Elliptic Curve point multiplication on GPUs

Samuel Antão, Jean-Claude Bajard, Leonel Sousa

Elliptic Curve Point Multiplication on GPUs

Samuel Antão
Instituto Superior Técnico/INESC-ID
Technical University of Lisbon
Lisbon, Portugal
Email: sfan@sips.inesc-id.pt

Jean-Claude Bajard
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
Paris, France
Email: jean-claude.bajard@lip6.fr

Leonel Sousa
Instituto Superior Técnico/INESC-ID
Technical University of Lisbon
Lisbon, Portugal
Email: las@sips.inesc-id.pt

***Abstract*—Acceleration of cryptographic applications on Graphical Processing Units (GPUs) platforms is a research topic with practical interest, because these platforms provide huge computational power for this type of applications. In this paper, we propose parallel algorithms for Elliptic Curve (EC) point multiplication in order to compute EC cryptography on GPUs. The porposed approach relies in using the Residue Number System (RNS) to extract parallelism on high precision integer arithmetic. For the best of our knowledge, this is the first implementation proposed for computing the EC point multiplication on GPUs supported on RNS. Results suggest a maximum throughput of 9990 EC multiplications per second and minimum latency od 24.3 ms for a 224-bit underlying field, for an Nvidia 285 GTX GPU. We present performances up to an order of magnitude better in latency and 54 % in throughput regarding other approaches reported in the related art.**

I. Introduction

Recently, Graphical Processing Units (GPUs) have been increasingly used in several applications as a powerful accelerator for high computational demanding applications [1]. The huge computational power of a single GPU allied with its low cost regarding other dedicated accelerator solutions, mainly because of mass production for the gaming market, is the main reason for the interest of using GPUs for General Purpose applications (GPGPU) [1].

Applications that take advantage of the GPU computing power can be found in different fields, namely physics, biology and cryptography [2], etc... These different applications may have different properties that turn them more or less suitable for implementations on GPUs. The following summarizes these properties.

- Low data dependencies, allowing for easier parallelization of the algorithm, by computing parallel instances over independent data sets.
- Regular description, allowing for the identification of different single computation flows among the algorithm, enhancing the parallelization and scalability for data sets with different sizes.
- Low memory accesses, taking advantage of the huge GPU processing power reducing stalling effects waiting for data from memory.
- High computation over small input/output data, reducing the impact of the communication delays in the overall performance.

Considering the above properties, despite the different application fields, implementations supported on GPUs have similar challenges that consist in designing/redesigning algorithms to use extensive parallel processing control flows on independent and small data sets.

In this paper we get through the GPU implementation of asymmetric cryptography supported on Elliptic Curve (EC). EC cryptography arose as a promising competitor to the widely used Rivest-Shamir-Adleman (RSA), due to the reduced key size required to provide the same security. The algorithms used in EC cryptography do not meet all the requirements for a direct and efficient implementation on GPUs. In particular, the EC point multiplication with a scalar (private key), which is the most demanding operation in EC cryptography, is constructed on several successive steps where the scalar is browsed, and there are data dependencies between all the successive steps. On the other hand, EC point multiplication represents a demanding computation supported on small input data sets (private and public key are usually represented with up to three 521-bit integers), which is an interesting property for the EC computation.

In order to overcome the inefficiency due to the data dependencies, we propose in this paper to use the Residue Number System (RNS) approach [3]. RNS is an alternative representation that splits the traditional integer into several smaller residues for a established basis. With the operands sharing a common RNS representation, the computation can be performed in parallel on the correspondent residues. Hence, RNS representation is an attractive approach to enhance the parallelization of algorithms. We combine the RNS representation with the Montgomery ladder algorithm for EC point multiplication in order to obtain a high performance accelerator for EC cryptography supported on GPUs. For the best of our knowledge, the presented work is the first presenting a solution supported on GPUs and RNS to accelerate EC cryptography.

Algorithm 1 Montgomery Ladder Algorithm

Require: EC point $G \in E(a, b, p)$, k-bit scalar s.
Ensure: $P = sG \in E(a, b, p)$.
1: $P = G$;
2: $Q = 2G$;
3: **for** $i = k - 2$ down to 0 **do**
4: **if** $s_i = 1$ **then**
5: $P = P + Q$;
6: $Q = 2Q$;
7: **else**
8: $Q = P + Q$;
9: $P = 2P$;
10: **end if**
11: **end for**

II. EC AND RNS BACKGROUND

This section provides the background on EC and RNS arithmetic, as well as the application of both for EC point multiplication.

A. EC cryptography over $GF(p)$

An EC $E(a, b, p)$ over $GF(p)$, with p a prime, is a set composed by a point at infinity $\mathcal{O}$ and the points $P_i = (x_i, y_i) \in GF(p) \times GF(p)$ that comply the following equation:

$$y_i^2 = x_i^3 + ax_i + b,\ a, b \in GF(p). \tag{1}$$

In order to obey smoothness conditions the parameters a and b obey $-\left(4a^3 + 27b^2\right) \neq 0 \bmod p$. By establishing the addition and doubling operation over the EC points, and by applying it recursively, it is possible to obtain the multiplication of a point P by a scalar s as $Q = P+P+\ldots+P = sP$ (Elliptic Curve Discrete Logarithm Problem).

The EC point addition and doubling are performed with operations over the underlying field $GF(p)$ applied to the points' coordinates. It is known that it is possible to obtain the x_R coordinate of a point addition $R = P + Q$ knowing the x coordinates of P, Q, and $P - Q$. This observation motivated the proposal of a double and add algorithm that does not require the y coordinate, known as the Montgomery Ladder for EC [4]. The Algorithm 1 details the Montgomery Ladder algorithm behavior for obtaining $P = sG$, where s is a scalar with size k (the most significant bit of s, $s_{k-1} = 1$).

The operations over the coordinates, used to obtain the EC point operations, require modular inversions, which is a computationally demanding operation over a finite field. In order to avoid a large number of inversions, the traditional (affine) representation of the coordinates is replaced by a projective representation. The projective representation introduces an extra coordinate Z. To commute between standard projective (X) and affine representation (x) of a coordinate, the following correspondence holds: $x \Leftrightarrow X/Z$. The projective versions of point addition and doubling to support Algorithm 1 are the following (for $s_i = 0$) [4]:

$$\begin{aligned} X_{P+Q} &= -4bZ_PZ_Q(X_PZ_Q + X_QZ_P) \\ &\quad + (X_PX_Q - aZ_PZ_Q)^2, \\ Z_{P+Q} &= x_G(X_PZ_Q - X_QZ_P)^2; \\ X_{2P} &= (X_P^2 - aZ_P^2)^2 - 8bX_PZ_P^3, \\ Z_{2P} &= 4Z_P(X_P^3 + aX_PZ_P^2 + bZ_P^3). \end{aligned} \tag{2}$$

For $s_i = 1$, the formula used to double P, is used to double Q instead. As it can be concluded from Algorithm 1, each iteration of the cycle has data dependencies from the previous iteration, which is an extra difficulty towards the design of a parallel algorithm. Moreover, the sizes of the prime p for standardized EC are 192, 224, 256, 384, and 521 bits [5]. Thus, the operations over the coordinates would require a datapath of the same size. In order to adapt the coordinates size to the GPU processing datapath, we propose use an RNS representation of the field elements. This representation allows to parallelize the field operations on the GPU, providing the required degree of parallelization towards an efficient implementation.

B. RNS Overview

The RNS has its fundamentals in the Chinese Remainder Theorem (CRT), which states that for a given basis B_n consisting of n coprime integers $(m_1, m_2, ..., m_n)$ there is a unique representation for the integer $X < M$ in the form $x_j = X \bmod m_j$, with $1 < j < n$ and $M = \prod_{i=1}^{n} m_i$ the dynamic range of the basis B_n. Having two integers X and Y in RNS representation, an operation $Z = X \otimes Y \bmod M$ is equivalent to $z_j = x_j \otimes y_j \bmod m_j$, where $\otimes$ represents addition, subtraction, or multiplication. In general, choosing a basis B_n such that $X \otimes Y < M$, the RNS representation can be used to perform any of the aforementioned operations in the traditional representation system. As suggested above, the advantage of using the RNS representation is the possibility of split the computation in n parallel flows (now onwards called channels), each one operating modulo a different m_i. The conversion from binary to the RNS representation of an integer X can be accomplished by computing the residues x_j directly and in parallel. For the opposite conversion, there are two equivalent methods that can be used: CRT and Mixed Radix System (MRS).

Due to the recursive nature of MRS based conversion, it is not suitable for efficient GPU implementations [2]. The alternative method relies on the CRT definition for computing the binary representation:

$$X = \sum_{i=1}^{n} \xi_i M_i \bmod M,\ \xi_i = \left| \frac{x_i}{M_i} \right|_{m_i}, \tag{3}$$

where $M_i = M/m_i$ and $|\cdot|_{m_i}$ denotes an operation modulo m_i. In order to avoid a reduction modulo a large number M, the expression in (3) can be rewritten as:

$$X = \sum_{i=1}^{n} \xi_i M_i - \alpha M, \ \alpha < n. \tag{4}$$

In (4) the operation modulo M is replaced by the subtraction of a multiple of M dependent of α. Two main methods have been used to compute the constant α. An extra moduli m_e can be established and all the operations performed not only on the basis B_n, but also on this extra moduli (Shenoy et. al. [6]). Hence, the RNS representation of the integer X is $(x_1, x_2, \ldots, x_n, x_e)$. Applying the reduction modulo m_e to (4), it is obtained:

$$x_e = \left| \sum_{i=1}^{n} \xi_i M_i \right|_{m_e} - |\alpha M|_{m_e}. \tag{5}$$

Rewriting (5), α can be obtained as:

$$|\alpha|_{m_e} = \left| \sum_{i=1}^{n} \xi_i M_i - x_e \right|_{m_e} \left| M^{-1} \right|_{m_e}. \tag{6}$$

Since $\alpha < n$, choosing $m_e \geq n$ results in $\alpha = |\alpha|_{m_e}$.

Other possible method to compute α involves a successive fixed point approximation approach (Kawamura et. al. [7]). In this method, (4) is rewritten as:

$$\sum_{i=1}^{n} \frac{\xi_i}{m_i} = \alpha + \frac{X}{M}, \tag{7}$$

and observe that knowing $X < M$ then $\frac{X}{M} < 1$:

$$\alpha = \left\lfloor \sum_{i=1}^{n} \frac{\xi_i}{m_i} \right\rfloor. \tag{8}$$

Since (8) requires costly divisions by m_i, an approximation ($\hat{\alpha}$) to this expression is suggested:

$$\hat{\alpha} = \left\lfloor \sum_{i=1}^{n} \frac{\text{trunc}(\xi_i)}{2^r} + \beta \right\rfloor, \tag{9}$$

where r is such that $2^{r-1} < m_i \leq 2^r$, and trunc(ξ_i) sets the $q - r$ least significant bits of ξ_i to zero, with $q < r$. The parameter β is a corrective term that should be carefully chosen such that $\alpha = \hat{\alpha}$. The authors state a set of inequalities that allow to chose good values for β, supported on the maximum initial approximation errors, respectively:

$$\epsilon = \max_i \left(\frac{2^r - m_i}{2^r} \right), \delta = \max_i \left(\frac{\xi_i - \text{trunc}(\xi_i)}{m_i} \right). \tag{10}$$

If a value of β is chosen such that $0 \leq n(\epsilon + \delta) \leq \beta < 1$ and $0 \leq X < (1 - \beta)M$, then $\alpha = \hat{\alpha}$.

III. RNS Montgomery Multiplication

In order to perform EC arithmetic with RNS, we should not only provide a method for add and multiply, but also to reduce. The Montgomery Modular Multiplication is an efficient method that allows replacing the reduction modulo an integer N (usually a prime) by a reduction modulo $R = 2^k$, which can be very easily accomplished operating on the binary representation of an integer [8]. This method employs a different domain for a field element X, as $\bar{X} = XR \bmod N$ and computes $\bar{Z} = \bar{X}\bar{Y}R^{-1} \bmod N$.

The rational of the Montgomery modular multiplication algorithm can also be applied to an RNS version, but using the RNS representation it is no longer easy to reduce modulo a power of two. Instead, defining a basis B_n with dynamic range M such that $M > N$ and $gcd(M, N) = 1$, it is easy to reduce an element represented with the basis B_n modulo the dynamic range M. It only requires to reduce modulo m_i in each one of the RNS channels. Hence, the RNS Montgomery Multiplication version compute $\bar{Z} = \bar{X}\bar{Y}M^{-1} \bmod N$. One of the drawbacks of the RNS version is that it is not possible to represent M^{-1} in B_n. Thus it required to set another basis $\tilde{B}_n$ with dynamic range $\tilde{M}$ such that $\tilde{M} > M$ and $gcd(\tilde{M}, M) = 1$.

With the modified Montgomery modular multiplication algorithm we need to compute $U = (T + QN)M^{-1}$, where $T = \bar{X}\bar{Y}$. It is easy to see that if we guarantee $T + QN \equiv 0 \bmod M$ then $U \equiv \bar{X}\bar{Y}M^{-1} \bmod N$. To ensure this, we need to compute in the RNS channel i for the basis B_n:

$$0 = t_i + q_i n_i \bmod m_i \Leftrightarrow q_i = -t_i(n_i)^{-1}_{m_i} \bmod m_i. \tag{11}$$

After computing the value of Q we can use one of the methods referred in Section II-B to convert the value of Q to the basis $\tilde{B}_n$. In this basis, for each RNS channel i we compute:

$$\tilde{u}_i = (\tilde{t}_i + \tilde{q}_i \tilde{n}_i)(M)^{-1}_{\tilde{m}_i} \bmod \tilde{m}_i. \tag{12}$$

Afterwards, we convert the result U from basis $\tilde{B}_n$ to basis B_n (note that $U < 2N$, since $T < MN$, $QN < MN$, and $N < M$). However, when computing the algorithms based on the Montgomery multiplication, we need the intermediary results to be bounded but not perfectly reduced. Thus, considering $Z = U$ and applying Z as input in further multiplications we will always have $U < 2N$ and the multiplication result will be correct modulo N, since $M > 4N$.

In the RNS version of the Montgomery multiplication algorithm, the most costly steps are the conversion between basis, base extension, since all the other steps correspond to independent operations in each RNS channel. Addressing this problem, an offset during the conversion of Q from B_n to $\tilde{B}_n$ can be allowed [3]. As (4) suggests, the conversion from an RNS basis implies the computation of a constant α that multiplied by the dynamic range M correct an offset

in the conversion to maintain the result bounded by M. In [3] it is suggested to use the CRT conversion without the correction term introduced by α during the first conversion (B_n to $\tilde{B}_n$). With this approach, after the conversion we obtain a value of $\hat{Q} = Q + \alpha M$ that contains an offset. With this offset, the value of U is given by:

$$\hat{U} = (T + \hat{Q}N)M^{-1} = (T + QN)M^{-1} + \alpha N. \quad (13)$$

Since $\alpha < n$, $\hat{U} < (n+2)N$. In order to feed this result in subsequent multiplications we must chose M such that $M > (n+2)^2 N$, since this condition complemented with the condition $\bar{X}\bar{Y} = T < NM$ ensures $\hat{U} < (n+2)N$. In summary, with this method, we are able to avoid the computation of α during one conversion while keeping the multiplication result bounded by an acceptable value (note that $n << N$).

IV. EC Parallel Algorithm for GPU

The Algorithm 1 to multiply a point G by the scalar s, can be split into two sections, the initialization section, and the loop controlled by the scalar. The initialization computes $P = G$ and $Q = 2G$, which can be performed computing:

$$\begin{aligned} X_P &= x_G, \quad (14)\\ Z_P &= 1;\\ X_Q &= (x_G^2 + 3)^2 - 8bx_G,\\ Z_Q &= 4(x_G^3 - 3x_G + b). \end{aligned}$$

Each iteration of the main loop computes the operation in (2) an appropriate scheduling of the operations presented in (2). The schedule in Table I can be adopted to perform the EC point multiplication; this schedule is only for $s_i = 0$. The loop schedule for $s_i = 1$ can be obtained by commutating P and Q in the loop section of Table I. The scheduling is divided in multiplications and additions sets. Each multiplication set is composed of several independent field multiplication operations, and each addition set is composed by field additions/subtractions and multiplications with small constants.

A. General Purpose Processing on GPUs

Tesla is a typical architecture of a GPU which consists of several general purpose scalar processors grouped in multiprocessor cores that allows for general purpose processing [9]. Furthermore, NVIDIA Compute Unified Device Architecture (CUDA) allows programmers to easily program NVIDIA GPUs for general purpose processing.

In order to exploit the parallel computation capabilities of GPUs, CUDA provides different units of parallelism. The smallest unit is the thread, each multiprocessor core is able to run up to 32 simultaneous threads, which have their own register file. A group of threads that run simultaneously in a multiprocessor core is called warp, and the way the threads in a warp are executed obeys a SIMD flow. The threads are organized in a higher level parallelism abstraction unit called block. Different blocks are independent and can run in parallel by using the several multiprocessor cores. A group of blocks that is executed in parallel in the existent multiprocessor cores is called a grid. The way a sequential algorithm can be parallelized in threads and blocks depends on the multiprocessor local resources (shared memory, cache, registers availability) and on the processing dependencies.

In each multiprocessor there is a 16 Kbytes shared memory, and a 8 Kbytes symbols cache that can be used for read only data (constants). Despite possible conflicts between different threads, the memory inside a multiprocessor can be accessed in the same amount a register can. There is also a global memory, where the initialization data is written by the GPU host. The global memory has a higher accessing latency (40 to 60 times higher than the shared memory latency [10]), thus its utilization and accessing patterns should be judiciously set to avoid long stall periods by a multiprocessor.

Each scalar processor has pipelined floating point adders and multipliers, which can be used for integer arithmetic. With CUDA a 24-bit multiplication is performed in the same time than other 32-bit integer operations, such as addition. A 32-bit integer multiplication is 4 times slower. Moreover, it is not possible to obtain the 16 most significant bits of a 24-bit multiplication, only the 32 least significant bits are available.

B. Parallel Algorithms, type I and type II

Let us assume that the GPU inputs and outputs are in RNS format, and that input data is already in the Montgomery domain. These assumptions are supported on the fact that the computational demanding core of the algorithm is in the loop for computing the resulting x coordinate. A general EC standardized by NIST is considered for a prime number with 224 bits, where $a = -3$ [5].

Regarding the proposed schedule, for each EC point multiplication 11 variables are required to store intermediate data. To perform modular operations among the RNS channels, the complete precision of a multiplication has to be available in order to perform reduction. Hence, since we can only obtain a 32-bit result from a multiplication we must use input operands of 16-bits. Thus, each RNS channel can compute 16-bit arithmetic modulo a basis element of the form $2^k - c$, with $k = 16$ and $c \geq 0$. The required number on RNS channels depends on the range to represent a field element. The number of EC point multiplications performed in parallel in each multiprocessor depends on the available memory required to efficiently store intermediate data and constants. Several variables have to be loaded and stored in each EC point multiplication steps for each RNS channel. Thus, with this extensive memory transactions, global memory should only be used to store the input data

Table I
OPERATIONS SCHEDULING (FOR $s_i = 0$).

Init.	mult. 1	$A{=}x_G^2$ $B{=}bx_G$	Loop	mult. 1	$A{=}X_PZ_Q$ $B{=}X_QZ_P$ $C{=}X_PX_Q$ $D{=}Z_PZ_Q$ $E{=}X_P^2$ $F{=}Z_P^2$ $H{=}bZ_P$	mult. 2	$D{=}DX_Q$ $X_Q{=}C^2$ $Z_Q{=}Z_Q^2$ $A{=}A^2$ $B{=}FH$ $E{=}EX_P$ $F{=}X_PF$	mult. 3	$D{=}bD$ $Z_Q{=}xZ_Q$ $B{=}X_PB$ $Z_P{=}Z_PE$
	add.	$C{=}A+3$								
	mult. 2	$C{=}C^2$ $A{=}x_GA$								
	add. 2	$X_Q{=}C-8B$ $Z_Q{=}4(A-3x_G+b)$		add. 1	$Z_Q{=}A-B$ $X_Q{=}A+B$ $C{=}C+3D$ $A{=}E+3F$	add. 2	$E{=}E+B-3F$	add. 3	$X_Q{=}X_Q-4D$ $X_P{=}A-8B$ $Z_P{=}4Z_P$	

and the final results. Hence, the number of multiplications handled by each multiprocessor is constrained by the size of the shared memory.

In the following subsections we propose two types of algorithms. The type I algorithm is supported on a dynamic range that allows for the computation of a complete loop in Table I prior to the reduction. The dynamic range considered in type II algorithm only supports the computation of a set of multiplications and a set of additions in Table I, prior to a reduction. In both algorithms, we store the projective coordinates of the input point in global memory, and consider the scalar a constant stored in constant memory.

1) Type I Parallel Algorithm: Algorithm Type I obtains the resulting projective coordinates X_P, Z_P, X_Q, and Z_P of the complete loop in Table I that are then multiplied by the unity M mod N in the Montgomery domain, using the method introduced in Section II, in order to obtain the partial reduced values. We allow an offset in the base extension from B_n to $\tilde{B}_n$ [3]. Considering the results in [2] that compare the Shenoy et. al. and Kawamura et. al. methods on a GPU to perform the base extension from $\tilde{B}_n$ to B_n, we follow the former one (based on (6)) because it achieves better performance.

Let us find the required dynamic range to compute (2) without having to reduce intermediary results mod M. As explained in Section III, the maximum value of an output, in the base extension method, is smaller than $u = (n+2)N$, where n is the number of channels and $N = p$, with p the prime that defines the underlying field $GF(p)$. In order to avoid obtaining negative values that would require reduction mod M (in (2) there are subtractions), new terms are added to (2):

$$\begin{aligned} X_{P+Q} &= 8u^5 - 4bZ_PZ_Q(X_PZ_Q + X_QZ_P) \\ &\quad + (X_PX_Q + 3Z_PZ_Q)^2, \\ Z_{P+Q} &= x_G(X_PZ_Q + u^2 - X_QZ_P)^2; \\ X_{2P} &= (X_P^2 + 3Z_P^2)^2 + 8u^5 - 8bX_PZ_P^3, \\ Z_{2P} &= 4Z_P(X_P^3 + 3u^3 - 3X_PZ_P^2 + bZ_P^3). \end{aligned} \tag{15}$$

Note that the inclusion of such constants do not change the result mod N. With these new terms the maximum value of a projective coordinate is:

$$\begin{aligned} [X_{P+Q}]_{\max} &< 8u^5 + 16u^4, \\ [Z_{P+Q}]_{\max} &< 4u^5; \\ [X_{2P}]_{\max} &< 8u^5 + 16u^4, \\ [Z_{2P}]_{\max} &< 4u^5 + 16u^4. \end{aligned} \tag{16}$$

Therefore, the dynamic range M has to comply $M \geq 8u^5 + 16u^4$. We define a moduli set composed of $2n$ elements of the form $2^{16} - c_i$, with c_i an odd number, $c_i < c_j$ for $i < j$, and $c_0 = 1$; an element $2^{16} - c_j$ is added to the set once it is relative prime with all the other elements $2^{16} - c_i$ with $0 < i < j$. The basis B_n is obtained by selecting the elements $2^{16} - c_i$ with even i, and the basis $\tilde{B}_n$ with the elements with odd i. With this we assure $M < \tilde{M}$. The required number of elements in order to comply $M \geq 8u^5 + 16u^4$ in each base was found to be $n = 73$. The extra element used to compute (6) is 2^k such that $2^k > n$; note than any number 2^k is relative prime to B_n and $\tilde{B}_n$.

Since we are also interested in low latency, an EC point multiplication is accomplished in a single block of threads, which runs in a single multiprocessor. The block has associated n threads, and each thread has one correspondent element of B_n (m_i) and $\tilde{B}_n$ ($\tilde{m}_i$). There is one thread that is associated with an element $m_e = 2^k$, being the thread responsible for computing (6) and the required operations mod m_e. Each thread performs arithmetic mod an element of the bases B_n and $\tilde{B}_n$. Algorithm Type I presents the flow of the computation held by each thread in a block, where element V represents any of the projective coordinates X_P, Z_P, X_Q, and Z_Q resulting from a single loop iteration, and $W = M$ mod N. The computation presented in Algorithm Type I uses the following precomputed constants:

- M mod B_n; N mod [B_n,m_e,$\tilde{B}_n$]; N^{-1} mod B_n;
- M_i mod B_n; $\tilde{M}_i$ mod $\tilde{B}_n$; M^{-1} mod [m_e,$\tilde{B}_n$];
- $\tilde{M}^{-1}$ mod m_e; $W = (M \bmod N)$ mod [B_n,m_e,$\tilde{B}_n$].

Algorithm 2 Type I (main loop computation for $n = 73$).

Threads involved	Computation for thread i
1 to n	Table I scheduling (mod m_i) and (mod $\tilde{m}_i$)
1 to n	$v_i = -w_i v_i (n_i)_{m_i}^{-1} \lvert M_i \rvert_{m_i}^{-1} \bmod m_i$
Synchronization	
1 to n	$\tilde{v}_i = \left(\left\lvert \sum_{j=1}^{n} v_j M_j \right\rvert_{\tilde{m}_i} \tilde{n}_i + \tilde{v}_i \tilde{w}_i \right) \lvert M \rvert_{\tilde{m}_i}^{-1}$
1 to n	$\tilde{a}_i = \tilde{v}_i \left\lvert \tilde{M}_i \right\rvert_{\tilde{m}_i}^{-1}$
1	Table I scheduling (mod m_e)
1	$v_e = \left(\left\lvert \sum_{j=1}^{n} v_j M_j \right\rvert_{m_e} n_e + v_e w_e \right) \lvert M \rvert_{m_e}^{-1}$
1	$\alpha_W = \left(\left\lvert \sum_{j=1}^{n} \tilde{v}_j \tilde{M}_j \right\rvert_{m_e} - v_e \right) \left\lvert \tilde{M} \right\rvert_{m_e}^{-1}$
Synchronization	
1 to n	$v_i = \left\lvert \sum_{j=1}^{n} \tilde{v}_j \tilde{M}_j - \alpha_W \tilde{M} \right\rvert_{m_i}$

C. Type II Parallel Algorithm

The latency value for the Type I Algorithm would not be practical in real applications (latency $> 1s$), as observed from Table II. The main drawbacks of the Type I Algorithm are related with the number of synchronizations, the number and size of the divergent code sections (sections computed in series), and with the complexity of computing the result $\mu = \left\{ M_i \bmod \tilde{m}_i,\ \tilde{M}_i \bmod m_i,\ M_i \bmod m_e,\ \tilde{M}_i \bmod m_e \right\}$ every time they are needed. Synchronization barriers and divergence cannot be removed once (6) has to be computed, which is a dependency for the next steps. The computation of μ exhibits quadratic complexity, thus trading the required dynamic range (RNS channels) by the number of times the base extension algorithm is called may improve the performance. Replacing this computation by table look-ups will not be efficient since we would require $2n(n+1)$ entries, (for 16-bit entries, 21,608 bytes are required, which exceeds the shared memory capacity)

In Algorithm Type II only a multiplication and addition set of the schedule presented in Table I is are computed in each base extension. Since the multiplications in the set are independent they can be computed simultaneously, and the result μ is computed only once and shared by the different running multiplications threads.

Since we are interested in supporting also an addition in one base extension, performing an analysis like the one suggested in (15) for Table I, with u the smallest multiple of N higher than an addition input, the following range has to be achieved:

- for Z_Q, in the initialization step, add. 2, result is bounded by $4(u + 4N)$;
- for X_P, in the loop step, add. 4, result is bounded by $9u$.

Algorithm 3 Alternative reduction algorithm

Require: $z' = z'_H 2^{16} + z'_L$, m, c.
Ensure: $z = z' \bmod m$.
1: **while** $z' > 2m - 1$ **do**
2: $\quad z' = cz'_H + z'_L$;
3: **end while**
4: $z = min(z', z' - m)$;
5: **return** z

By using $u = (n+2)N$ (see Section II (13), where n is the number of RNS channels, we get that $9u > 4(u + 4N)$ for $n > 1$. Since a minimum $n = 14$ is required to represent a field element in 16-bit channels, the precision is bounded by $9u$. Considering that the multiplication inputs are bounded by $9u$, setting $M > (9(n+2))^2 N$ will bound the multiplication output (addition input) to $u = (n+2)N$, which is the bound from where we depart. The condition $M > (9(n+2))^2 N$ results in $n = 15$.

The reduction operation over the RNS channels using the C '%' operation is known to be very demanding in the GPU platforms; evaluation of a simple program that performs several logical operations mod a basis element using '%' in the GPU shows that 73% of the time is consumed computing the reduction. Since a basis element has the form $m = 2^{16} - c$, when we compute an operation we get a result $z' = z'_H 2^{16} + z'_L$, and we want to obtain $z = z' \bmod m$. This operation can be accomplished recurring to Algorithm 3. Note that the step 4 of Algorithm 3 return the correct result since we are considering unsigned arithmetic. The maximum number of iterations in the loop is constrained by the maximum value of c. In the adopted basis, for computing $(m-1)^2$ only up to 2 iterations are required. This bounds the required number of iterations to reduce after a multiplication, which avoids the evaluation of loop conditions. Following the same idea, for an addition $z = (x + y) \bmod m$ we can compute only $z = min(x + y, x + y - m)$ and for a subtraction $z = (x - y) \bmod m$ we can compute $z = min(x-y, x-y+m)$. The computation of the minimum corresponds to only one GPU instruction, and allows to avoid conditions that can potentially create divergent sections of the program, thus serialization of the computation.

Considering Algorithm Type I, there are steps that apply different constants that can be more efficiently applied if merged into only one. These changes allow to save memory and computation resources. The following summarizes the constants replacement:

- New constant $r_i = \lvert n_i M_i \rvert_{m_i}^{-1}$;
- New constant $s_i = \left\lvert \tilde{n}_i \left\lvert \tilde{M}_i \right\rvert_{\tilde{m}_i}^{-2} \right\rvert_{\tilde{m}_i}$;
- New constant $t_i = \left\lvert \lvert M \rvert_{\tilde{m}_i}^{-1} \left\lvert \tilde{M}_i \right\rvert_{\tilde{m}_i} \right\rvert_{\tilde{m}_i}$;

Algorithm 4 Type II, ver. 2, loop k computation ($n = 15$).

Threads involved	Computation for thread i
1 to n	$z_i = -x_i y_i r_i \bmod m_i$
	Synchronization
1 to n	$\tilde{\xi}_{z_i} = \left(\left\lvert \sum_{j=1}^{n} z_j M_j \right\rvert_{\tilde{m}_i} s_i + \tilde{\xi}_{x_i} \tilde{\xi}_{y_i} \right) t_i$
1	$z_e = \left(\left\lvert \sum_{j=1}^{n} z_j M_j \right\rvert_{m_e} n_e + x_e y_e \right) \lvert M \rvert_{m_e}^{-1}$
1	$\alpha_z = \left(\left\lvert \sum_{j=1}^{n} \tilde{\xi}_{z_j} \tilde{M}_j \right\rvert_{m_e} - z_e \right) \left\lvert \tilde{M} \right\rvert_{m_e}^{-1}$
1	Compute set add. k operations mod m_e
	Synchronization
1 to n	$z_i = \left\lvert \sum_{j=1}^{n} \tilde{\xi}_{z_j} \tilde{M}_j - \alpha_z \tilde{M} \right\rvert_{m_i}$
1 to n	Compute set add. k operations mod $[m_i/\tilde{m}_i]$

- Remove constants $\lvert n_i \rvert_{m_i}^{-1}$, $\lvert M_i \rvert_{m_i}^{-1}$, $\left\lvert \tilde{M}_i \right\rvert_{\tilde{m}_i}^{-1}$, and $\lvert M \rvert_{\tilde{m}_i}^{-1}$;
- The operands $\tilde{x}$ in the basis $\tilde{B}_n$ are stored as $\tilde{\xi}_x = \tilde{x} \left\lvert \tilde{M}_i \right\rvert_{\tilde{m}_i}^{-1}$. Note that the results of this basis are not needed to retrieve the final results, thus the algorithm output remains in the same format.

The Kawamura et. al method for computing α in (4) can be used instead of the Shenoy et. al method by computing (9) as explained in Section II-B. In this paper we have to choose $r = 16$. We have to compute the lower bound for β from the approximation error given by $\Delta = n(\epsilon + \delta)$, where n is the number of channels and:

$$\epsilon = \max\left(\frac{2^r - k}{2^r}\right), \delta = \max\left(\frac{2^{r-q} - 1}{k}\right). \tag{17}$$

with k any element of B_n or $\tilde{B}_n$, and q the number of bits that we truncate in the approximation. Then we choose a value β such that $\Delta \leq \beta < 1$ and we must assure $X < (1 - \beta)\tilde{M}$, where X is the multiplication result bounded by $X < (n + 2)N$, as previously pointed. In order to have as small as possible dynamic ranges we are interested in small β. Choosing $q = 9$, we get $\Delta = 0.063$, and we can choose $\beta = \Delta$. Since $(1 - \beta) > 1/2$ and we already set $M > (9(n+2))^2 N$, the condition $X < (1-\beta)\tilde{M}$ hold. We can define a constant $\Phi = \lceil 2^r \beta \rceil$ and we can compute:

$$\Gamma = \sum_{i=1}^{n} \text{trunc}(\xi_i) + \Phi, \tag{18}$$

After computing Γ we can obtain $\alpha = \Gamma / 2^r$, which correspond to a 16-bit right shift. With this method we avoid the computations over the basis m_e and α is obtained as:

$$\alpha_z = \left(\sum_{i=1}^{n} \tilde{\xi}_{z_i} + \Phi \right) / 2^{16}. \tag{19}$$

Table II
DIFFERENT VERSIONS' LATENCY SUMMARY.

Type	Description	Latency[ms]
I	-	1800.2
II	constant mem.	263.4
	shared mem.	264.8
	μ look-up computing (shared mem.)	97.7
	μ look-up precomputing (const. mem.)	112.0
	optimized reduction method (Version S)	24.3
	uses Kawamura et. al. method (Version K)	28.4

We get rid of the computation over the extra basis m_e, significantly reducing the size of the divergent computation section.

V. EXPERIMENTAL EVALUATION

In this subsection we discuss the implementation and summarize the results for the different Algorithm types. Relative assessment is also presented by considering related art.

A. Implementation and Experimental Results

Table II presents a summary for the obtained latency results. The Algorithm Type II without look-up tables for the result μ suggests a latency of 263.4 ms for the complete point multiplication regardingless the data transfers. We also exploited the effect of getting the required constants from shared memory, copying them at a first moment, from the constant memory, since shared memory allows up to 16 simultaneous accesses while constant memory only allows 1. However, the results of this modification did not showed fruitful, since the latency was 0.5% higher (264.8 ms) as Table II suggests.

A table look-up for the results μ is possible for the Algorithm type II, since for $n = 15$ only 960 bytes are required. We evaluated the look-up solution with the look-up table stored in shared memory, computed at the beginning, and a pre-computed table loaded in constant memory. The obtained latency is 97.7 ms for the shared memory look-up approach, and 112.0 ms (15% higher) for the constant memory look-up as Table II suggests. These results suggest that the look-up is a good option, and also that the memory conflicts accessing constant memory begin to have a significant impact while the latency is decreasing.

Introducing the optimized reduction method and constants, the proposed Algorithm Type II provides a latency of 24.3 ms as shown in Table II. The Kawamura et. al method [7] (version K) did not result in the latency figure improvement, since one EC point multiplication takes 28.4 ms to perform, approximately 17% higher than the version supported in the Shenoy et. al. method [6] (version S) (see Table II). The fact that the insertion of the Kawamura et. al. method do not result in lower latency is not an expected

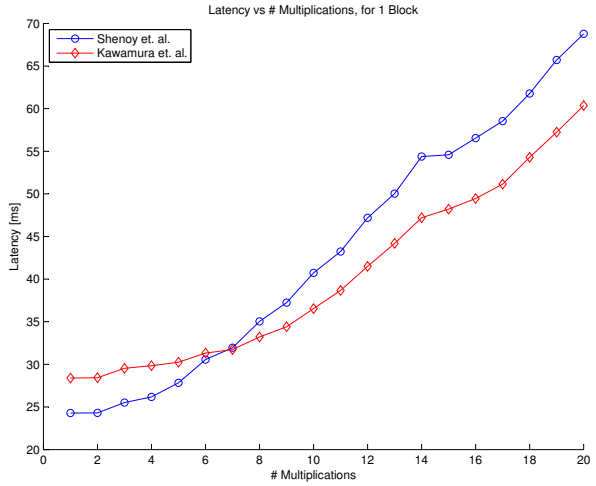


Figure 1. Latency for a different number of multiplications per block.

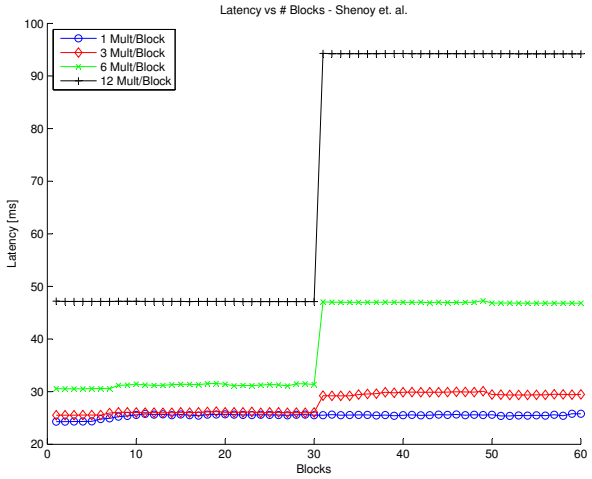


Figure 2. Version S latency vs. the number of blocks.

result, since the computation in the divergent part was substantially reduced without affecting the other parts. This result should be related with conflicts assessing the shared memory banks.

Another important metric for the EC point multiplication is the throughput. For an EC point multiplication we are using 15 threads corresponding to the 15 RNS channels. The CUDA framework allows for up to 512 threads per multiprocessor, thus we can perform more than one EC point multiplication per block, as long we have enough shared memory. The different multiplications performed within the block can share the same constants, including the look-up tables. Regarding the shared memory constraint, we are able to run up to 20 EC point multiplication within the same block, which corresponds to 300 threads. Figure 1 depicts the latency behavior while the number of multiplications per block is increased. We compare the Version S (Shenoy et. al. method) and Version K (Kawamura et. al. method) methods since they present very close latency values for only one EC point multiplication. As explained in Section IV-C, we would expect a better performance for version K, which did not occur for only one EC point multiplication. However, as Figure 1 suggests, the version K performance is better for more than 7 simultaneous EC point multiplications. This result suggest that, despite the version K is not able to provide lower latency than version S, it can provide higher throughput. The reason for the version K optimizations to pay off for a number of EC point multiplications bigger than 7 can be related with simultaneous computation of the divergent section of the code by the different EC point multiplications. Thus, for more multiplications, more advantage can be taken from the shortened divergent section.

We can expand our throughput also by taking advantage of the 30 existent multiprocessors in the employed GPU. In other words, we can use more than 1 block. Figure 2 shows the version S latency while expanding the number of blocks, for different number of EC point multiplications per block. From Figure 2 we observe the development of a gap at the 30 block reference while increasing the number of threads per block. The number of multiprocessors is 30, thus this gap is related with the ability of the compiler to assign different blocks to be computed simultaneously in the same multiprocessor. While the number of multiplications per block increases, the multiprocessors start being loaded with a larger amount of computation demands, hence the compiler starts splitting different blocks in sets of 30, computed in series by the 30 multiprocessors, and the gap increases.

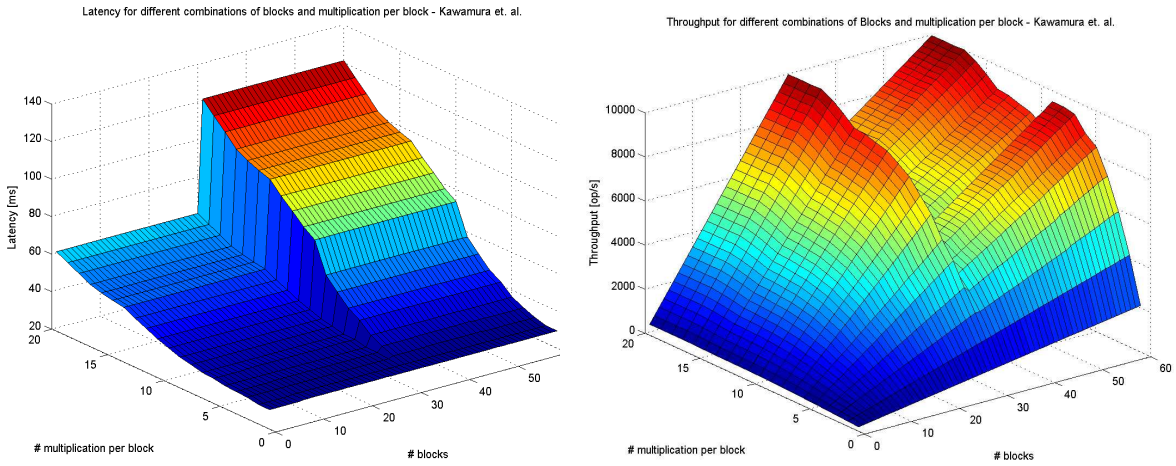
Figure 3 shows the version K latency and the throughput behavior for different combinations in the number of blocks and multiplications per block. From Figure 3 we can confirm the existence of the gap at the 30 blocks mark in version K. Another result of Figure 3 is that it is not worthwhile to use more than 30 blocks to achieve higher throughputs, specially for a large number of multiplications per block. The obtained results suggest a maximum throughput of 8730 op/s for the version S, and 9990 op/s for the version K. Version K can compute 600 EC point multiplications in 60.3 ms.

All these results do not consider data transfers since the data transfer latency is negligible in the overall latency measures. In our measurements the data transfer delay is at most 0.19% of the computation latency.

B. Related Art Comparison

The comparison with the related art, namely the experimental results, is not straightforward since different GPU platforms are employed, each one with different architectural and performance characteristics.

In [2] different approaches are proposed and compared to compute asymmetric cryptography, namely RSA and EC cryptography on a Nvidia 8800GTS GPU. For EC point multiplication, the authors only present results for a method based on schoolbook-type multiplication with reduction



(a) Latency (b) Throughput

Figure 3. Version K latency and throughput vs. different combinations of blocks and multiplications per block.

modulo a Mersenne number. Due to the lack of inherent parallelism in this method, an EC point multiplication is performed in only one thread, and the number of threads per block is limited to 36, due to shared memory restrictions. The authors implementation suggest a latency of 305 ms and a throughput of 1412.6 operations/s.

In [11] EC point multiplication is evaluated on a GPU for integer factorization. In this work, the authors use Montgomery representation for integers and set a multiprocessor as an 8-way array capable of simultaneously computing 8 field operations. Then they divide the EC point multiplication based on Non-Adjacent Form scalar recoding in 3 different instructions sets corresponding to double-double, double-add, and add-double operations using mixed (projective+affine) coordinate representation. Then, the authors map these sets in the 8-way array. Authors extrapolated a throughput figure that is about 2.14 times higher than the proposal in [2]. The authors do not present results for the latency of an EC point multiplication.

In [10] is evaluated a C++ library (PACE) to support modular arithmetic on an Nvidia 9800GX2 GPU. Using this library, the authors present results for an EC point multiplication. The Montgomery representation of integers is used to perform multi-precision arithmetic using the Finely Integrated Operand Scanning (FIOS) [8]. For 192-bit precision, results suggest a throughput of 1972 operations/s.

Table III summarizes the related art performance figures compared with the work herein proposed. We also present results for our best implementation running on a 8800GTS GPU, in order to perform a fair comparison with the other related art figures. Although, due to register restrictions, we were not able to compute the optimized implementation that run on the 285 GTX platform. For the 8800 GTS test we were only able to test an implementation that run up to 12 multiplications per block. For this platform, version K

Table III
RELATED ART COMPARISON FOR 224-BIT EC POINT MULTIPLICATION.

Ref	Platform	Lat.[ms]	T.put [op/s]	Observations
[2]	8800 GTS	305	1412.6	
[11]	8800 GTS	-	3019	ECM results
[10]	9800 GTX	-	1972	
Ours	8800 GTS	30.3	3138	tp. II, 12 mul./block
Ours	285 GTX	24.3	9990	tp. II, 20 mul./block

offers better performance both in latency and throughput. In order to compare to the 9800 GTX implementation, we have to bear in mind that this GPU has more computational resources than the 8800 GTS one.

With our implementation, we were able to beat in an order of magnitude the latency figures of the related art. We were not able to achieve similar gains in the throughput metric. However, we were able to provide 37% more throughput than [10] with our 8800 GTS implementation. We provide 3% more throughput than the extrapolation described in [11] and 54% more throughput than [2].

VI. CONCLUSIONS

In this paper we have proposed parallel algorithms for EC point multiplication on a GPU device by adopting a new RNS approach. This RNS approach achieves higher level of parallelism, thus higher performance in the massive parallel architecture of the GPU. We tested different implementation versions, that required different methods for the modular multiplication based on RNS base extension and different RNS precisions.

Experimental results suggest a maximum throughput of 9990 EC point multiplication per second and minimum latency of 24.3 ms, using an Nvidia 285 GTX GPU. We run our implementation in a lower end GPU for related art comparison, obtaining up to an order of magnitude

reduction in latency and up to 54% throughput improvement. The gains of the proposed implementation result from the higher utilization of the multiprocessor cores, by running up to 20 simultaneous EC point multiplications in each GPU multiprocessor.

REFERENCES

[1] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[2] R. Szerwinski and T. Guneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography," *Proc. Workshop on Cryptographic Hardware and Embedded Systems CHES*, pp. 79–99, Aug. 2008.

[3] J. Bajard, L. Didier, and P. Kornerup, "Modular Multiplication and Base Extension in Residue Number Systems," *Proc. 15th IEEE Symposium on Computer Arithmetic, ARITH'15*, pp. 59–65, 2001.

[4] J. Bajard, S. Duquenne, and N. Meloni, "Combining Montgomery Ladder for Elliptic Curves Defined Over Fp and RNS Representation," *Research Report LIRMM*, vol. 6041, 2006.

[5] N. I. of Standards and Technology, "Federal Information Processing Standards Publication 186-3: Digital Signature Standard," June 2009.

[6] A. Shenoy and R. Kumaresan, "Fast Base Extension using a Redundant Modulus in RNS," *IEEE Transactions on Computers*, vol. 38, no. 2, pp. 292–297, February 1989.

[7] S. Kawamura, M. Koike, F. Sano, and A. Shimbo, "Cox-Rower Architecture for Fast Parallel Montgomery Multiplication," *LNCS - Advances in Cryptology EUROCRYPT'2000*, pp. 523–538, January 2000.

[8] C. Kaya Koc, T. Acar, and J. Kaliski, B.S., "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, June 1996.

[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, March-April 2008.

[10] P. Giorgi, T. Izard, and A. Tisserand, "Comparison of Modular Arithmetic Algorithms on GPUs," *Proc. International Conference on Parallel Computing - ParCo'09*, October 2009.

[11] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, "ECM on Graphics Cards," *LNCS - Advances in Cryptology - EUROCRYPT'2009*, pp. 483–501, April 2009.