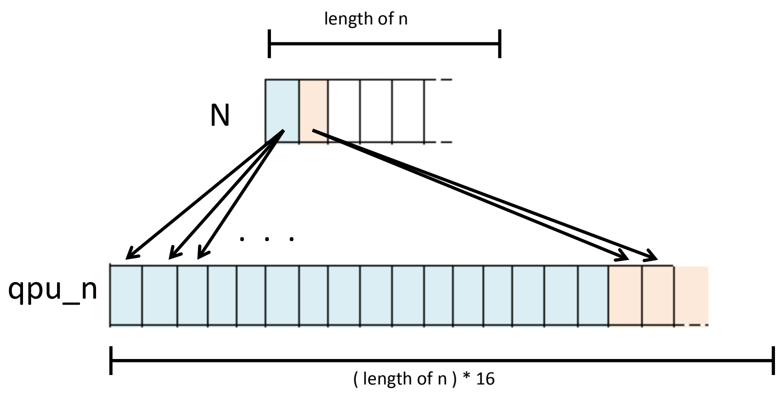


Figure 4.4: Sequence of loading the multiprecision integer N into the GPU data `qpu_n`

In order to use the computing power of the QPU, 16 different messages should always be calculated simultaneously on one QPU. Additionally several QPUs are used. Since the GPU executes the same instructions on all data, all data used by the QPUs which are all $h(m)$ must be loaded into the same `SharedArray`. This means that the `SharedArray`, which should contain all multiprecision integers $h(m)$, must have the length of one multiprecision integer $h(m)$ times 16, for all vector elements calculated simultaneously in a QPU, times the number of QPUs used. For 12 used QPUs and a length of 35 elements per multiprecision integer $h(m)$ the `SharedArray<int> qpu_h_m` reaches a length of 16*12*35 integers.

Figure 4.5 shows the order in which the `SharedArray<int> qpu_h_m` is filled with the different messages in Montgomery form $h(m_\mathrm{i})$. All 16*12 1st digits of the multiprecision integers $h(m_\mathrm{i})$ are loaded into the first 16*12 addresses of the shared array. All 16*12 second digits of the multiprecision integers are loaded int the addresses 16*12 to 2*16*12-1, and so on.

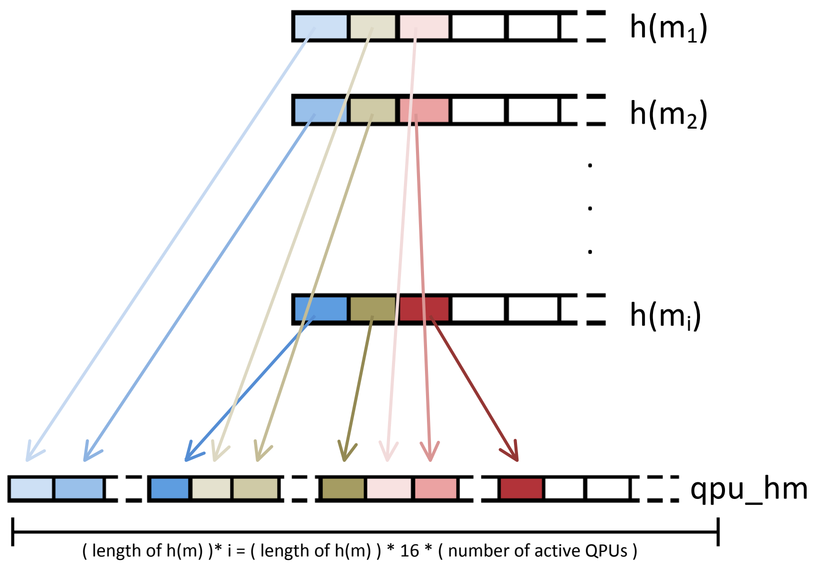

h(m1)
h(m2)
h(mi)
qpu_hm
(length of h(m))* i = (length of h(m)) * 16 * (number of active QPUs)

Figure 4.5: Sequence of loading the multiprecision integers $h(m_i)$ into the GPU data `qpu_h_m`

The Listing 4.5 shows how the filling of three variables `qpu_n_`, `qpu_n` and `qpu_h` with data is implemented.

```
//qpu_n__
for (int i = 0; i < 16; i++) {
    qpu_n_[i] = n__;
}
//qpu_n
for (int i = 0; i < 16*35; i++) {
    qpu_n[i] = N[i/16];
}
//qpu_h_m
for (int i = 0; i < 16*35; i++) {
    qpu_h_m[i] = H_M[((i*35)%(35*16)) + i/16];
}
```

Listing 4.5: Loading the data into the needed `SharedArray<Int>`

The `SharedArrays qpu_new_hm, qpu_t, qpu_s` are not filled with data, because they will be filled during the algorithm.

4.1.5 Exponentiation by Squaring

The exponentiation by squaring takes place directly afterwards. Before it starts, the QPULib function used, the Montgomery multiplication, has to be compiled and the number of QPUs is set, as mentioned in the background in chapter 2.4.3. The exponentiation is done as described in the theoretical algorithm in chapter 2.3.2. The exponent, which is available as GMP data type `mpz_t`, is divided by 2 using the GMP library after each iteration. Within each iteration the Montgomery multiplication is executed by a QPULib call on the GPU.

During each multiplication, the data for each $h(m)$ in the `SharedArray` is changed by the GPU operations and serves as a new value in the next iteration. As soon as the exponent is 0 the exponentiation is finished. This comparison is also done with the GMP library. In the following the implementation of the exponentiation by squaring is shown:

```
// Compiling the Montgomery multiplication performed on the GPU
auto kernel = compile(mont_mul);
// Set number of QPUs
kernel.setNumQPUs(numberOfQPUs);

//Exponentiation by Squaring
int exponentiation_flag = 0; // flag for first iteration of exponentiation

while (0 < mpz_cmp_d(mpz_exponent, 0)) {
    if (mpz_odd_p(mpz_exponent)){
        if(exponentiation_flag != 0){
            kernel(&qpu_h_m, &qpu_new_hm, &qpu_t, &qpu_s, &qpu_n, &qpu_n_);
        }else {
            for (int i = 0; i < 16*p_len * numberOfQPUs; i++) {
                qpu_new_hm[i] = H_M[((i*p_len)%(p_len*16*numberOfQPUs))
                                + i/(16*numberOfQPUs)];
            }
        exponentiation_flag = 1;
        }
    }
    kernel(&qpu_h_m, &qpu_h_m, &qpu_t, &qpu_s, &qpu_n, &qpu_n_);      mpz_tdiv_q_2exp(mpz_exponent, mpz_exponent, 1);
}
```

Listing 4.6: Exponentiation by Squaring

4.1.6 Transformation out of Montgomery form

After the Montgomery exponentiation, implemented as exponentiation by squaring, all results for each message in Montgomery form $h(m_\mathrm{i})$ are now ciphers in Montgomery form further called $h(c_\mathrm{i})$. They have to be transformed out of the Montgomery form into their original form c_i.

Therefore all $h(c_\mathrm{i})$ are loaded individually from the `qpu_new_hm` in a for loop exactly opposite to the loading in chapter 4.1.3, converted into the `mpz_t` variable `result_h` and then transformed back out of Montgomery form using the GMP library. All used numbers are already available as GMP data. The implementation for the re-transformation is as follows:

```
mpz_t result;
mpz_init(result);
```

```
mpz_set_ui(result,0);

// h(m) = (m*r)mod N
mpz_mul (result, result_h, r_);
mpz_mod (result, result, n);
```

Listing 4.7: Transformation out of Montgomery form

All ciphers c_i calculated in the variable result are converted into a string and stored in the two dimensional array M, returned as the result.

4.2 Implementation on the GPU

As in chapter 3.4 explained the implementation on the GPU performs solely the Montgomery multiplication.

The QPULib is used to execute the Montgomery multiplication on the GPU. Each QPU can process 16 data simultaneously and the VideoCore IV of the Raspberry Pi has a total of 12 QPUs. All 12 QPUs are used during the implementation.

In the following the implementation of the Montgomery multiplication with the help of the QPULib is explained. In addition, the implementations of the called subfunctions, the multiprecision multiplication, multiplication, multiprecision subtraction and multiprecision comparison are shown. The special aspect is that all QPUs have to access the same `SharedArray` and have to be coordinated.

4.2.1 Montgomery Multiplication

To understand the implementation of the Montgomery multiplication, the algorithm introduced in chapter 2.3.4 is recalled. Algorithm 7 shows the same Algorithm, but numbers all parts to enable a more structured description of the implementation. To give a short summary of the algorithm, first the normal product of the two multiplicants is calculated in part 1. The product T is then processed in a double for loop so that it is finally reduced again to half the length in part 2. The result S is then compared with the given modulo N in part 3, and in the case that S is greater than N, N is subtracted from S in part 4. This result is returned as NEW_H_M so that it can be used for the next iteration as described in the sequential implementation.

As explained in the introduction of the QPULib the data types `SharedArray<int>` get transformed to the GPU accessible data type `Ptr<Int>`. That's why the implementation of the Montgomery multiplication `mont_mul` has input parameters of the type `Ptr<Int>` only in the number matching to the kernel called in the CPU implementation. For better reading, the variables are now again written in Capital letters. Listing 4.8 shows the function with its parameters.

```
void mont_mul(Ptr<Int> H_M, Ptr<Int> NEW_H_M, Ptr<Int> T,
                        Ptr<Int> S, Ptr<Int> N, Ptr<Int> n__)
```

Listing 4.8: Montgomery Multiplication parameters

For the multiplication of both input multiplicants `H_M`, `NEW_H_M` in part 1 the method `qpu_mul` is called, that performs a multiprecision multiplication. Its implementation is ex-

Algorithm 7 Montgomery Multiplication in four parts

```
Input: Base $B = 2^x$ with x $\in N$
  $H\_M$, $NEW\_H\_M$ represented in $p$ digits,
  $N$ with $gcd(b, N) = 1$ and represented in $p$ digits,
  $N'$ with $0 \leq N' < B$
Output: $S$ with $0 \leq S < N$ and represented in $p$ digits

  // part 1
  $T \leftarrow H\_M * NEW\_H\_M$
  $T[r + p] \leftarrow 0$

  // part 2
  for $0 \leq i < r$ do
    $c \leftarrow 0$
    $m \leftarrow T[i] * N' \mod B$
    for $0 \leq j < p$ do
      $x \leftarrow T[i + j] + m * N[j] + c$
      $T[i + j] \leftarrow x \mod B$
      $c \leftarrow \lfloor x/B \rfloor$
    end for
    for $p \leq j \leq r + p - i$ do
      $x \leftarrow T[i + j] + c$
      $T[i + j] \leftarrow x \mod B$
      $c \leftarrow \lfloor x/B \rfloor$
    end for
  end for
  for $0 \leq i \leq p$ do
    $S[i] \leftarrow T[i + r]$
  end for

  // part 3
  if $S \geq N$ then

    // part 4
    return $S - N$
  else
    return $S$
  end if
```

plained in chapter 4.2.2. The highest value of the multiprecision integer T is set to 0 by a store call, to not block the processing through waiting for stores.

Then the result T of the multiplication is processed in part 2. Therefore two for loops are processed in one for loop. Its incrementation parameters are shown in listing 4.9.

```
For (Int i = 0 + me()*16, i < 35*16*num_QPUs, i = i+16*num_QPUs
   )
    ...
    For (Int j=0, j<35*16*num_QPUs, j=j+16*num_QPUs)
        ...
    End

    For (Int j=35*16*num_QPUs, j<=2*35*16*num_QPUs-i,
                                        j=j+16*num_QPUs)
        ...
    End
End
```

Listing 4.9: Incrementations in the double For loop

In the implementation all often used values as of example 16*num_QPUs are stored in variables so that they don't have to be calculated newly every time.

The increment of the counters i and j by 16 times the number of QPUs is explained by the fact that each QPU processes 16 values at the same time. So after an iteration 16 times number of QPUs any values were processed and the new ones, starting at the new value of the increment have to be processed now.

The 35 in the For loop results from the fact that the multiprecision integer in this implementation consists of 35 elements. Therefore the Shared arrays are also 35*16*num_QPUs long.

So the parallel processing of different data takes place via the iteration, whereby all QPUs get different start positions of the multiprecision integer T. Therefore each QPU calculates its starting value with the function me(), and adds this result, the unique id of the QPU, times 16 to the given starting value that is the same on each QPU: T = T+me()*16. This way each QPU processes different data in the end.

The individual operations used within the loops do not yet require multiprecision operations, since each element of the multiprecision integer T is iterated over individually.

However, there are three special features of the implementation at this point.

First, each modulo operation is replaced by a bitwise AND, as explained in chapter 2.3.3. This is possible because the used module B is a power of 2. Only this allows a modulo operation to be executed on the GPU at all, since not even a division is hardcore implemented on the VideoCore IV.

Second, each division is calculated by a bitshift, because the divident is also a power of 2. How this is possible was explained in chapter 2.3.3 as well.

And third, the modular multiplication of two integers is done by a split multiplication, which is described in chapter 4.2.3, because the multiplication of two 30 bit integers could otherwise cause an overflow.

Furthermore, in order not to wait for write and load processes, all memory accesses of the inner For loops were implemented with gather, receive and store, as explained in chapter 2.4.3.

After the execution of part 2, S is returned to all QPUs.

In part 3 S is now compared with N and in case S is larger than N, N is subtracted from S in part 4 and later the result is stored in the multiprecision integer NEW_H_M.

Since, as explained in chapter 2.4.3, because of the SIMD principle, no `For` loop can be executed in a `Where` condition, the algorithm cannot be implemented here as written. The multiprecision subtraction can only be performed in a `For` loop, and therefore must not be implemented in a `Where` statement that compares the size.

That is why the comparison of S and N is done first. Therefore the function `qpu_compare` is called whose implementation is described in chapter 4.2.4. As a result, each QPU receives an `Int`, whose 16 elements indicate for all 16 different S whether it is greater than N. The call of the function is shown in Listing 4.10.

```
Int compare = 1;
qpu_compare(S, N, &compare);
```

Listing 4.10: Comparison of `S` and `N`

To perform the subtraction now nevertheless only on those where `S` is greater than `N`, all steps of the multiprecision subtraction which require only read accesses are performed on all 16 vector elements in every iteration over the multiprecision integer. Before the result of each iteration is written into the result `NEW_H_M`, it is now tested by a Where whether this vector element is part of an `S` grater than `N` at all and only then actually changed.

That is why the multiprecision subtraction was not implemented in a separate function, but embedded in the Montgomery multiplication. In detail it is performed as follows. To subtract `S` from `N`, the same element of the multiprecision integer `N` is subtracted from the element of the multiprecision integer `S` in each iteration of a `For` loop, starting with index 0, the least significant digit of the multiprecision integers. Since each element is a normal integer, the normal subtraction of the QPULib can also be used. Using a bitwise AND modulo of the base on the difference the new digit for the same index of the resulting multiprecision integers `NEW_H_M` is calculated. If the result is less than 0, a carry of 1 is stored for the next multiprecision integer element, which is then additionally subtracted from the next digit. Before jumping into the next iteration, the result of the `compare` call is used to check whether `S` is greater than `N`, and if so, the result diff is also written to the appropriate element of `NEW_H_M`.

```
Int c = 0;
Int diff = 0;
For (Int i = 0+me()*16, i < 35*16*num_QPUs, i = i+16*num_QPUs)
        diff = (*(S+i) - *(N+i) - c) & BASE;
        c = 0;
        Where (*(S+i) - *(N+i) - c < 0)
            c = 1;
        End
        Where (compare > 0)
            *(NEW_H_M+i) = diff;
        End
End
```

Listing 4.11: Subtraction of `N` from `S`

Listing 4.11 shows the basic implementation described above where part 3 and part 4 of the classic algorithm is marked. Here, as well, the parallel processing of different data is implemented by each vector element starting with a different pointer to `S`, `N` and `NEW_H_M` because of the call `me()` and the incrementation of counter `i` leads to a skipping of all data that was already processed by other QPUs. In addition, and not shown in the listing, all memory accesses are also performed with gather, receive and store.

At this time a whole Montgomery multiplication has been performed, with the result of all `16*numQPUs` multiprecision integer stored unter the `Ptr<Int> NEW_H_M`. The execution is passed back to the CPU, which completes the exponentiation by squaring, and if necessary calls the Montgomery multiplication on the GPU again, with the data in `NEW_H_M` having now changed.

4.2.2 Multiprecision Multiplication

The multiprecision multiplication `qpu_mul` is the most complex multiprecision operation that is implemented, because it iterates in a double loop over both multiplicants. It's implementation can be seen in the appendix 6.1. As in textbook mathematics, each digit of the two numbers, in our case each element of the multiprecision integer, is multiplied by each other. Depending on the size of the result, a carry is set for the next iteration, that means the next larger digit of the result.

Also here the iteration and the addition of `me()` to `Ptr<Int>` allows the parallel processing of different data. Furthermore, the required multiprecision elements are loaded in both `For` loops via `gather` and `receive`, and stored with `store`.

The special feature of the implementation lies in the multiplication of the individual 30 bit multiprecision elements and the storage of the carry, without the results being falsified by an overflow. The split multiplication from chapter 4.2.3 is used for this.

4.2.3 Multiplication

Integer multiplications are performed at several points of the Montgomery algorithm. As an example, during multiprecision multiplication, each element of a multiplicant is multiplied by each element of the other multiplicant.

The problem with this multiplication is that both multiplicants can be up to 30 bit in size. So their product could take up to 60 bit and therefore not fit into the used data type `Int`. There can not be calculated with bigger data types on the QPU and in addition, only 24-bit multiplications are possible on the VideoCore IV.

The function `qpu_split_mul` enables this calculation and returns the result in an array with two elements, where again each element stands for a digit in the number system of the base. Its entire implementation can be found in appendix 6.2.

The calculation is carried out by first dividing the multiplicants with right shifts and AND operations into four integers, which are filled in half of the base, that is 15 bit. These four elements are now handled like a new multiprecision integer with base 15 and a multiprecision multiplication is performed with them, only that the multiplication of the single 15 bit elements is now possible. At the end the result is again in 4 integers, which are only filled up to 15 bit. This process is shown in figure 4.6. Then the four integers can be stored in two integers again with the help of left shifts and OR operations, which are again filled up to 30 bit. This result can be seen as a two-digit number of the original number system with the base 30 again, whereby the higher digit in the multiprecision multiplication can be taken over immediately as carry.

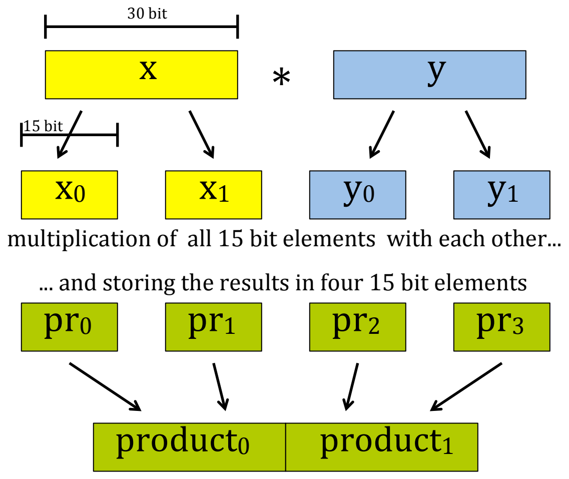

30 bit
x
*
y
15 bit
x_0
x_1
y_0
y_1
multiplication of all 15 bit elements with each other...
... and storing the results in four 15 bit elements
pr_0
pr_1
pr_2
pr_3
$product_0$
$product_1$

Figure 4.6: Multiplication of two 30 bit numbers by splitting into 15 bit numbers and merging after the multiplication of the single 15 bit numbers

In the case of modular multiplication, as for example when calculating `T[i] * n' mod B = t*n' mod B`, during the `For` loops in Montgomery multiplication also the split multiplication is used, whereby the least significant result digit directly represents the result of the modulo operation. A normal multiplication is not possible, although the possible overflow would not matter because of the following modulo operation. This is because only 24 bit numbers and no 30 bit numbers can be calculated on the QPU. The result would still be wrong.

4.2.4 Multiprecision Comparison

The comparison of two multiprecision integers is performed with the function `qpu_compare`. To do this, pointers to the two shared arrays `S`, `N` with the multiprecision integers and a result Int are passed as parameters.

As before, all QPUs get the same pointers, which they use to access different data for different multiprecision integers using `me()`. As result, each QPU passes its own `Int`.

The method iterates over the given multiprecision integers, starting with the most significant bit. As soon as it is noticed that one element is larger than the other, the result can be set. If `S` is greater than `N`, the result is set to 1. If the size is same than 0, and otherwise -1.

Because each QPU compares 16 values simultaneously, this is done via a Where. Depending on where `x > y` the result `Int` is set to 1, otherwise it remains as it is.

As soon as the result for one of the 16 elements is fixed, this element is set to 1 in another `Int`, the `flag`. This shows in the next iteration if a decision is fixed or if you still have to set the result for this element.

How to compare the `flag` and the elements in the `For` loop is shown in Listing 4.12. The gather and receive commands used in the real implementation have been omitted for better understanding. The full method is listed in the appendix 6.3.

```
For (Int i = 0+me()*16, i < 35*16*qpu_nums, i = i+16*qpu_nums)
        Int x = *(S + (34*16 * qpu_nums - i + 16));
        Int y = *(N + (34*16 * qpu_nums - i + 16));
        receive(xOld);
        receive(yOld);
        Where (flag == 0)
            Where (x > y)
                *result = 1;
                flag = 1;
            End
            Where (x < y)
                *result = -1;
                flag = 1;
            End
        End
End
```

Listing 4.12: Comparison of single multiprecision integer elements

4.3 Difficulties of the RSA Implementation

The previous sections described exactly how the RSA algorithm from chapter 3.4 was implemented. First the part that was calculated sequentially on the CPU was described, then the part that is executed in parallel on the VideoCore IV. In addition to the concepts already described, there are additional optimization possibilities and special issues considered during implementation that are discussed in detail here.

Especially with the GPU implementation a further optimization could be done by omitting the multiprecision integer S in the Montgomery multiplication and simply replacing it with a pointer pointing to the desired position in T. This could skip a few operations that are currently being performed. Unfortunately this was not possible in this implementation due to time constraints.

Some behavior of the QPULib or VideoCore IV could not be explained during the implementation and led to individual differences in the implementation which have not yet been listed. For example, the application of gather and receive does not always work. At one place the wrong data is delivered. For this reason, the data is read and written back using pointers instead. The values loaded using gather and receive are therefore not used. But as soon as the gathers are commented out, the Montgomery multiplication returns wrong results at some stage of the exponentiation although the FIFO was empty at the beginning and at the end again. Therefore, in the final implementation, gather and receive are called, but not used.

Also unclear is the different behavior of all QPUs in some cases. In general, variables which store the results of very frequent calculations have been created for the implementation so that they do not always have to be recalculated. For example, in all For loops the value of `me()*16` was replaced by a variable. At exactly one point in the implementation, this only does not work for some QPUs. Which and how many is always different. In any case, 3 of the 12 QPUs usually give incorrect results, while the others give the correct result as soon as the variable instead of `me()*16` is given at a certain position. Because always different

QPUs give wrong results, it was found that every QPU can sometimes get the right result. Again, no explanation could be given why the operation `me()*16` has to be recalculated at this point instead of being loaded from a variable.

With special attention to these peculiarities the implementation of RSA which runs parallel on VideoCore IV works. How the implementation was tested and which results were drawn can be found in the next chapter.

5 Evaluation

In this chapter the developed RSA implementation, that runs partly on the Raspberry Pi's GPU, will be tested.

The fact that the implementation also fulfills the RSA standard was verified by encrypting data on the GPU and afterwards successfully decrypting it with the raw OpenSSL RSA algorithm and the other way around. Therefore, the RSA implementation is considered correct. At this point it is mentioned again that the algorithm is still not necessarily secure because it does not automatically pad the message to be encrypted. But still outsourcing of parts of the RSA algorithm to the GPU is thus possible.

Furthermore, the question shall be answered whether the parallel execution on the GPU has an advantage over the sequential execution on the CPU. For this the execution time of the partially parallel algorithm is compared with that of a sequential implementation.

As a sequential comparative algorithm the RSA implementation of OpenSSL was chosen. In order to be able to compare the two execution times in the correct ratio, it was decided to execute both implementations the same way using an OpenSSL engine. How this was implemented is explained in chapter 5.1.

To test the performance of cryptography, OpenSSL provides the functionality speed, which measures the maximum number of engine calls for 10 seconds and calculates the throughput of bytes per second for each engine. Unfortunately, the speed function cannot be used for the engines used because the input of the GPU implementation does not correspond to the usual number of bytes. For this reason it was decided to measure the execution time of 100 invocations for both engines and then to calculate a comparable throughput for each. The exact details of this test can be found in chapter 5.2.

After that the measured test results are presented and compared in chapter 5.3 and finally possible reasons for the results are discussed.

5.1 OpenSSL Engine

The OpenSSL engine is provided by OpenSSL to modify only single parts of existing OpenSSL implementations, to for example optimize them to a special hardware, as it is the case here. It has to be implemented in c code. To run such an engine instead of the usual OpenSSL algorithm, simply the flag `-engine` and the absolute path to the compiled engine have to be added to the usual OpenSSL call.

Two RSA engines were implemented for the evaluation. One that tests the GPU implementation and one that tests the standard sequential RSA encryption on the CPU in the same experimental setup to produce comparable results. In both engines the standard methods `rsa_pub_enc` for encryption and `rsa_priv_dec` for decryption were overwritten.

Normally both engines read the key and data automatically and provide it in every part of the engine. The result is also written to a file automatically. A developer just gets the message in a pointer, encrypts it with his own implementation, and then writes the cipher to another pointer again. The key is provided the same way, in a pointer on a OpenSSL `BIGNUM` data type and doesn't have to be read manually neither. But the length of input data is limited to 256 bytes, whereby the output must not exceed 128 bytes. This is much less than the 24576 bytes that the developed RSA algorithm processes. For this reason, both engines read a file with the data to be processed and also write the result to a file during the actual encryption code. The file names are given inside the code. The padding is also not taken over by the engine. For this reason, both engines encrypt without padding, which is why the encryption is correct, but not secure. The padding could be added as future work.

Apart from reading the same data and writing the results to a file, the encryption and decryption methods called by the engines behave differently.

The engine for the GPU implementation brings all 24576 bytes data into the form expected by the implementation as parameters and then executes the parallel implementation only once on all data at the same time. The result is again ordered from a two-dimensional array into a sequential byte sequence before it is written into the result file. The encryption and decryption of the GPU engine differs only in the use of either the public key for encryption or the private key for decryption.

The engine that runs the sequential OpenSSL RSA algorithm splits the 24576 bytes read data into 128 byte packets and successively performs the usual RSA encryption or decryption method on all 128 byte packets.

This means that both engines process the same amount of data, with the only difference that one algorithm runs partially in parallel on the GPU and the other completely sequentially. The implementation of both engines can be found in Appendix 6.

5.2 Testsetup

In addition to the fact that the RSA implementation on the GPU is also correctly encrypted and decrypted, the execution time of the parallel GPU implementation shall be examined against a sequential implementation on the CPU. That's why the execution times of the encryption and decryption of the two engines introduced in the previous chapter are tested. The GPU engine represents the execution time of the RSA algorithm which is partially implemented on the GPU in parallel. The OpenSSL engine represents an RSA algorithm executed sequentially on the CPU.

By measuring the execution times of the four methods, encrypting with the GPU engine, decrypting with the GPU engine, encrypting with the OpenSSL engine and decrypting with the OpenSSL engine, the four variants can be compared and conclusions drawn about the benefits of outsourcing to the GPU.

In order to measure the execution times of the two engines and compare them later, all four methods were executed 100 times in sequence and the time command was used to measure the time from the first to the last execution. By measuring the time of one hundred executions, the four methods can already be compared well with each other. In addition, an average execution time per call and a throughput of bytes per second can be calculated, considering that in 100 executions exactly 24.576 KB are processed.

The tests were carried out on the Raspberry Pi 1B+, which ran with a CPU frequency of 700 GHz and a GPU frequency of 250 GHz. The RAM was equally split between CPU and GPU. Further hardware details can also be found in chapter 2.4.

5.3 Results

In this chapter, the results of the tests on the parallel RSA algorithm implemented in this thesis are compared with the test results of the sequentially executed standard OpenSSL RSA algorithm, that were used on exactly the same data. Later, possible reasons for the results are discussed.

With the OpenSSL implementation, the encryption of 24.576 KB sequentially takes 34.547 seconds with a throughput of approximately 71.1379 KB/s. With the GPU implementation, 174.286 seconds are required, which corresponds to a throughput of 13.9132 KB/s. The comparison of both times for 100 executions is shown in graph 5.1. As a result, the OpenSSL implementation requires only 19.8% of the execution time of the GPU implementation on exactly the same data.

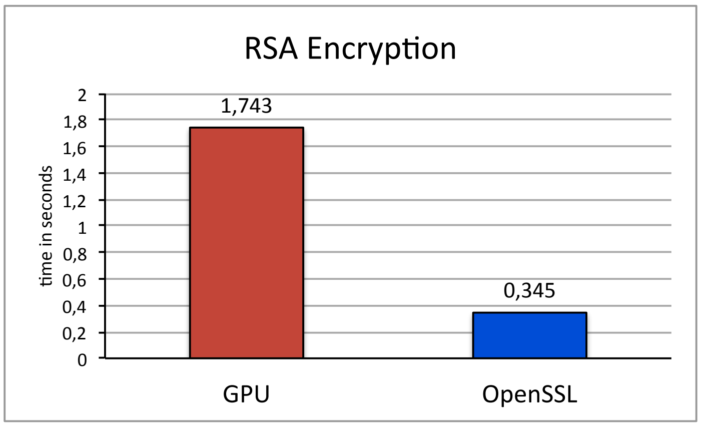


Figure 5.1: Comparison of RSA Encryption executed on the CPU and on the GPU

The decryption by the sequential OpenSSL engine processes 24.576 KB in 244.103 seconds, which corresponds to a throughput of 10067.8812 Bytes/s. The 100 executions of the GPU implementation take 4768.421 seconds, resulting in a throughput of 514.6127 Bytes/s. These values are reflected in figure 5.2. For the more complex decryption, the sequential implementation also takes 5.1% of the time required by the GPU algorithm.

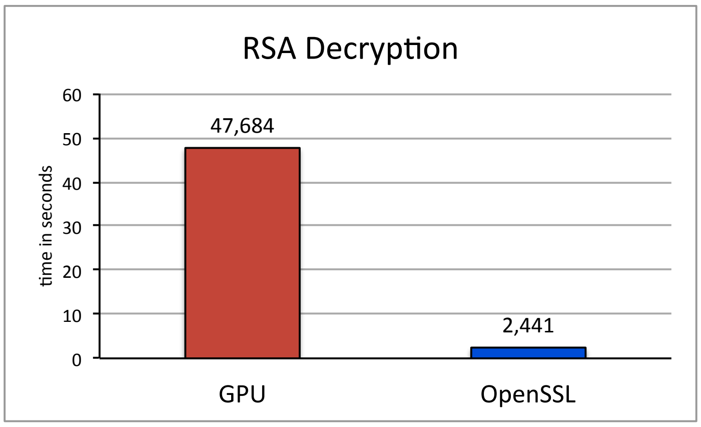


Figure 5.2: Comparison of RSA Decryption executed on the CPU and on the GPU

The table 5.3 summarizes all results at once.

Operation	**Time 100 Executions**	**Time single Execution**	**Throughput**
OpenSSL Encryption	34.547 s	0.345 s	71.2348 KB/s
GPU Encryption	174.286 s	1.743 s	14.0998 KB/s
OpenSSL Decryption	244.103 s	2.441 s	10.0680 KB/s
GPU Decryption	4768.421 s	47.684 s	0.5154 KB/s.

Figure 5.3: Overview of test results

Overall, the sequential execution on the CPU, as implemented by OpenSSL, is more than five times faster than the algorithm executed partially parallel on the GPU. Whether this result generally speaks against offloading calculations to the GPU, and which reasons could cause the longer execution time of RSA, are discussed in the following.

Discussion

The implementation, which has been partially tested on the GPU, requires a much longer execution time than a sequentially working one. Chapter 2.4.2 has already shown that the GPU with its multiple vector processors actually has a theoretically much greater computational potential than the CPU. Why the resulting GPU implementation is still slower than the sequential one will be discussed in the following. But it should be pointed out that the OpenSSL implementation is probably one of the best optimized RSA implementations and very difficult to rival.

One possibility for the slow calculation might be that the implementation is not optimized well. In fact, the implementation still has small weaknesses that could not be fixed due to time constraints. For example, in very few cases results are written into a new variable and then further calculated in this new variable instead of in the old one. However, this happens so rarely and only once that it cannot be the reason for the much slower calculation time.

What already costs more time is that the implementation works with the multiprecision operations of the GMP library. But the OpenSSL engine provides the numbers for the RSA encryption as OpenSSL data type `BIGNUM`. Therefore a transformation from `BIGNUM` multiprecision integers to GMP multiprecision integers still takes a lot of time and could be saved if the GPU implementation would use only OpenSSL `BIGNUM` operations instead of the operations offered by the GMP library. But since this also only happens once for input and output, it can't be the reason for the slow computation time either.

In addition, it has not yet been tested for which number of QPUs the execution time is fastest in proportion to the sequential implementation. It may be that processing less than 192 data, in other words using less than 12 QPUs, leads to a faster processing time due to less memory usage.

Another possibility for the long execution time is that RSA is generally not suitable for acceleration by outsourcing to VideoCore IV.

Chapter 3.3 showed that RSA can be parallelized in different ways. It also explained that many of these possibilities are not yet feasible with the QPULib and VideoCore IV. The finally implemented algorithm switches between the CPU and the GPU for each Montgomery multiplication. This switching always causes a large overhead [Sta17]. During decryption this switching may be done almost 2*1024 times and thus brings overhead just as often.

Because the decryption is even slower in relation to the encryption, it is assumed that this overhead is to a great part responsible for the slower execution time. This change, and the resulting overhead, would not occur so often with other parallelization options. Maybe the migration to the GPU could be accelerated by implementing another parallelization proposal. For this the QPULib would of course have to be extended, or another method for programming the QPUs would have to be used.

In addition, the 16-SIMD feature of the QPUs does not only bring advantages. Many steps in the implementation must be carried out, although they would not be necessary for all 16 data. For example, almost every step of multiprecision subtraction is performed on all data, even those that do not need to be subtracted. Their calculated result is simply discarded and not written back to memory. But it is calculated anyway. In the sequential algorithm this would not be necessary, but instead these steps could be omitted and therefore the whole process would be shorter.

The probably biggest factor that leads to the slow execution time of the GPU implementation is the bottleneck, which occurs during memory accesses. Although all QPUs can perform parallel operations, they share a VPM for memory access. This can only work sequentially and takes a very long time to load the data, since always 16 data have to be loaded. If all 12 QPUs load data from memory at the same time, then 11 QPUs have to wait and accordingly the last QPU is blocked for 11 memory accesses, which means more than 100 clock cycles [Sta17]. Since all multiprecision operations work in such a way that an array element is read and written directly back into memory and then the next element is loaded again, a lot of memory accesses occur and result in a lot of blocking. Because always new data is read from memory and never the same in a row, the L2 cache also doesn't bring any advantage. And since RSA consists almost exclusively of multiprecision operations, a huge bottleneck is created by memory accesses during RSA on the GPU, which becomes even more visible during decryption because many more memory accesses occur than during encryption. Maybe this could be reduced by always fetching as much data as possible from memory and then calculating all of them. This could still be tried although the registers of the QPUs are so few, that it should not make a really big difference. The bottleneck nevertheless cannot be avoided when accessing memory, and is seen as one of the main reasons for the bad performance of the GPU RSA implementation.

In summary, the GPU implementation does not nearly reach the performance of the CPU implementation of OpenSSL. This is mainly caused by the overhead of switching between CPU and GPU during exponentiation by squaring, as well as a large bottleneck in memory accesses by the QPUs.

6 Conclusion

The aim of this work was to run RSA on the GPU of the Raspberry Pi instead of on the CPU to relieve the CPU's workload. Furthermore, due to the theoretical potential of the GPU, an acceleration of the RSA algorithm was attempted. Due to the programmable 12 vector processors of the GPU, the theoretical computing power of the GPU is considerably higher than that of the CPU despite the lower frequency.

As a first step it was worked out how RSA can be calculated efficiently and in which way the found algorithm on the VideoCore IV, the GPU of the Raspberry Pi, could be parallelized.

The selected variant was then programmed on CPU and GPU using the c++ library QPULib. The implementation encrypts and decrypts 192 messages at once, whereby the individual multiplications of the exponentiation are simultaneously calculated on all 192 messages in parallel on the GPU.

After the implementation is able to successfully encrypt and decrypt RSA, it was integrated into the OpenSSL library. With the help of this OpenSSL engine the GPU implementation was tested and compared to a purely CPU calculated implementation. It appeared that the GPU implementation takes much longer than the RSA encryption on the CPU, even though the GPU's computing power is more powerful.

As a result of this work, it was concluded that the implementation of RSA according to the algorithm selected in chapter 3.3 makes no sense. Probably not even the CPU was relieved, but GPU and CPU were additionally loaded, since the implementation needs at least 5 times longer than the pure CPU implementation.

This is mainly attributed to two reasons. First, the frequent change between CPU and GPU execution with each multiplication leads each time to an overhead. In addition, RSA is based on multiprecision operations. And for these, a lot of memory accesses have to be made one after the other, leading to a large bottleneck when sequentially loading data from memory for 12 QPUs requesting simultaneously. For these reasons, the partially GPU executed RSA implementation is much slower than the CPU computed one by OpenSSL, with decryption being proportionally even slower than encryption.

In the future, we could work on methodologies to reduce these two problems of the current implementation and improve the run time a bit. Exactly what these are will be explained in the next section. However, the result of this work is that RSA is not suitable for running on the GPU of the Raspberry Pi, since the multiprecision operations required are none of the calculations that would make ideal use of the GPU's parallel computing power. It is not expected that RSA on the VideoCore IV can be greatly accelerated in comparison to a CPU implementation.

Future Work

This work is a first attempt to calculate RSA to the GPU of the Raspberry Pi. Although only one attempt for parallelization was implemented, several possibilities of how RSA could be executed on the VideoCore IV were developed.

For this reason, the next step would be to test the other parallelization possibilities in comparison to this one. Some of these variants were excluded because the QPULib does not allow the required number of memory accesses. However, it is assumed that the QPULib can be modified so that the number of accesses is unlimited. So this modification should be done so that not only the single multiplications but also the whole exponentiation can be executed to the GPU. Thus, the overhead that occurs when switching between CPU and GPU and currently can be almost 2*1024 times during decryption would only occur once. This implementation could therefore have a big run time advantage over the current implementation. In addition, different messages could be encrypted with different keys, or a message could be encrypted faster by parallelizing a modular exponentiation.

This work provides enough ideas to execute RSA in a different way on the GPU. Nevertheless, I believe that there are some cryptographic algorithms that have greater potential to be accelerated through the features of VideoCore IV. For this reason, I hope that this work will be used as a negative example to give an insight into the behavior of VideoCore IV and QPULib for RSA. The strengths of VideoCore IV lie in performing many operations on the same data with as few memory accesses as possible. Repeated read accesses to the same data can also be performed relatively quickly. In addition, its special feature is that it can rotate the data within the 16 vector without having to store it temporarily. Furthermore the cryptographic algorithm should encrypt so much data that all 192 possible operations can be used efficiently. Hopefully with this information a better choice for a new cryptographic algorithm can be made and successfully accelerated on the GPU of the Raspberry Pi.

Appendix

GPU methods

```
#define BASE_SIZE 30
#define HALF_BASE_SIZE 15
#define MOD_BASE 32767
#define qpu_nums 12

void qpu_big_mul(Ptr<Int> a, Ptr<Int> b, Ptr<Int> res){
    Int c;
    Ptr<Int> p = a + index() + (me()<<4);
    Ptr<Int> q = b + index() + (me()<<4);
    Ptr<Int> r = res + index() + (me()<<4);
    gather(p);
    Int aOld, bOld;

    For (Int i=0, i<35*16*qpu_nums, i=i+16*qpu_nums)
        c = 0;
        gather(q);
        receive(aOld);

        For (Int j=0, j<35*16*qpu_nums, j=j+16*qpu_nums)
            gather(q+j+16 * qpu_nums);
            receive(bOld);

            //split into smaller datatypes
            Int a_0 = aOld & MOD_BASE;
            Int a_1 = aOld >> HALF_BASE_SIZE;
            Int b_0 = bOld & MOD_BASE;
            Int b_1 = bOld >> HALF_BASE_SIZE;

            //Multiply small parts
            // a_0*b_0
            Int res_0 = (a_0 * b_0) & MOD_BASE;
            Int res_1 = ((a_0 * b_0) >> HALF_BASE_SIZE)
                        +((a_1 * b_0) & MOD_BASE);
            // a_1*b_0
            Int c_c = ((a_1 * b_0) >> HALF_BASE_SIZE)
                      + (res_1>> HALF_BASE_SIZE);
            res_1 = (res_1& MOD_BASE)+((a_0*b_1) & MOD_BASE);
            Int res_2 = c_c;
            Int res_3 = res_2>> HALF_BASE_SIZE;
            res_2 = res_2& MOD_BASE;
            // a_0*b_1
            c_c = ((a_0 * b_1)>> HALF_BASE_SIZE)
                  + (res_1>> HALF_BASE_SIZE);
            res_1 = res_1& MOD_BASE;
            res_2 = res_2 + c_c;
            res_3 = res_2>> HALF_BASE_SIZE;
            res_2 = res_2& MOD_BASE;
            // a_1*b_1
            res_2 = res_2 + ((a_1*b_1) & MOD_BASE);
            c_c = ((a_1 * b_1)>> HALF_BASE_SIZE)
                  + (res_2>> HALF_BASE_SIZE);
            res_2 = res_2& MOD_BASE;
            res_3 = res_3 + c_c;

            // calculate result in original base_size
            Int low15 = res_1 << HALF_BASE_SIZE;
            low15 = res_0 | low15;
            Int high15 = res_3 << HALF_BASE_SIZE;
            high15 = res_2 | high15;

            // end of split multiplication

            low15 = low15+res[i+j + (me()<<4)];
            Where (low15 >= BASE)
                high15++;
                low15 = low15 - BASE;
            End
            low15 = low15+c;
            Where (low15 >= BASE)
                high15++;
                low15 = low15 - BASE;
            End

            *(res+i+j + (me()<<4)) = low15;
            c = high15;
        End
        gather(p+i+16 * qpu_nums);
        receive(bOld);
        *(res+i+35*16 * qpu_nums + (me()<<4))=c;
    End
    receive(aOld);
}
```

Listing 6.1: GPU Multiprecision Multiplication

```
#define BASE_SIZE 30
#define HALF_BASE_SIZE 15
#define MOD_BASE 32767

void qpu_split_mul(Int& a, Ptr<Int> b, Int& high30, Int& low30)
   {
    Int a_0 = a & MOD_BASE;
    Int a_1 = a >> HALF_BASE_SIZE;
    Int b_0 = *b & MOD_BASE;
    Int b_1 = *b >> HALF_BASE_SIZE;

    // a_0*b_0
    Int res_0 = (a_0 * b_0) & MOD_BASE;
    Int res_1 = ((a_0 * b_0) >> HALF_BASE_SIZE)
                +((a_1 * b_0) & MOD_BASE);

    // a_1*b_0
    Int c_c = ((a_1 * b_0) >> HALF_BASE_SIZE)
              +(res_1>> HALF_BASE_SIZE);
    res_1 = (res_1& MOD_BASE)+((a_0*b_1) & MOD_BASE);
    Int res_2 = c_c;
    Int res_3 = res_2>> HALF_BASE_SIZE;
    res_2 = res_2& MOD_BASE;

    // a_0*b_1
    c_c = ((a_0 * b_1)>> HALF_BASE_SIZE)
          +(res_1>> HALF_BASE_SIZE);
    res_1 = res_1& MOD_BASE;
    res_2 = res_2 + c_c;
    res_3 = res_2>> HALF_BASE_SIZE;
    res_2 = res_2& MOD_BASE;

    // a_1*b_1
    res_2 = res_2 + ((a_1*b_1) & MOD_BASE);
    c_c = ((a_1 * b_1)>> HALF_BASE_SIZE)
          +(res_2>> HALF_BASE_SIZE);
    res_2 = res_2& MOD_BASE;
    res_3 = res_3 + c_c;

    // calculate result in original base_size
    low30 = res_1 << HALF_BASE_SIZE;
    low30 = res_0 | low30;
    high30 = res_3 << HALF_BASE_SIZE;
    high30 = res_2 | high30;
}
```

Listing 6.2: GPU Split Multiplication

```
void qpu_compare(Ptr<Int> a, Ptr<Int> b, Int *result){

    Ptr<Int> p =  a + index() + (me()<<4);
    Ptr<Int> q =  b + index();
    Int xOld, yOld;

    gather(p + (35*16 * qpu_nums));
    gather(p + (34*16 * qpu_nums));
    gather(q + (34*16 * qpu_nums));

    Int flag = 0;
    *result = 0;

    receive(xOld);

    Where (xOld > 0)
        *result = 1;
        flag = 1;
    End

    For (Int i = 0, i < 35*16 * qpu_nums, i = i+16 * qpu_nums)

        gather(p + (34*16 * qpu_nums - i + 16));
        gather(q + (34*16 * qpu_nums - i + 16));

        receive(xOld);
        receive(yOld);

        Where (flag == 0)
            Where (xOld > yOld)
                *result = 1;
                flag = 1;
            End
            Where (xOld < yOld)
                *result = -1;
                flag = 1;
            End
        End
    End
    receive(xOld);
    receive(yOld);
}
```

Listing 6.3: Multiprecision Comparison

Engines

```c
#include <stdio.h>
#include <string.h>
#include <openssl/engine.h>
#include <openssl/rsa.h>
#include <openssl/bn.h>

#include "gpu_methods.h"

#define NO_QPUs 12

static const char *engine_id = "GPU_RSA";
static const char *engine_name = "RSA performed partially on
    the GPU of the RPi";

static int public_encrypt(int flen, const unsigned char *from,
    unsigned char *to, RSA *rsa, int padding){
  char *filename = "message.txt";
  printf("Public RSA encryption of file %s\n", filename);

    // read in message
  char * buffer = 0;
  int length;
  FILE * f = fopen (filename, "rb");
  if (f){
    fseek (f, 0, SEEK_END);
    length = ftell (f);
    if(length != 128*16*NO_QPUs){
          printf("Message has the wrong size.");
      return 0;
    }
    fseek (f, 0, SEEK_SET);
    buffer = malloc (length);
    if (buffer){fread (buffer, 1, length, f);}
    fclose (f);
  }

  const BIGNUM *n;
  const BIGNUM *e;
  RSA_get0_key(rsa, &n, &e, NULL);
  char *module_string = BN_bn2dec(n);
  char *exponent_string = BN_bn2dec(e);

  // message
      char strm_array[16*NO_QPUs][310];
      for (size_t i = 0; i < 16 * NO_QPUs; i++) {
      BIGNUM * message = BN_bin2bn(buffer+i*128, sizeof(char)
 *128, NULL);
      char *message_string = BN_bn2dec(message);
      memset(strm_array[i], 0, sizeof(char)*309);
      strncpy(strm_array[i], message_string, sizeof(char)*309);
      }

  // message encryption
  gpu_RSA_encrypt(strm_array,
      exponent_string,
      module_string
      );

  unsigned char dest[128*16*NO_QPUs+1];
  memset(dest, 0, 128*16*NO_QPUs);

  for (size_t i = 0; i < 16 * NO_QPUs; i++) {
    BIGNUM *result;
          BN_dec2bn(&result, strm_array[i]);
    unsigned char result_string[309];
    memset(result_string, 0, sizeof(char)*309);
    BN_bn2bin(result, result_string);
    int string_length = BN_num_bytes(result);
    memcpy(dest+i*128+128-string_length, result_string,
 string_length*sizeof(char));
  }

  // write cipher to file
  FILE *fp;
  fp = fopen("gpu_result_cipher.txt", "wb");
        fwrite(dest, sizeof(char), sizeof(char)*128*16*NO_QPUs
 +1, fp);
        fclose(fp);
  printf("Result is written into: gpu_result_cipher.txt");
  return flen;

}

static int private_decrypt(int flen, const unsigned char *from,
    unsigned char *to, RSA *rsa, int padding){
  char *filename = "gpu_result_cipher.txt";
  printf("Private RSA decryption of file %s\n", filename);

    // read in message
  char * buffer = 0;
  int length;
```

```c
FILE * f = fopen ("gpu_result_cipher.txt", "rb");
if (f){
  fseek (f, 0, SEEK_END);
  length = ftell (f);
  if(length != 128*16*NO_QPUs+1){
        printf("Message has the wrong size.");
    return 0;
  }
  fseek (f, 0, SEEK_SET);
  buffer = malloc (length);
  if (buffer){fread (buffer, 1, length, f);}
  fclose (f);
}

const BIGNUM *n;
const BIGNUM *d;
RSA_get0_key(rsa, &n, NULL, &d);
char *module_string = BN_bn2dec(n);
char *exponent_string = BN_bn2dec(d);

// message
  char strm_array[16*NO_QPUs][310];
  for (size_t i = 0; i < 16 * NO_QPUs; i++) {
    BIGNUM * message = BN_bin2bn(buffer+i*128, sizeof(char)
 *128, NULL);
    char *message_string = BN_bn2dec(message);
    strncpy(strm_array[i], message_string, sizeof(char)*310);
    }

  // decrypt cipher
gpu_RSA_encrypt(strm_array,
    exponent_string,
    module_string
    );

unsigned char dest[128*16*NO_QPUs];
for (size_t i = 0; i < 16 * NO_QPUs; i++) {
  BIGNUM *result;
      BN_dec2bn(&result, strm_array[i]);
  unsigned char result_string[309];
  memset(result_string, 0, sizeof(char)*309);
  BN_bn2bin(result, result_string);
  int string_length = BN_num_bytes(result);
  memcpy(dest+i*128+128-string_length, result_string,
 string_length*sizeof(char));
}

```

```
    // Write result to file
  FILE *fp;
  fp = fopen("gpu_result_message.txt", "wb");
        fwrite(dest, sizeof(char), sizeof(char)*128*16*NO_QPUs,
    fp);
        fclose(fp);
  printf("Result is written into: gpu_result_message.txt");
  return flen;
}

static int bind(ENGINE *e, const char *id){
  int ret = 0;
  //set new RSA encryption and decryption methods
  RSA_METHOD *meth = RSA_meth_dup(RSA_get_default_method());
  if (!RSA_meth_set_pub_enc(meth, &public_encrypt)) {
    printf("set_meth failed");
    goto end;
  }
  if (!RSA_meth_set_priv_dec(meth, &private_decrypt)) {
    printf("set_meth failed");
    goto end;
  }

  if (!ENGINE_set_id(e, engine_id)) {
            fprintf(stderr, "ENGINE_set_id failed\n");
          goto end;
            }
  if (!ENGINE_set_name(e, engine_name)) {
              printf("ENGINE_set_name failed\n");
            goto end;
              }
  if (!ENGINE_set_RSA(e, meth)) {
    fprintf(stderr, "ENGINE_set_default_RSA failed\n");
    goto end;
  }
  ret = 1;
  end:
  return ret;
}

IMPLEMENT_DYNAMIC_BIND_FN(bind)
IMPLEMENT_DYNAMIC_CHECK_FN()
```

Listing 6.4: Engine for GPU excecution

```c
#include <stdio.h>
#include <string.h>
#include <openssl/engine.h>
#include <openssl/rsa.h>

#define NO_QPUs 12 // This is equivalent to number of 16*128
    bit messages

static const char *engine_id = "sequential_RSA";
static const char *engine_name = "RSA performed fully on the
    CPU";

static int public_encrypt(int flen, const unsigned char *from,
    unsigned char *to, RSA *rsa, int padding) {
      char * buffer = 0;
      int length;
      FILE * f = fopen ("message.txt", "rb");
      if (f){
        fseek (f, 0, SEEK_END);
        length = ftell (f);
        if(length != 128*16*NO_QPUs){
          printf("Message has the wrong size.");
          return 0;
        }
        fseek (f, 0, SEEK_SET);
        buffer = malloc (length);
        if (buffer){fread (buffer, 1, length, f);}
        fclose (f);
      }

      const RSA_METHOD *default_rsa = RSA_get_default_method();

        unsigned char dest[128*16*NO_QPUs];

        int len;
        for (size_t i = 0; i < 16*NO_QPUs; i++) {
      char msg[129];
      msg[128] = 0;
      memcpy(msg, buffer+i*128, sizeof(char)*128);
            len =  (RSA_meth_get_pub_enc(default_rsa)) (128,
    msg, to, rsa, 3);
            memcpy(dest+i*128, to, 128*sizeof(char));
        }

        FILE *fp;
        fp = fopen("openssl_result_cipher.txt", "wb");
        fwrite(dest, sizeof(char), sizeof(char)*128*NO_QPUs*16,
    fp);
        fclose(fp);
        printf("Result is written into: openssl_result_cipher.
   txt");

        return len;
}

static int private_decrypt(int flen, const unsigned char *from,
    unsigned char *to, RSA *rsa, int padding){
      char *buffer;
      int length;
      FILE * f = fopen ("openssl_result_cipher.txt", "rb");
      if (f){
        fseek (f, 0, SEEK_END);
        length = ftell (f);
        if(length != 128*16*NO_QPUs){
          printf("Message has the wrong size.");
          return 0;
        }
        fseek (f, 0, SEEK_SET);
        buffer = malloc (length);
        if (buffer){fread (buffer, 1, length, f);}
        fclose (f);
      }

      const RSA_METHOD *default_rsa = RSA_get_default_method();

        unsigned char dest[128*16*NO_QPUs];

        int len;
        for (size_t i = 0; i < 16*NO_QPUs; i++) {
      char msg[129];
      msg[128] = 0;
      memcpy(msg, buffer+i*128, sizeof(char)*128);
            len =  (RSA_meth_get_priv_dec(default_rsa)) (128,
   msg, to, rsa, 3);
            memcpy(dest+i*128, to, 128*sizeof(char));
        }

        FILE *fp;
        fp = fopen("openssl_result_message.txt", "wb");
        fwrite(dest, sizeof(char), sizeof(char)*128*16*NO_QPUs,
    fp);
        fclose(fp);
        printf("Result is written into: openssl_result_message.
   txt");
```

```

        return len;

  return 1;
}

static int bind(ENGINE *e, const char *id){
  int ret = 0;
  //set new RSA encryption and decryption methods
  RSA_METHOD *meth = RSA_meth_dup(RSA_get_default_method());
  if (!RSA_meth_set_pub_enc(meth, &public_encrypt)) {
    printf("set_meth failed");
    goto end;
  }
  if (!RSA_meth_set_priv_dec(meth, &private_decrypt)) {
    printf("set_meth failed");
    goto end;
  }

  if (!ENGINE_set_id(e, engine_id)) {
          fprintf(stderr, "ENGINE_set_id failed\n");
        goto end;
  }
  if (!ENGINE_set_name(e, engine_name)) {
            printf("ENGINE_set_name failed\n");
          goto end;
  }
  if (!ENGINE_set_RSA(e, meth)) {
    fprintf(stderr, "ENGINE_set_default_RSA failed\n");
    goto end;
  }
  ret = 1;
  end:
  return ret;
}

IMPLEMENT_DYNAMIC_BIND_FN(bind)
IMPLEMENT_DYNAMIC_CHECK_FN()
```
Listing 6.5: Engine with sequential openSSL RSA implementation

List of Figures

Bibliography

[ARM19] ARM: *ARM1176JZF-S Technical Reference Manual.* `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/index.html`. Version: 2019. – Online; accessed 23-May-2019

[BGV93] Bosselaers, Antoon ; Govaerts, René ; Vandewalle, Joos: Comparison of three modular reduction functions. In: *Annual International Cryptology Conference* Springer, 1993, S. 175–186

[Bro19a] Broadcom: *BCM2835 ARM Peripherals.* `https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf`. Version: 2019. – Online; accessed 23-May-2019

[Bro19b] Broadcom: *VideoCore IV 3D, Architecture Reference Guide.* `https://docs.broadcom.com/docs/12358545`. Version: 2019. – Online; accessed 23-May-2019

[BS13] Baktir, Selçuk ; Savaş, Erkay: Highly-parallel montgomery multiplication for multi-core general-purpose microprocessors. In: *Computer and Information Sciences III.* Springer, 2013, S. 467–476

[BSW15] Beutelspacher, Albrecht ; Schwenk, Jörg ; Wolfenstetter, Klaus-Dieter: *Moderne Verfahren der Kryptographie: Von RSA zu Zero-Knowledge.* Springer-Verlag, 2015

[Buc01] Buchmann, Johannes: *Einführung in die Kryptographie.* Bd. 3. Springer, 2001

[Dev19] Developer, ARM: *ARM11.* `https://developer.arm.com/ip-products/processors/classic-processors`. Version: 2019. – Online; accessed 23-May-2019

[Flea] Flensburg, Hochschule: *Modulare Exponentiation iterativ.* `http://www.inf.fh-flensburg.de/lang/krypto/algo/modexp2.htm`. – Online; accessed 26-November-2019

[Fleb] Flensburg, Hochschule: *Montgomery Multiplication.* `http://www.inf.fh-flensburg.de/lang/algorithmen/arithmetik/montgomery-multiplikation.htm`. – Online; accessed 26-November-2019

[FY14] Fadhil, Heba M. ; Younis, Mohammed I.: Parallelizing RSA algorithm on multicore CPU and GPU. In: *International Journal of Computer Applications* 87 (2014), Nr. 6

[G$^+$] Granlund, Torbjörn u. a.: *GNU Multiple Precision Arithmetic Library 6.1.2.* https://gmplib.org/. – Online; accessed 27-November-2019

[Her19] Hermitage, Herman: *VideoCore IV Programmers Manual.* https://github.com/hermanhermitage/videocoreiv/wiki/VideoCore-IV-Programmers-Manual. Version: 2019. – Online; accessed 23-May-2019

[KAF$^+$10] Kleinjung, Thorsten ; Aoki, Kazumaro ; Franke, Jens ; Lenstra, Arjen K. ; Thomé, Emmanuel ; Bos, Joppe W. ; Gaudry, Pierrick ; Kruppa, Alexander ; Montgomery, Peter L. ; Osvik, Dag A. u. a.: Factorization of a 768-bit RSA modulus. In: *Annual Cryptology Conference* Springer, 2010, S. 333–350

[KMVOV96] Katz, Jonathan ; Menezes, Alfred J. ; Van Oorschot, Paul C. ; Vanstone, Scott A.: *Handbook of applied cryptography.* CRC press, 1996

[LBPN12] Lara, Pedro ; Borges, Fábio ; Portugal, Renato ; Nedjah, Nadia: Parallel modular exponentiation using load balancing without precomputation. In: *Journal of Computer and System Sciences* 78 (2012), Nr. 2, S. 575–582

[Mag17] MagPi: *Sales soar and Raspberry Pi beats Commodore 64.* https://www.raspberrypi.org/magpi/raspberry-pi-sales/. Version: 2017. – Online; accessed 23-May-2019

[Mon85] Montgomery, Peter L.: Modular multiplication without trial division. In: *Mathematics of computation* 44 (1985), Nr. 170, S. 519–521

[Nay16] Naylor, Matthew: *QPULib.* https://github.com/mn416/QPULib. Version: 2016. – Online; accessed 23-May-2019

[NW15] Niederreiter, Harald ; Winterhof, Arne: *Applied number theory.* Springer, 2015

[Pau17] Pauls, Paul: *Parallelization of AES on Raspberry Pi GPU in Assembly.* Munich, Ludwig-Maximilians-Universität, Bachelorthesis, 2017. http://mnm-team.org/pub/Fopras/paul17/

[Pea96] Pearson, David: A parallel implementation of RSA. In: *Cornell University (July 1996)* (1996)

[Ras] RaspberryPi: *Products.* https://www.raspberrypi.org/products/. – Online; accessed 23-May-2019

[Ras19a] RaspberryPi: *The computer hardware.* https://www.raspberrypi.org/documentation/faqs/#hardware. Version: 2019. – Online; accessed 23-May-2019

[Ras19b] RaspberryPi: *Raspberry Pi hardware.* https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md. Version: 2019. – Online; accessed 23-May-2019

[rfc] *Public-Key Cryptography Standards (PKCS) No.1: RSA Cryptography; Specifications Version 2.1*. `https://tools.ietf.org/html/rfc3447#appendix-A.1.2`. – Online; accessed 26-November-2019

[Rix19] Rixen, Yannek: *SIMD processing of AES on the Raspberry Pi's GPU*. Munich, Ludwig-Maximilians-Universität, Bachelorthesis, 2019

[RSA78] Rivest, Ronald L. ; Shamir, Adi ; Adleman, Leonard: A method for obtaining digital signatures and public-key cryptosystems. In: *Communications of the ACM* 21 (1978), Nr. 2, S. 120–126

[Sch06] Schulz, Ralph-Hardo: Mathematische Grundlagen von öffentlichen Verschlüsselungsverfahren. (2006)

[Sev13] Severance, Charles: Eben upton: Raspberry pi. In: *Computer* 46 (2013), Nr. 10, S. 14–16

[Sha13] Sharoun, Assaid O.: Residue number system (RNS). In: *Poznan University of Technology Academic Journals. Electrical Engineering* (2013)

[SK15] Saxena, Sapna ; Kapoor, Bhanu: State of the art parallel approaches for RSA public key based cryptosystem. In: *arXiv preprint arXiv:1503.03593* (2015)

[Sta17] Stadelmann, Daniel: *Entwicklung einer OpenCL-Implementierung für die VideoCore IV GPU des Raspberry Pi*. 2017. – 7 –12 S.

[Tit00] Tittel, Inka: *Das RSA - Verfahren und die Implementierung in Mathematica*. 2000

[Wag13] Wagstaff, Samuel S.: *The joy of factoring*. Bd. 68. American Mathematical Soc., 2013

[Wik19] Wikipedia: *RaspberryPi, Specifications*. `https://en.wikipedia.org/wiki/Raspberry_Pi`. Version: 2019. – Online; accessed 23-May-2019

[ZG15] Zur Gathen, Joachim v.: *CryptoSchool*. Springer, 2015