The Computer Journal Advance Access published November 30, 2011

doi:10.1093/comjnl/bxr119

RNS-Based Elliptic Curve Point Multiplication for Massive Parallel Architectures

SAMUEL ANTÃO[1,*], JEAN-CLAUDE BAJARD[2] AND LEONEL SOUSA[3]

[1] *Instituto Superior Técnico/INESC-ID, Technical University of Lisbon, Lisbon, Portugal*
[2] *Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie, Paris, France*
[3] *Instituto Superior Técnico/INESC-ID, Technical University of Lisbon, Lisbon, Portugal*
**Corresponding author: sfan@sips.inesc-id.pt*

Acceleration of cryptographic applications on massive parallel computing platforms, such as Graphic Processing Units (GPUs), becomes a real challenge concerning practical implementations. In this paper, we propose a parallel algorithm for Elliptic Curve (EC) point multiplication in order to compute EC cryptography on these platforms. The proposed approach relies on the usage of the Residue Number System (RNS) to extract parallelism on high-precision integer arithmetic. Results suggest a maximum throughput of 9827 EC multiplications per second and minimum latency of 29.2 ms for a 224-bit underlying field, in a commercial Nvidia 285 GTX GPU. Performances up to an order of magnitude better in latency and 122% in throughput are achieved regarding other approaches reported in the related art. An experimental analysis of the scalability, based on OpenCL descriptions of the proposed algorithms, suggest that further advantage can be obtained from the proposed RNS approach for GPUs and EC curves supported by underlying finite fields of smaller size, regarding implementations on general purpose multi-cores.

Keywords: residue number system; elliptic curve; cryptography; GPU

Received 24 May 2011; revised 19 October 2011
Handling editor: Albert Levi

1. INTRODUCTION

The recent developments on computing architectures suggest that computing devices tend to contain an increasing number of simpler cores as a way to boost the computational power of such parallel devices. However, these developments demand an extra effort to design efficient and scalable algorithms to exploit the parallel capabilities of the computing platforms. Introducing a controlled additional penalty may result in enhanced parallelization solutions that take advantage of the existing massive multi-core platform's capabilities.

Examples of massive parallel computing devices are Graphical Processing Units (GPUs). GPUs have been increasingly used as a powerful accelerator in several high computational demanding applications [1]. The huge computational power of a GPU allied with its low cost, mainly because of mass production for the gaming market, turn the GPUs interesting for General Purpose Processing (GPGPU) [1]. Applications that take advantage of the GPU computing power can be found in different fields, such as physics [2, 3], biology [4], cryptography [5, 6], and of course image and video processing and coding.

The general purpose Central Processing Units (CPUs), which are present in every single desktop or laptop, nowadays tend to contain increasingly more cores, although less numerous and individually more complex than the GPU's cores. Nevertheless, the evolution of GPUs and CPUs tends to approach both solutions to the same path, hence the constraints and performance guidelines for both devices tend to be common.

The different applications may have different properties that turn them more or less suitable to be implemented on massive parallel platforms, namely: (i) low data dependencies, allowing to compute in parallel instances over independent data sets; (ii) regular description, allowing for the identification of different single computation flows to enhance the parallelization and scalability for data sets with different sizes; (iii) low number of memory accesses, taking advantage of the huge processing power reducing stalling effects waiting for data from memory; and (iv) 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 on massive parallel platforms have similar challenges that demand designing/redesigning algorithms to use extensive parallel processing on independent data sets.

In this paper, we get through the efficient implementation of asymmetric cryptography supported on Elliptic Curves (ECs) on massive parallel computation devices. 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 highly parallel computing devices. In particular, the EC point multiplication with a scalar (private key), which is the most computational demanding operation in EC cryptography, represents a demanding computation which input is a small data set represented by three integers of up to 521 bits, consisting of a private and a public key [7]. Yet, the EC point multiplication is constructed on several successive steps where the scalar is browsed, and there are data dependencies between the successive steps.

In order to overcome the inefficiency due to the data dependencies and compute in parallel the EC point multiplication, we propose in this paper to use the Residue Number System (RNS) approach [8]. RNS is an alternative representation of integer numbers in several smaller residues for an established basis. With the operands sharing a common RNS representation, the computation can be performed in parallel on each residue (channel). 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 high performance implementations of EC cryptography supported on massive parallel computation devices. We propose parallel algorithms and evaluate their efficiency on both GPUs and multi-core CPUs, in order to conclude about the best practices and generalize the results for other parallel architectures.

This paper is organized as follows. In Section 2, we provide background on the operations required by EC cryptography supported on the underlying field $GF(p)$, and give an overview of the RNS arithmetic. Section 3 describes in detail the Montgomery method to efficiently perform multi-precision modular multiplication with RNS representation. Section 4 proposes the RNS-based algorithms and presents the main aspects for implementing them in a GPU architecture with CUDA. Section 5 discusses the obtained experimental results and accesses the obtained performance figures regarding the related art. In Section 5.3, based on an OpenCL implementation, the results are generalized for different architectures and ECs supported on differently sized underlying fields. Finally, Section 6 draws some conclusions from the developed work and obtained results.

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: $P = G$, $Q = 2G$;
2: for $l = k - 2$ down to 0 do
3:     if $s_l = 1$ then
4:         $P = P + Q$, $Q = 2Q$;
5:     else
6:         $Q = P + Q$, $P = 2P$;
7:     end if
8: end for
```

2. EC AND RNS BACKGROUND

This section provides the background on EC over $GF(p)$ and RNS arithmetic required for the rest of this paper.

2.1. EC cryptography over $GF(p)$

An EC $E(a, b, p)$ over the finite field $GF(p)$, 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 verify the following equation:

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

In order to obey smoothness conditions, the parameters a and b verify $-(4a^3 + 27b^2) \neq 0 \bmod p$. 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 $Q = P + P + \cdots + P = sP$. It is known to be computationally hard to compute s knowing Q and P (Elliptic Curve Discrete Logarithm Problem) [9].

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 [10]. Algorithm 1 presents the Montgomery Ladder method 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 inversion, 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. In order to transpose between the standard projective (X) and the affine (x) representations of a coordinate, the equivalence $x \Leftrightarrow X/Z$ holds. The projective versions of point addition and doubling to support Algorithm 1 are the following (for $s_l = 0$) [10]:

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

For $s_l = 1$, the formula used to double P is used to double Q instead. Note that the expressions in (2) do not require the y coordinate of an EC point. At the end of Algorithm 1, the affine x and y coordinates can be obtained from the projective ones (X_P,Z_P) and (X_Q,Z_Q), which involves a modular inversion [10]:

$$\begin{aligned} x_P &= X_P/Z_P, \; x_Q = X_Q/Z_Q, \\ y_P &= \frac{-2b + (a + x_Gx_P)(x_G + x_P) - x_Q(x_G - x_P)^2}{2y_G}. \end{aligned} \tag{3}$$

As it can be concluded from Algorithm 1, each iteration of the loop has data dependencies from the previous iteration, which is an extra difficulty toward the design of a parallel algorithm. Moreover, the sizes of the prime p for standardized EC are 192, 224, 256, 384 and 521 bits [7]. Thus, the operations over the coordinates would require a datapath of the same size. In order to adapt the coordinates' size to the processing cores' datapath (usually 32-bit or 64-bit width) and expose more parallelism, we propose to use an RNS representation of the field elements. This representation allows parallelizing the field operations among the available processing cores, providing the required degree of parallelization toward an efficient implementation.

2.2. RNS overview

The RNS is based on the Chinese Remainder Theorem (CRT), which states that, for a given basis B_n consisting of n relatively prime integers $(m_1, m_2, \ldots, m_n)$, there is a unique representation for the integer $X < M$ in the form

$$x_j = X \bmod m_j, \quad 1 < j < n, \quad M = \prod_{i=1}^{n} m_i.$$

For two integers X and Y represented in RNS with the same basis, the operation $Z = X \otimes Y \bmod M$ can be performed directly on the channels by computing $z_j = x_j \otimes y_j \bmod m_j$, where $\otimes$ represents addition, subtraction or multiplication. As suggested above, the advantage of using the RNS representation is the possibility of splitting the computation in n parallel flows (henceforth called channels), each one operating modulo a different m_i. The conversion from binary to the RNS representation for an integer X can be accomplished by computing the residues x_j directly and in parallel. For the opposite conversion, there are two methods that can be used: the Mixed Radix System (MRS), which uses sequential processing, and the CRT, which allows parallelizing the computation [11].

The MRS method employs an intermediate mixed radix representation during the conversion. The binary representation X can be obtained from the mixed radix digits x'_i as follows:

$$X = x'_1 + x'_2m_1 + x'_3m_1m_2 + \cdots + x'_nm_1 \cdots m_{n-1}. \tag{4}$$

The mixed radix digits are obtained from the RNS representation of X using a recursive method [12], which is not suitable for parallel implementations [6]. Since this method does not suggest being worthwhile for GPU architectures, we do not consider this method in the present work.

The alternative method relies on the CRT definition for computing the binary representation:

$$X = \sum_{i=1}^{n} \xi_i M_i - \alpha M, \; \alpha < n, \; \xi_i = \left| \frac{x_i}{M_i} \right|_{m_i}, \tag{5}$$

where $M_i = M/m_i$, $|\cdot|_{m_i}$ denotes an operation modulo m_i, and $|1/M_i|_{m_i}$ is the multiplicative inverse of M_i modulo m_i. In (5), the subtraction of αM allows obtaining the required reduction modulo M. Two main methods have been used to compute the constant α. An extra modulus m_e can be defined and all the operations performed not only on the basis B_n, but also on this extra modulus (Shenoy and Kumaresan [13]). 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 (5), it is obtained that

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

Rewriting (6), α can be obtained as follows:

$$|\alpha|_{m_e} = \left| \left| \sum_{i=1}^{n} \xi_i M_i - x_e \right|_{m_e} |M^{-1}|_{m_e} \right|_{m_e}. \tag{7}$$

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

Another possible method to compute α involves a successive fixed point approximation approach (Kawamura *et al.* [14]). Knowing that $X < M$ or $X/M < 1$, this method observes that (5) can be rewritten as

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

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

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

where r is such that $2^{r-1} < m_i \leq 2^r$, and $\mathrm{trunc}_q(\xi_i)$ sets the $r-q$ 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 choosing good values for β supported on the maximum initial approximation errors, respectively:

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

In [14] two Theorems are stated concerning the computation of the value $\hat{\alpha}$ and its relationship with β and the errors stated in (10). One of these Theorems refers to the computation of a non-exact, although bounded, value of $\hat{\alpha}$ that allows a non-exact RNS to binary conversion. The other Theorem refers to an exact conversion, suggesting that 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 $\alpha = \hat{\alpha}$.

3. RNS MONTGOMERY MULTIPLICATION

In order to perform EC arithmetic with RNS, we should not only provide a method to add and to 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 [15]. This method evaluates $\bar{Z} = (XY)R = \bar{X}\bar{Y}R^{-1} \bmod N$, operating in a so-called Montgomery domain for which a field element X is replaced by $\bar{X} = XR \bmod N$.

An RNS version of the Montgomery modular multiplication algorithm can be designed [8]. The RNS version follows the same concepts of the binary version. However, with the RNS representation it is no longer easy to reduce modulo a power of 2. Instead, defining an RNS basis B_n with dynamic range M, it is easy to reduce an element represented with the basis B_n modulo the dynamic range M. It only requires reducing modulo m_i in each one of the RNS channels. Hence, in the RNS Montgomery Multiplication version $R = M$, thus the following expression is evaluated instead:

$$\bar{Z} = \bar{X}\bar{Y}M^{-1} \bmod N. \quad (11)$$

In order to make the RNS version to behave correctly, the conditions $M > N$ and $\gcd(M, N) = 1$ must be verified. One of the drawbacks of the RNS version is its impossibility to represent M^{-1} in B_n. Therefore, it is required to set another basis $\tilde{B}_n$ with dynamic range $\tilde{M}$ such that $\tilde{M} > M$, $\gcd(\tilde{M}, M) = 1$ and $\gcd(\tilde{M}, N) = 1$ to evaluate (11).

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 $U \equiv \bar{X}\bar{Y}M^{-1} \bmod N$. As the inverse of M cannot be defined in B_n, an extra RNS basis $\tilde{B}_n$ is set, which will be used to compute U. The division by M is accomplished by multiplying by its inverse, which maintains the multiplication result bounded and avoids the reduction modulo N. Hence, we must ensure that $M|(T + QN) \Leftrightarrow T + QN \equiv 0 \bmod M$. Therefore, we must set the value of Q such that the aforementioned conditions are verified. To ensure this, we need to compute Q in each RNS channel i for the basis B_n:

$$0 = t_i + q_i n_i \bmod m_i \quad \Leftrightarrow \quad q_i = -t_i |n_i^{-1}|_{m_i} \bmod m_i. \quad (12)$$

After computing the value of Q, we can use one of the conversion methods based on the CRT referred to in Section 2.2 to convert the value of Q to the basis $\tilde{B}_n$. In this basis, for each RNS channel i we compute:

$$\tilde{u}_i = (\tilde{t}_i + \tilde{q}_i \tilde{n}_i)|M^{-1}|_{\tilde{m}_i} \bmod \tilde{m}_i. \quad (13)$$

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 $N < M$). While computing algorithms based on the Montgomery multiplication, we can allow the intermediary results to be not exactly reduced but bounded. Thus, setting M such that $M > 4N$, considering $Z = U$ and applying Z as input in further multiplications, the multiplication result U', despite not exactly reduced, will be correct modulo N and bounded, since $U' < [(2N)^2 + MN]/M < 2MN/M = 2N$ shares the same upper bound with U. Figure 1 depicts an overview of the aforementioned described base extension method.

The conversion between bases, base extension, can be achieved by applying any of the RNS to binary conversion algorithms described in Section 2.2 modulo each of the new basis moduli. In other words, n parallel instances of an RNS to binary conversion algorithm are computed, each one over a different RNS channel of the new basis. In the RNS version of the Montgomery multiplication algorithm, the most costly steps are precisely the conversion between bases, 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. As (5) suggests, the conversion from an RNS basis implies the computation of a constant α that multiplied by the dynamic range M corrects an offset in the conversion to maintain the result bounded by M. In [8] it is suggested to use the CRT conversion without the correction term introduced by α during the first conversion

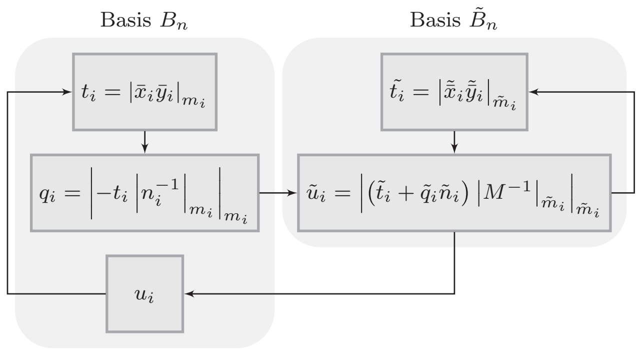

Basis B_n
Basis $\tilde{B}_n$
$t_i = |\bar{x}_i \bar{y}_i|_{m_i}$
$q_i = \left| -t_i \left| n_i^{-1} \right|_{m_i} \right|_{m_i}$
$\tilde{t}_i = |\tilde{\bar{x}}_i \tilde{\bar{y}}_i|_{\tilde{m}_i}$
$\tilde{u}_i = \left| (\tilde{t}_i + \tilde{q}_i \tilde{n}_i) \left| M^{-1} \right|_{\tilde{m}_i} \right|_{\tilde{m}_i}$
u_i

FIGURE 1. Base extension method overview.

(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

$$\hat{U} = (T + \hat{Q}N)M^{-1} = (T + QN)M^{-1} + \alpha N. \qquad (14)$$

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 $QN < NM$ ensures that the multiplication result is

$$\begin{aligned}\hat{U}' &< [((n+2)N)^2 + MN]/M + nN \\ &< 2MN/M + nN = (n+2)N. \qquad (15)\end{aligned}$$

In summary, with this method the computation of α during one conversion is avoided while keeping the multiplication result bounded by an acceptable value (note that $n << N$).

4. THE EC PARALLEL ALGORITHMS

In this section, we propose parallel algorithms for EC point multiplication supported on Algorithm 1, based on the projective representation of the EC coordinates, and by adopting RNS arithmetic, as introduced in Section 2. We target and evaluate the proposed algorithms for a particular Nvidia GPU (Nvidia GTX 285 GPU) platform and EC using the CUDA framework [16], in order to identify the performance guidelines prior to the generalization for any type of curve and parallel computing device. For this particularization a general EC standardized by NIST is considered for a prime number p with 224 bits, where $a = -3$ [7]. Since it is a standard, using this curve may enhance the inter-operability of the algorithms with other application following the standard. Nevertheless, this standardized curve was not chosen because of any performance issue, since the adopted algorithms are able to offer the same performance for other curve parameters, namely, other underlying finite fields. Hence, the proposed algorithms suit the demands of industrial or military users interested in more generic algorithms, generic curves or generic underlying finite fields to enhance security or due to patent issues [17]. 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 $Q = 2G$, and each iteration of the main loop computes the operations in (2). The schedule in Table 1 can be adopted to perform each of the EC point multiplication's loop iterations; this schedule is only for $s_l = 0$. The loop schedule for $s_l = 1$ can be obtained by commutating P and Q in the loop section of Table 1. In Table 1 is also presented a schedule for the initialization section of the algorithm. Note that the EC point multiplication algorithm's schedule (Montgomery Ladder) only has two inputs: the EC parameter b and the x coordinate of the input point G (x_G). The scheduling is organized around sets of multiplications and additions. Each multiplication set is composed of several independent field multiplications, and each addition set is composed of field additions/subtractions and multiplications by small constants.

TABLE 1. Operations scheduling for the initialization and for each iteration of the loop in the EC point multiplication algorithm (for $s_l = 0$).

Init.	mult. 1	$A = x_G^2$
		$B = bx_G$
	add.	$C = A + 3$
	mult. 2	$C = C^2$
		$A = x_G A$
	add. 2	$X_Q = C - 8B$
		$Z_Q = 4(A - 3x_G + b)$
Loop	mult. 1	$A = X_P Z_Q$
		$B = X_Q Z_P$
		$C = X_P X_Q$
		$D = Z_P Z_Q$
		$E = X_P^2$
		$F = Z_P^2$
		$H = bZ_P$
	add. 1	$Z_Q = A - B$
		$X_Q = A + B$
		$C = C + 3D$
		$A = E + 3F$
	mult. 2	$D = DX_Q$
		$X_Q = C^2$
		$Z_Q = Z_Q^2$
		$A = A^2$
		$B = FH$
		$E = EX_P$
		$F = X_P F$
	add. 2	$G = E + B - 3F$
	mult. 3	$D = bD$
		$Z_Q = xZ_Q$
		$B = X_P B$
		$F = Z_P G$
	add. 3	$X_Q = X_Q - 4D$
		$X_P = A - 8B$
		$Z_P = 4F$

The generic flow of the proposed parallel algorithms is presented in Fig. 2. In step 1, n threads perform the required EC arithmetic in the n channels determined by each of the basis B_n and $\tilde{B}_n$. The operations performed prior the reduction will constrain the minimum number of RNS channels (number of threads), since the operations will determine the required dynamic range which must bound the precision of the operations. Higher number of RNS channels will allow for higher parallelization but will also increase the reduction complexity, thus the best trade-off must be achieved. In this

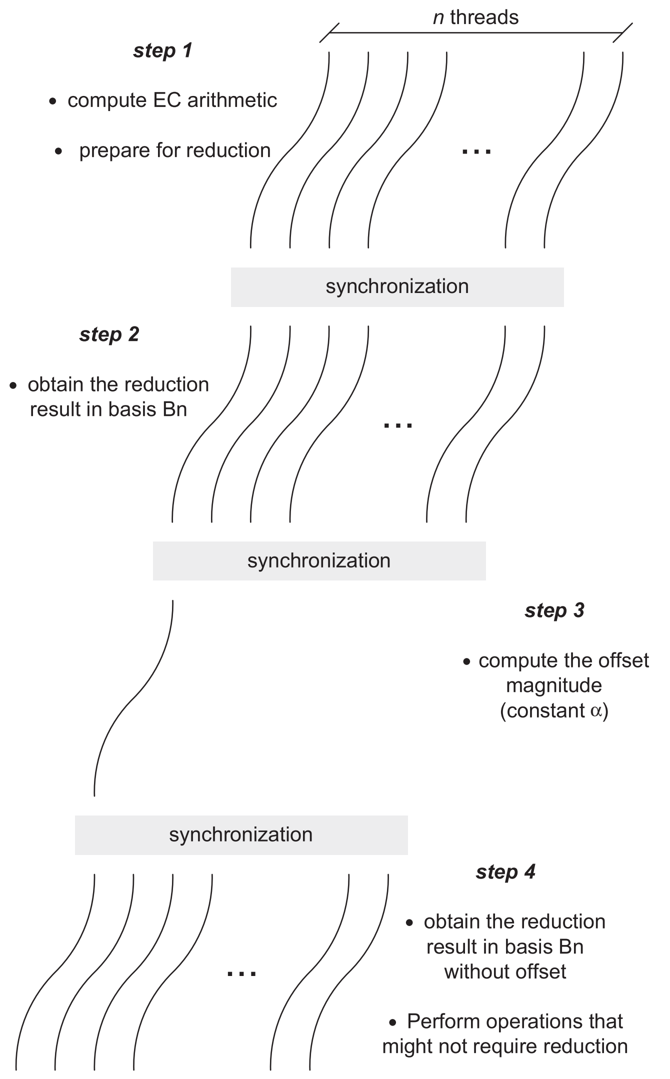

n threads
step 1
• compute EC arithmetic
• prepare for reduction
synchronization
step 2
• obtain the reduction result in basis Bn
synchronization
step 3
• compute the offset magnitude (constant α)
synchronization
step 4
• obtain the reduction result in basis Bn without offset
• Perform operations that might not require reduction

FIGURE 2. RNS-based algorithm flow for EC point multiplication.

paper, different operations are considered prior to the reduction, namely the operations that compute a complete iteration of the EC point multiplication algorithm's loop (see type I algorithm in Section 4.2), and the operations that compound a single multiplication and addition set of each loop's iteration (see type II algorithm in Section 4.3). Also, in this step the reduction is prepared by computing the value Q (see (12)) in basis B_n, which is required for the following reduction steps. In the step 2, the conversion of Q to $\tilde{Q}$ is accomplished. In this conversion an offset is allowed as described in Section 3, since it allows reducing the conversion complexity without a significant penalty in increased precision. Hence, in step 2, the result $\hat{U}$ in (14) is also computed. Since during conversion the results of all the RNS channels of the basis B_n are required, a synchronization barrier is adopted before step 2. Step 3 is responsible for computing the value α in (5), which is required to correct the offset that occurs in the final conversion from $\tilde{B}_n$ to B_n that will take place in step 4. Computing α is a single task for a single thread that gathers information on $\hat{U}$ from all RNS channels as the methods to compute this constant in Section 2.2 suggest; thus a synchronization barrier is required prior to this step. Step 4 converts $\hat{U}$ in basis $\tilde{B}_n$ to basis B_n, removing the offset quantified by α obtaining the final reduced result U.

In the following sections, we give insight into the GPU parallel architecture that will be used for evaluating the proposed algorithms, and describe these algorithms with details. Two main algorithms are tested: a Type I algorithm, which requires higher dynamic range (more RNS channels), and a Type II algorithm, which demands lower dynamic range. As it will be shown, the type II algorithm is much more efficient to perform the EC point multiplication on GPUs than the type I algorithm. Nevertheless, for the sake of completeness, we present both types of algorithms and the respective experimental results, discussing the advantages of the type II compared with the type I algorithm.

4.1. General purpose processing on GPUs

Tesla is a typical architecture of a GPU that consists of several general purpose scalar processors grouped in multiprocessor cores that allows for general purpose processing [18]. Figure 3 depicts an overview of the Tesla architecture, adopted in the modern NVIDIA GPUs. Furthermore, the NVIDIA Compute Unified Device Architecture (CUDA) allows programmers to easily develop applications for these devices.

In order to exploit data parallelism on GPUs, CUDA provides different units of parallelism. The smallest unit is the thread, and each multiprocessor 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 is called a warp, and the way the threads in a warp are executed obeys a SIMD flow. Whenever there is divergence in the executed threads of a warp, the execution is serialized as only one thread is executed at a time. The threads are organized in a higher level parallelism abstraction unit called a block. Different blocks are independent and can run in parallel by using the several multiprocessors. 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 mainly depends on the multiprocessor local resources (shared memory, cache, registers availability) and on the data dependencies.

In each multiprocessor there is a 16 kB shared memory, and a 8 kB symbols cache that can be used for read only data (constants). Despite possible conflicts between different threads, the memory inside a multiprocessor is accessed in the same amount of time as a register. There is also a global memory, where the initialization data is written by the GPU host. The global memory has a higher accessing latency (40–60 times higher than the shared memory latency [19]), thus its utilization and accessing patterns should be judiciously set to avoid long stall periods by a multiprocessor.

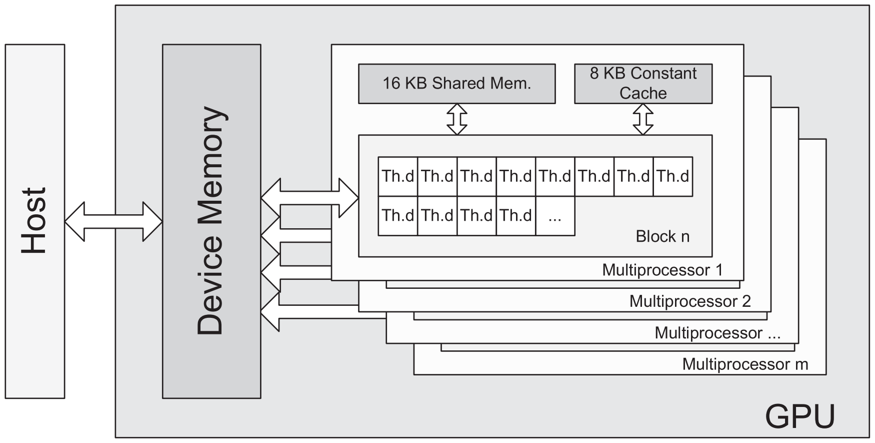


FIGURE 3. Tesla architecture overview employed in the Nvidia GPUs.

Each scalar processor core 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 as 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.

An alternative to the CUDA framework for programming GPUs is the Open Computing Language (OpenCL) framework [20]. OpenCL is a programming framework that allows users to program once and execute across heterogeneous platforms consisting not only of GPUs but also CPUs and other accelerators [21]. OpenCL supports a language based on C++ for coding kernels (functions that execute on OpenCL devices), and provides an Application Programming Interface (API) recently standardized. Two main API layers are provided: (i) a platform layer API for hardware abstraction that allows querying, selecting and initializing computing devices, and also creating computing contexts and queues; (ii) a runtime API layer for managing resources and executing computing kernels, as well as for scheduling the execution on the available resources. It provides on-line or off-line compilation and generates computing kernel executables. With OpenCL data is communicated between the host and computing devices as a buffer object, which is an unstructured buffer such as C++ arrays of scalars, vectors or other user-defined structures (these data structures are accessed in the kernel programs via pointers). The programming of the kernels is made in the C++ programming language, with well-defined numerical accuracy, and IEEE 754 rounding with specified maximum error.

Several GPU and CPU providers developed support for OpenCL implementations. E.g.: (i) Nvidia supports OpenCL on the CUDA compliant GPU devices [22]; (ii) AMD developed the ATIstream technology that allows running OpenCL implementations on the ATI GPUs, as well as on the AMD multi-core processors [23]; and (iii) Apple supports OpenCL on the Mac platforms (both CPU and GPU) that natively ships with the Mac OS X operating system. In comparison with frameworks that target single devices (e.g. the CUDA framework for NVIDIA GPUs), the flexible implementations based on OpenCL may present some overhead due to the management implied in handling such flexibility.

4.2. Type I algorithm

Let us assume that the GPU inputs and outputs are in the RNS format. These assumptions are based on the fact that the computational demanding core of the algorithm is in the loop for computing the resulting X and Z projective coordinates.

Regarding the operation schedule in Table 1, for each EC point multiplication 11 variables are required to store intermediate data. To perform modular operations among the RNS channels, a multiplication has to be performed with full precision, 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 compute 16-bit arithmetic modulo a basis element of the form $2^k - c$, with $k = 16$ and $c \geq 0$. The required number of RNS channels depends on the range to represent the field operations result. 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 step for each RNS channel. Thus, with this intensive memory usage, global memory should only be used to store the input data and the final results. Hence, the number of multiplications handled by each multiprocessor is constrained by the size of the shared memory.

The type I algorithm is based on the RNS Montgomery Multiplication introduced in Section 3. The projective coordinates of the input point are stored in global memory, and the scalar is stored in constant memory. The type I algorithm obtains the resulting projective coordinates X_P, Z_P, X_Q and Z_Q of the complete loop in Table 1 prior to the reduction. Note that for the type I algorithm the inputs are not in the Montgomery domain. Hence, the unreduced results are then multiplied by $M \bmod N$, using the method introduced in Section 3, in order to obtain the partial reduced values. Multiplying by $M \bmod N$ allows canceling the factor $M^{-1} \bmod N$ introduced by the Montgomery multiplication method, thus the output remains in the same domain as the inputs. Thereafter, for the type I algorithm, the schedule in Table 1 is modified to the schedule in Table 2. Considering the results in [6] that compare the Shenoy *et al.* and Kawamura *et al.* methods for computing α (see Section 2.2) on a GPU, in order to obtain the final results in the basis B_n from the results in the basis $\tilde{B}_n$, we follow the former approach (based on (7)) because it allows achieving better performance.

Let us find the required dynamic range to compute the Table 2 loop without having to reduce intermediary results mod M. As explained in Section 3, 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 $GF(p)$. We also know that the EC parameter $b < N$. Since Table 2 loop operations are equivalent to (2), the maximum value of a projective coordinate prior to the multiplication by $M \bmod N$ is

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

Considering the inequalities above, the maximum value we are willing to reduce is upper bounded by $(8N+16)u^4(M \bmod N)$. Therefore, recalling that during the reduction we are interested in computing (14), we must ensure that $M \geq (8N+16)u^4$, since this condition allows bounding the result $\hat{U} < ((8N+16)u^4(M \bmod N) + MN)/M + nN < (MN + MN)/M + nN = (n+2)N = u$. We define a moduli set composed of $2n$ elements of the form $2^{16} - c_i$, with c_i an odd number, and $c_1 = 1$; and compose the bases B_n and $\tilde{B}_n$ of elements of this moduli set assuring $M < \tilde{M}$. The required number of elements in order to satisfy $M \geq (8N+16)u^4$ in each base was found to be $n = 72$. The extra element used to compute (7) is 2^k such that $2^k > n$; note than any number 2^k is relative prime to B_n and $\tilde{B}_n$.

The multiplication by $M \bmod N$ and the following reduction is accomplished simultaneously for all the unreduced results X_P, Z_P, X_Q and Z_Q. Table 3 details the operations performed in each step of the algorithm flow in Fig. 2 by each of n threads. 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, allowing for the threads to cooperate. We refer to any of the unreduced results X_P, Z_P, X_Q and Z_Q as V, and $W = M \bmod N$. Each element of B_n (m_i) and $\tilde{B}_n$ ($\tilde{m}_i$) is assigned to the thread i. There is one thread associated with an element $m_e = 2^k$, being the thread responsible for computing (7) and the required operations mod m_e. Each thread performs arithmetic mod an element of the bases B_n and $\tilde{B}_n$.

TABLE 2. Operations scheduling for the initialization and for each iteration of the loop in the implemented EC point multiplication type I algorithm (for $s_l = 0$).

Init.	mult. 1	$A = x_G^2$
		$B = bx_G$
	add.	$C = A + 3$
	mult. 2	$C = C^2$
		$A = x_G A$
	add. 2	$X_Q = C - 8B$
		$Z_Q = 4(A - 3x_G + b)$
		$X_Q = X_Q(M \bmod N)$
		$Z_Q = Z_Q(M \bmod N)$
	reduce X_Q, Z_Q	
Loop	mult. 1	$A = X_P Z_Q$
		$B = X_Q Z_P$
		$C = X_P X_Q$
		$D = Z_P Z_Q$
		$E = X_P^2$
		$F = Z_P^2$
		$H = bZ_P$
	add. 1	$Z_Q = A - B$
		$X_Q = A + B$
		$C = C + 3D$
		$A = E + 3F$
	mult. 2	$D = DX_Q$
		$X_Q = C^2$
		$Z_Q = Z_Q^2$
		$A = A^2$
		$B = FH$
		$E = EX_P$
		$F = X_P F$
	add. 2	$G = E + B - 3F$
	mult. 3	$D = bD$
		$Z_Q = xZ_Q$
		$B = X_P B$
		$F = Z_P G$
	add. 3	$X_Q = X_Q - 4D$
		$X_P = A - 8B$
		$Z_P = 4F$
		$X_P = X_P(M \bmod N)$
		$Z_P = Z_P(M \bmod N)$
		$X_Q = X_Q(M \bmod N)$
		$Z_Q = Z_Q(M \bmod N)$
	reduce X_P, Z_P, X_Q, Z_Q	

TABLE 3. Type I algorithm's steps for the thread i.

Step in Figure 2	Computation for thread i
Step 1	Initialization or loop operations in Table 2 (mod m_i) and (mod $\tilde{m}_i$) $v_i = -w_i v_i (n_i)^{-1}_{m_i} \lvert M_i \rvert^{-1}_{m_i} \bmod m_i$
Step 2	$\tilde{v}_i = \left(\left\lvert \sum_{j=1}^{n} v_j M_j \right\rvert_{\tilde{m}_i} \tilde{n}_i + \tilde{v}_i \tilde{w}_i \right) \lvert M \rvert^{-1}_{\tilde{m}_i}$ $\tilde{a}_i = \tilde{v}_i \lvert \tilde{M}_i \rvert^{-1}_{\tilde{m}_i}$
Step 3	Init. or loop operations in Table 2 (mod m_e) $v_e = \left(\left\lvert \sum_{j=1}^{n} v_j M_j \right\rvert_{m_e} n_e + v_e w_e \right) \lvert M \rvert^{-1}_{m_e}$ $\alpha_W = \left(\left\lvert \sum_{j=1}^{n} \tilde{a}_j \tilde{M}_j \right\rvert_{m_e} - v_e \right) \lvert \tilde{M} \rvert^{-1}_{m_e}$
Step 4	$v_i = \left\lvert \sum_{j=1}^{n} \tilde{a}_j \tilde{M}_j - \alpha_W \tilde{M} \right\rvert_{m_i}$

The steps presented in Table 3 uses the following precomputed constants:

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

Since the steps presented in Table 3 are computed to obtain four reduced results simultaneously, the result $\mu = \{M_i \bmod \tilde{m}_j, \tilde{M}_i \bmod m_j, \ M_i \bmod m_e, \ \tilde{M}_i \bmod m_e\}$ is computed only once and shared by the four different running reductions in each iteration. The latency value for the type I algorithm would not be practical in real applications (latency > 1 s), as observed from Table 6. The main drawbacks of the type I algorithm are related with the number of synchronizations, the number and size of the divergent code sections (computed in series by a single thread), and with the complexity of computing the result μ every iteration needed. Synchronization barriers and divergence cannot be removed once (7) has to be computed before the next dependent steps. The computation of μ exhibits quadratic complexity, and thus trading the required dynamic range (RNS channels) by the number of calls of the base extension algorithm may improve the performance. Replacing this computation by table look-ups will not be efficient since we would require $2n(n+1)$ entries: for $n = 72$ and 16-bit entries, this corresponds to 21,024 bytes, which exceeds the shared memory capacity.

4.3. Type II algorithm

Once the potential inefficiencies of the type I algorithm are identified, a type II algorithm is proposed. These inefficiencies are mainly due to the large M/N ratio in the type I algorithm, which is caused by a higher dynamic range M, resulting in reduced performance in the basis extension method. The dynamic range considered in type II algorithm only supports the computation of a set of multiplications and a set of additions in Table 1 prior to a reduction or, in other words, prior to a base extension. Considering the modifications introduced by the type II algorithm, the schedule in Table 1 is updated to the schedule in Table 4.

Let us find the required dynamic range to compute any set of multiplications followed by a set of additions in Table 4. As explained in Section 3, the maximum value of the 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 $GF(p)$. We also know that the EC parameter $b < N$, and $x_G < N$. Performing an analysis on Table 4, similar to the one presented in (16), the required dynamic range has its lower bounds in Z_Q (initialization step, add. 2), bounded by $4(u + 4N)$, or in X_P (loop step, add. 4) bounded by $9u$. By using $u = (n+2)N$, we get that $9u > 4(u + 4N)$ for $n > 1$. Since a minimum $n = 14$ is required to represent N in 16-bit channels, the precision is bounded by $9u$. Performing an analysis as the one presented in (15), considering that the multiplication inputs are bounded by $9u$, setting $M > (9u)^2/N = (9(n+2))^2 N$ will bound the multiplication output (addition input) to

$$\hat{U} < [(9u)^2 + MN]/M + nN < \\ < [MN + MN]/M + nN = (n+2)N = u, \qquad (17)$$

which is the bound from where we depart. For a 224-bit prime N, the condition $M > (9(n+2))^2 N$ results in $n = 15$, considering that the moduli that compound the RNS bases are obtained as in the type I algorithm, i.e. are of the form $2^k - c$ [24].

The type II algorithm also follows the flow in Fig. 2 and its steps. The steps detailed in Table 5 already contain the optimizations to the type II algorithm, where X and Y are any input of a multiplication set in the initialization or loop in Table 4, and Z is an output. Similarly to type I algorithm, each thread has one correspondent element of B_n (m_i) and $\tilde{B}_n$ $(\tilde{m}_i)$. There is a thread that is associated with an element $m_e = 2^k$, being the thread responsible for computing α in (7) and all the required operations mod m_e, assuming a divergent behavior for this purpose.

In the RNS Montgomery Multiplication (Section 3), there are several constants employed. These constants can be more efficiently applied if merged into only one [25]. These changes allow saving memory and computation resources. The following summarizes the merging of constants and the required changes in the RNS Montgomery Multiplication:

(i) New constant $r_i = -\lvert n_i M_i \rvert^{-1}_{m_i}$;
(ii) New constant $s_i = \left\lvert \tilde{n}_i \lvert \tilde{M}_i \rvert^{-2}_{\tilde{m}_i} \right\rvert_{\tilde{m}_i}$;
(iii) New constant $t_i = \left\lvert \lvert M \rvert^{-1}_{\tilde{m}_i} \lvert \tilde{M}_i \rvert_{\tilde{m}_i} \right\rvert_{\tilde{m}_i}$;
(iv) Remove constants $\lvert n_i \rvert^{-1}_{m_i}$, $\lvert M_i \rvert^{-1}_{m_i}$, $\lvert \tilde{M}_i \rvert^{-1}_{\tilde{m}_i}$, and $\lvert M \rvert^{-1}_{\tilde{m}_i}$;

TABLE 4. Operations scheduling for the initialization and for each iteration of the loop in the implemented EC point multiplication type II algorithm (for $s_l = 0$).

Inputs x_G and b in Montgomery domain.		
Init.	mult. 1	$A = x_G^2$ $B = bx_G$
	reduce $A,\ B$	
	add.	$C = A + 3$
	mult. 2	$C = C^2$ $A = x_G A$
	reduce $C,\ A$	
	add. 2	$X_Q = C - 8B$ $Z_Q = 4(A - 3x_G + b)$
Loop	mult. 1	$A = X_P Z_Q$ $B = X_Q Z_P$ $C = X_P X_Q$ $D = Z_P Z_Q$ $E = X_P^2$ $F = Z_P^2$ $H = bZ_P$
	reduce $A,\ B,\ C,\ D,\ E,\ F,\ H$	
	add. 1	$Z_Q = A - B$ $X_Q = A + B$ $C = C + 3D$ $A = E + 3F$
	mult. 2	$D = DX_Q$ $X_Q = C^2$ $Z_Q = Z_Q^2$ $A = A^2$ $B = FH$ $E = EX_P$ $F = X_P F$
	reduce $D,\ X_Q,\ Z_Q,\ A,\ B,\ E,\ F$	
	add. 2	$G = E + B - 3F$
	mult. 3	$D = bD$ $Z_Q = xZ_Q$ $B = X_P B$ $F = Z_P G$
	reduce $D,\ Z_Q,\ B,\ F$	
	add. 3	$X_Q = X_Q - 4D$ $X_P = A - 8B$ $Z_P = 4F$

(v) The operands $\tilde{x}$ in the basis $\tilde{B}_n$ are stored as $\tilde{\xi}_x = |\tilde{x}|\tilde{M}_i|_{\tilde{m}_i}^{-1}|_{\tilde{m}_i}$. 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 latency for the optimized algorithm was measured and is 263.0 ms (see Table 6) for the complete point multiplication,

TABLE 5. Type II algorithm's steps for the thread i.

Step in Figure 2	Computation for thread i
Step 1	$z_i = x_i y_i r_i \bmod m_i$
Step 2	$\tilde{\xi}_{z_i} = \left(\left\| \sum_{j=1}^{n} z_j M_j \right\|_{\tilde{m}_i} s_i + \tilde{\xi}_{x_i} \tilde{\xi}_{y_i} \right) t_i$
Step 3	$z_e = \left(\left\| \sum_{j=1}^{n} z_j M_j \right\|_{m_e} n_e + x_e y_e \right) \|M\|_{m_e}^{-1}$ $\alpha_z = \left(\left\| \sum_{j=1}^{n} \tilde{\xi}_{z_j} \tilde{M}_j \right\|_{m_e} - z_e \right) \|\tilde{M}\|_{m_e}^{-1}$ Compute set add. k operations mod m_e
Step 4	$z_i = \left\| \sum_{j=1}^{n} \tilde{\xi}_{z_j} \tilde{M}_j - \alpha_z \tilde{M} \right\|_{m_i}$ Compute set add. k operations mod $[m_i/\tilde{m}_i]$

TABLE 6. Total latency of EC point multiplication considering different approaches.

Type	Description	Latency[ms]
I	–	1665.9
II	Constant mem.	263.0
	Shared mem.	266.1
	μ look-up computing (shared mem.)	97.0
	μ look-up precomputing (const. mem.)	111.5
	Optimized reduction method (Version S)	33.4
	Uses Kawamura method (Version K)	29.2

regardless of the data transfers. We also exploited the effect of getting the 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 allows only 1. However, the results of this modification did not prove fruitful, since the latency was 1.2% higher (266.1 ms) as Table 6 suggests.

With the reduction of the RNS precision regarding the type I algorithm, a table look-up for the results μ is now possible, since for $n = 15$ only 960 bytes are required. We implemented two versions of the algorithm that use the table look-up stored in shared memory, computed at the beginning, and a pre-computed table loaded in constant memory. The obtained latency is 97.0 ms for the shared memory look-up approach, and 111.5 ms (15% higher) for the constant memory look-up as Table 6 suggests. These results suggest that the look-up is a good option, and also that the memory conflicts in accessing the constant memory start having a significant impact while the latency is decreasing.

The reduction operation over the RNS channels using the C '%' operation is known to be very demanding in the GPU platforms. 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

Algorithm 2 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:     $z' = cz'_H + z'_L$;
3: end while
4: $z = min(z', z' - m)$;
5: return $z$
```

accomplished recurring to Algorithm 2. Note that the step 4 of Algorithm 2 returns 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 avoiding conditions that can potentially create divergent sections of the program, thus serializing the computation. Employing this optimized reduction technique, the latency of the EC point multiplication algorithm is reduced to 33.4 ms (2.9 times lower than the previous best implementation) as Table 6 suggests.

The Kawamura *et al.* method for computing α in (5) can be used instead of the Shenoy *et al.* method by computing (9) as explained in Section 2.2. In the current GPU implementation we have to choose the RNS channel's precision $r = 16$. For this alternative conversion method, we must set the dynamic range $\tilde{M}$ of the basis $\tilde{B}_n$ as $\tilde{M}(1-\beta) > \hat{U}$, where $\hat{U}$ is the multiplication result prior to the conversion from $\tilde{B}_n$ to B_n that is known to be bounded by $\hat{U} < (n+2)N$. Therefore, we have to compute the lower bound for β since we are not interested in increasing the dynamic range $\tilde{M}$ for representing $\hat{U}$ due to this conversion. The bounds for β are $\Delta \leq \beta < 1$, where Δ is obtained from the approximation errors given by $\Delta = n(\epsilon + \delta)$, and n is the number of channels (see (10)). Since the moduli of the basis $\tilde{B}_n$ are obtained as $\tilde{m}_i = 2^r - \tilde{c}_i$, the minimum moduli magnitude is obtained when $\tilde{c}_i$ is maximum. We denote the modulus with higher $\tilde{c}_i$ as k. Hence, the value of ϵ is given by

$$\epsilon = \max\left(\frac{2^r - \tilde{m}_i}{2^r}\right) = \frac{2^r - k}{2^r}. \tag{18}$$

Following the same rational, the value of δ is obtained as

$$\delta = \max_i\left(\frac{\xi_i - \mathrm{trunc}_q(\xi_i)}{m_i}\right) \leq \frac{2^{r-q} - 1}{k}. \tag{19}$$

Recall that the $\mathrm{trunc}_q()$ function truncates the q most significant bits, and hence the value of δ is upper bounded when the numerator has only the $r-q$ less significant bits set.

Since we are interested in low values of β, we can assign

$$\beta = \Delta = \frac{n}{k}[2^r + 2^{r-q} - k - 1]. \tag{20}$$

The value of β decreases while q increases, although the required precision for computing α with this method also increases with q. Hence, we shall obtain the minimum value of q that allows complying $\tilde{M}(1-\beta) > \hat{U}$ without increasing the number of channels ($n = 15$) given by $M > (9(n+2))^2 N$.

Let us compute the value of q for this purpose. By knowing that $\tilde{M} > M > (9(n+2))^2 N$, $\hat{U} > (n+2)N$, and $\hat{U} < (1-\beta)\tilde{M}$, the following inequalities comply

$$\begin{aligned}(n+2)N &< (1-\beta)(9(n+2))^2 N \Leftrightarrow \\ &\Leftrightarrow \beta < 1 - \frac{1}{81(n+2)}.\end{aligned} \tag{21}$$

We find that (21) provides an upper bound for β in order to maintain the number of channels n (for this particular case $n = 15$). With this upper bound (20) can be used to seek the minimum value of q that allows β to comply (21). For this particular case ($n = 15$ and $r = 16$) $q = 5$. Once the value of q is set, the value of α in the step 3 of Table 5 can be obtained as

$$\alpha = \frac{\left\lfloor \sum_{i=1}^{n}(\mathrm{trunc}_q(\tilde{\xi}_i)/2^{r-q}) + \Phi \right\rfloor}{2^q}, \tag{22}$$

where $\Phi = \lfloor 2^q \beta \rfloor$. Note that the computation is obtained only with shift operations and an accumulation with maximum value $n(2^q - 1)$. With this method we get rid of all the computation over the extra modulus m_e, significantly reducing the size of the divergent computation section. For computing α with the Kawamura *et al.* method, the latency is shortened by 12% compared with the Shenoy *et al.* method (see Table 6).

5. EXPERIMENTAL EVALUATION

In this subsection, we discuss the presented GPU implementations and summarize the results for the proposed algorithms. Relative assessment is also presented by considering the related art.

5.1. Implementation and experimental results

Table 6 presents a summary for the obtained latency results for both proposed algorithms, and the different approaches for the type II algorithm. These results were obtained using the NVIDIA 285 GTX GPU and the release 3.1 of the CUDA tools. Table 7 summarizes the experimental setup employed in the measurements. Note that the stated latency results do not include the processing time that would be required for converting the inputs and outputs to and from RNS, respectively. Concerning the binary to RNS (forward) conversion of the inputs, each thread i involved in an EC point multiplication has to compute

TABLE 7. Experimental setup summary for the CUDA measurements.

Implementation	Characteristics
GPU	NVIDIA GeForce GTX 285
	30 multiprocessors (1476 MHz)
	1 GB video memory (1242 MHz)
	Nvidia CUDA version 3.1
	Nvidia CUDA-OpenCL version 3.1
Host	Intel Core 2 Quad Q9550 (2.83 GHz)
	2×2 GB Memory (DDR2 1066)
	PCI Express 16x

$g_i = |G|_{m_i}$ and $\tilde{g}_i = |G|_{\tilde{m}_i}$. Given that G can be split into n w-bit words ς_j such that

$$G = \sum_{j=1}^{n} 2^{(j-1)w} \varsigma_j, \tag{23}$$

the result g_i can be obtained as

$$g_i = \left| \sum_{j=1}^{n} \varsigma_j \tau_j \right|_{m_i}, \tag{24}$$

where the constant $\tau_j = |2^{(j-1)w}|_{m_i}$. Regarding that $\tilde{g}_i$ can be obtained with the same method expressed in (24), and since this method has less complexity than, for example, step 2 of Table 5, we can estimate that the forward conversion impact in the performance is less than that for a base extension. Thus, since a base extension is performed for each bit of the scalar involved in the EC point multiplication, for a 224-bit scalar the forward conversion penalty is $<1/224 = 0.44\%$. Concerning the RNS to binary (reverse) conversion, the final result X can be obtained from the residues $\tilde{x}_i$ by using (5). With this purpose, each thread i compute $\tilde{\xi}_i\ \tilde{M}_i$ which can be accomplished with $n-1$ multiply-and-add operations with w-bit entries. The value $-\alpha\tilde{M}$ can be computed with n multiply-and-add operations with w-bit entries. Finally, an accumulation of all the values $\tilde{\xi}_i\ \tilde{M}_i$ and $-\alpha\tilde{M}$ retrieves the final result. To compute the final accumulation, each thread i computes a digit with weight $2^{(i-1)w}$ with n additions and, finally, a single thread propagates the data that may be overlapping in the digits of different weights obtaining the final result with $n-1$ additions. Hence, the final accumulation latency corresponds to $2n-1$ additions. Therefore, the computation of $\tilde{\xi}_i\ \tilde{M}_i$ and the final accumulation have less complexity than step 4 and step 2 in Table 5, respectively. Summarizing, the reverse conversion is faster than a single base extension. Given the aforementioned considerations, the impact of both forward and reverse conversions are expected to be negligible in the performance ($<1\%$), and hence we do not consider them in the experimental results presented in this paper.

As suggested in Table 6, the type I algorithm results in a huge latency penalty due to the large dynamic range employed in this approach, increasing the complexity of the base extension procedure, which is not compensated by the increased parallelism.

The proposed type II algorithm without look-up tables for the result μ suggests a latency of 263.0 ms for the complete point multiplication regardless of the data transfers. The effect of getting the required constants from shared memory, copying them at a first moment resulted in 1.2% higher (266.1 ms) latency, despite the shared memory allows up to 16 simultaneous accesses while the constant memory only allows 1.

A table look-up for the results μ is possible for the type II algorithm, since for $n = 15$ we require 960 bytes ($2n(n+1)$ look-up entries) to store the table, which fit both shared and constant memory. The solutions with the look-up table stored in shared memory, computed at the beginning, and a precomputed table loaded in the constant memory were evaluated. A latency of 97.0 ms is obtained for the shared memory look-up approach, and 111.5 ms (15% higher) for the constant memory look-up. 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 when the latency decreases.

With the introduction of the optimized reduction method and by merging constants, the proposed type II algorithm provides a latency of 33.4 ms as shown in Table 6. The Kawamura method [14] (Version K) resulted in a latency figure improvement, since one EC point multiplication takes 29.2 ms to be computed, ∼12.6% lower than the version supported in the Shenoy method [13] (Version S) (see Table 6). The fact that the insertion of the Kawamura method result in lower latency, contradicts some former performance results concerning these two methods on GPUs [6, 26]. In [6] these two methods are evaluated in the base extension method, but separately from the application itself, in this case the RSA exponentiation. The results in [6] suggest that the Shenoy method is able to provide higher throughput, which can be measured as:

$$\text{Throughput} = \frac{\text{\# simultaneous EC point mults.}}{\text{Latency}}. \tag{25}$$

In [26] the base extension method is also evaluated for the EC point multiplication. The results in [26] suggested that the Shenoy method provides lower latency but the Kawamura method provides higher throughput. Although results in [26] are for the same application herein considered, they use a previous version of CUDA. Unfortunately, the CUDA profiling capabilities did not allow us to infer about the reason for the contradiction in the results between [26] and this work. Given that the correctness of the program is verified in this work and in [26], the reason for the observed deviation is due to the CUDA programming framework and run-time environment improvements/modifications, thus proprietary information would be required to verify this issue.

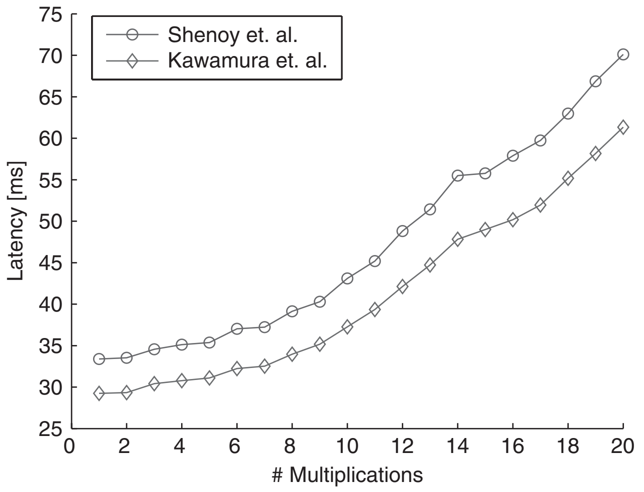


FIGURE 4. Latency for a different number of multiplications per block.

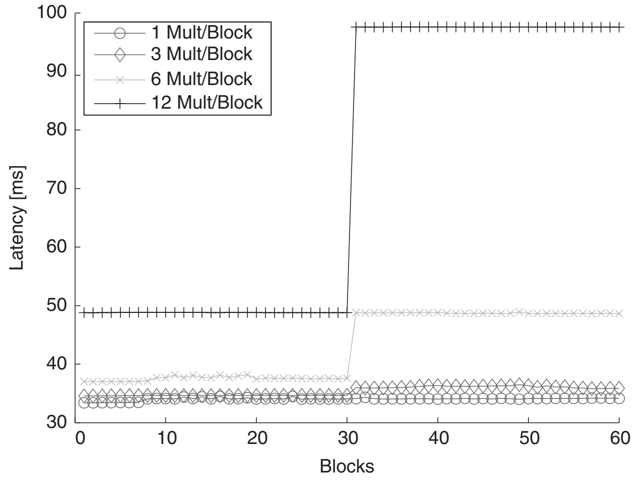


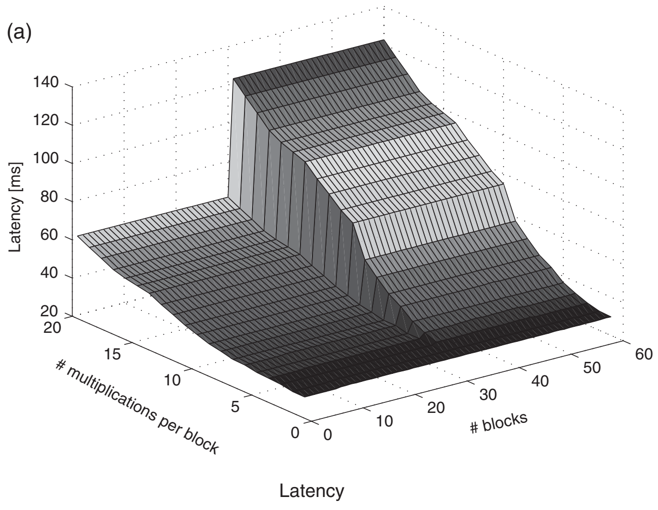
FIGURE 5. Version S latency vs. the number of blocks.

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 as 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 multiplications, which correspond to 300 threads per block. Figure 4 depicts the latency behavior while the number of multiplications per block is increasing. We compare the Version S (Shenoy method) and Version K (Kawamura method) methods since they present very close latency values for only one EC point multiplication. As explained, we expect a better performance for Version K. Actually, due to its expected efficiency, Version K motivated the design of cryptographic processors such as the one in [25]. As Fig. 4 suggests, the Version K performance is better than Version S for any number of simultaneous EC point multiplications. This result suggests that, despite the results in [6, 26], the Version K is able to provide both lower latency and higher throughput than Version S. Moreover, the difference between the latencies of both versions tends to increase with the number of simultaneous EC point multiplications (4.2 ms for 1 EC point multiplication and 8.8 ms for 20 simultaneous EC point multiplications), which suggest that the advantage of the Version K will be even more pronounced regarding the throughput metric.

We can expand our throughput also by taking advantage of the 30 existent multiprocessors in the employed GPU by using several blocks of threads. Figure 5 shows the Version S latency while expanding the number of blocks, for different number of EC point multiplications per block. Figure 5 shows the shape of a step with the discontinuity at the 30-block mark while increasing the number of threads per block. The number of multiprocessors in the GPU is 30, thus this step 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 6 shows the Version K latency and the throughput behavior for different combinations of the number of blocks and multiplications per block. From Fig. 6, we can confirm the existence of the step at the 30 blocks. Another result of Fig. 6 is that it is not worthwhile to use more than 30 blocks to achieve higher throughputs, especially for a large number of multiplications per block. The obtained results suggest a maximum throughput of 8579 operations (op)/s for the Version S, and 9827 op/s for the Version K. Version K can compute 600 EC point multiplications in 61.3 ms.

Despite the presented results referring to a 224-bit prime field, the proposed approach can be applied to any finite field given that the computational resources are not a limitation. The limitation for large finite fields arises from the available shared memory which, for the Version K implementation, must store twelve $2n$ variables (see Table 4) and the value μ introduced in Section 4.2 that has $2n^2$ entries; the μ is the main contributor for the memory limitation since it grows quadratically with the number of channels. Since the channel width $w = 16$ bits, which corresponds to 2 bytes, the total memory required for a given n is $4(n^2 + 12n)$ bytes. Regarding the 16 385 bytes of shared memory available in the GPU in Table 7, the maximum number of channels that can be supported is $n = 58$. Therefore, we estimate that the maximum dynamic range M must have <928 bits. Given that $M > (9(n+2))^2 N$ (see Section 4.3), the maximum size of the prime N that defines the underlying finite field should be 908 bits.



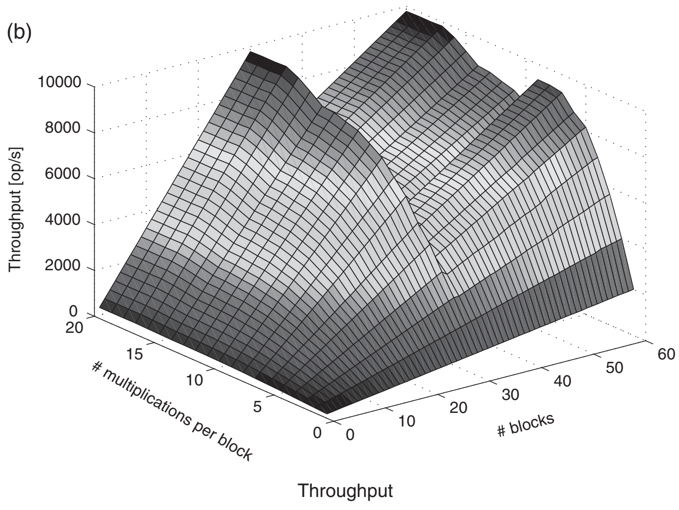


FIGURE 6. Version K latency and throughput vs. different combinations of blocks and multiplications per block.

In the obtained experimental results, the latency of the data transfers between the GPU and the host CPU and vice versa were not considered because it is negligible with regard to the processing latency in the GPU. Experimental results suggest that, for the experimental setup in Table 7, the latency of the data transfers ranges from 0.07%, for the configuration with 1 processing block and 1 EC point multiplication per block, to at most 0.19%, for the configuration with 60 processing blocks and 20 EC point multiplications per block.

5.2. Related art

The comparison with the related art, namely the experimental results, is not straightforward since different GPU platforms are employed, with different architectural characteristics and performances. Table 8 summarizes the related art performance figures compared with the work herein proposed. In this table we also included the results of the Version K algorithm for an OpenCL implementation to allow a performance comparison between the CUDA framework and the OpenCL implementation. This comparison suggests that similar latency values can be obtained but the throughput values are 1.3 times higher for the CUDA implementation. The decrease in the throughput figure for the OpenCL implementation results from overheads due to the enhanced flexibility of the implementation. It becomes more pronounced while the number of concurrent EC point multiplications in the GPU is increased.

TABLE 8. Related art comparison for 224-bit EC point multiplication.

References	Platform	Lat. (ms)	T.put (op/s)	Observations
[6]	8800 GTS	305	1413	
[27]	8800 GTS	–	3019	ECM results
[19]	9800 GX2	–	1972	
Ours	8800 GTS	30.3	3138	12 mul./block
[28]	295 GTX	–	5895	ECM results
Ours	285 GTX	29.2	9827	20 mul./block
Ours	285 GTX	29.2	7474	OpenCL impl.

In [6] 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 the schoolbook-type multiplication with reduction 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 suggests a latency of 305 ms and a throughput of 1413 operations/s.

In [27] EC point multiplication is evaluated on a GPU for integer factorization (ECM). 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. Authors extrapolated a throughput figure that is about 2.14 times higher than the one in [6]; however, results for the latency of an EC point multiplication are not provided.

A C++ library (PACE) that supports modular arithmetic on an Nvidia 9800GX2 GPU is evaluated in [19]. 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) [15]. For 224-bit precision, results suggest a throughput of 1972 op/s.

In [28] results for EC-based integer factorization suggest a throughput of 5895 op/s for a 192-bit Edward Elliptic Curve and an Nvidia 295 GTX GPU. The authors employ the Montgomery modular multiplication using a schoolbook algorithm and for the EC point multiplication the scalar is browsed in a double-and -add algorithm using a windowed integer recoding method.

Table 8 also present results for our best implementation running on an NVIDIA 8800GTS GPU, in order to perform a fair comparison with the other related art figures. However, due to register restrictions, it is not possible to apply the optimized implementation that runs on the 285 GTX platform. For the 8800 GTS, the implementation runs up to 12 multiplications per block. For this platform, Version K also offers better performance both in latency and throughput. Comparing the 9800 GX2 implementation with the 8800 GTS one, we have to bear in mind that the GPU of the former has more computational resources than the latter.

The proposed implementation surpasses in an order of magnitude the latency figures of the related art. The proposed implementation provides 59% more throughput than [19] with our 8800 GTS implementation. Moreover, 4% more throughput is achieved than the extrapolation described in [27] and 122% more throughput than [6]. Also, despite using a higher-end GPU, a smaller curve and an EC that requires less modular multiplications to perform the EC point multiplication, the authors in [28] report a throughput figure that is about 1.6 times lower than the one of the implementation herein proposed with the 285 GTX GPU.

Table 9 presents results reported in the related art concerning platforms other than GPUs, with standard ECs otherwise specified. These results were gathered after a thorough review of the related art and are, to the best of our knowledge, the most competitive in terms of latency and throughput for EC point multiplication. The purpose of this table is neither to directly compare the presented results nor relatively assess the herein achieved ones, since different platforms have different computational capabilities and different aims. While ASIC implementations target both low power and static computation figures, General Purpose (GP) processors target an inexpensive and flexible implementation. FPGAs target a compromise between the two previous implementations. Also, this table does not introduce considerations on the circuit area and power efficiency of the implementations, which results in the ASIC implementations to suggest lower performance regarding other platforms such as FPGAs. Moreover, not only the devices that support the implementations but also the algorithmic approaches vary. Regarding the EC arithmetic, several EC types can be used (e.g. Edward and Montgomery ECs [29, 30]) that are known to reduce the number of the required field operations, namely the field multiplications. Furthermore, different projective coordinates can be used to represent EC points that change the algorithmic demands in terms of field operations. Also, other algorithms to perform EC point multiplication can be used instead of the Montgomery Ladder, such as algorithms that involve recoding of the scalar in order to reduce the number of point additions when a double and add approach is used [29]. Despite the possible approaches, the work proposed herein focus is on the acceleration/parallelization of the field operations using RNS, addressing the Montgomery Ladder algorithm as an example to obtain the EC point multiplication over standard curves, which has the advantage of being resistant against time attacks. Thus, the same concepts and methodology herein presented can be easily applied to any other EC algorithm. The work in [31] proposes an RNS implementation of the EC point multiplication that supports the finite field multiplication in the Horner scheme. Still, it is not possible to relatively assess the results in [31] because they refer to an FPGA platform. Considering the aforementioned reasons, these results are intended only to provide a state of the art overview just to

TABLE 9. Related art comparison for EC point multiplication supported in platforms other than GPUs.

References	Platform	Lat. (ms)	T.put (op/s)	p size (bits)	Details
[32]	ASIC	0.095	10 526	256	0.18 μm CMOS technology
[33]	ASIC	1.01	990	256	0.13 μm CMOS technology
[31]	FPGA	4.08	302	224	Xilinx Virtex, RNS approach using Horner multiplication scheme
[34]	FPGA	0.365	37 700	224	Xilinx Virtex-4, employing embedded DSPs
[29]	GP proc.	0.089	22 472	256	2.6 GHz AMD Opt., Twisted Edward EC
[29]	GP proc.	0.105	19 048	256	2.6 GHz AMD Opt., Weierstrass EC, Jacobian coord.
[30]	GP proc.	0.128	15 600	255	2.4 GHz AMD Opt., Montgomery EC
[35]	Cell	0.218	27 474	255	Sony Playstation 3, Montgomery EC
[36]	DSP	1.69	592	224	Texas Instruments C6416 DSP
[37]	μ cont.	11.25	89	160	416 MHz Imote 2, TinyECC library
[38]	GP proc.	0.717	5574	224	3.0 GHz AMD Ph. II X4, Crypto++ Library 5.6.1
[39]	GP proc.	0.275	14 509	224	3.0 GHz AMD Ph. II X4, MIRACL Library 5.4.2
[40]	GP proc.	0.714	5600	224	3.2 GHz AMD Ph. II X4, eBACS result (OpenSSL based)
–	GP proc.	1.093	3703	224	3.0 GHz AMD Ph. II X4, Alg. 1 using GMP Library 5.0.1

allow an easier contextualization of the work herein proposed and its performance metrics.

Regarding Table 9, we can identify implementations supported in platforms other than GPUs that offer better latency figures than the proposed GPU implementation. This is due to the more complex datapath in these implementations, which are likely to suit in a more efficient way the requirements of the irregular EC point multiplication's flow. Concerning the throughput metric, some implementations beat the GPU's figures. Note that we estimate the throughput by assuming that the multi-core processors are able to provide an EC point multiplication in the same latency for each one of their single cores. Filtering the non-library-based implementations supported on general purpose platforms, we identify better throughputs in [29] (22 472 op/s) and in [35] (27 474 op/s). Nevertheless, we have to bear in mind that these implementations support special ECs (Montgomery and Twisted Edward curves) that allow a reduction of the field operations required to accomplish an EC point multiplication, while the results for the work herein proposed were obtained using standardized ECs [7]. Also, the implementation in [29] not only is obtained from a highly optimized assembly description, but also does not have the advantages of the Montgomery Ladder concerning side channel and time attacks. Instead, a scalar recoding method is used that allows avoiding additions in a double- and -add algorithm to compute the EC point multiplication. In Table 9, we also present performance figures for three well-known optimized libraries with support for EC cryptography. We compiled the most recent version of both libraries Crypto++ [38] and MIRACL [39] for a system based on the quad-core 3.0 GHz AMD Phenom II general purpose processor using the GNU C and C++ compiler version 4.3.1, while the results for the OpenSSL-based library were obtained from the asymmetric cryptography benchmark repository eBACS [40] for a 224-bit standard curve. Results suggest that the proposed GPU solution supersede by up to 1.7 times the throughput metric of the library-based implementations except for the MIRACL library. In the last line of Table 9 is also included an implementation that does not use the RNS approach in the CPU. It results from a direct implementation of Algorithm 1 with the point addition and doubling in (2), using the GNU Multiple Precision Arithmetic Library (GMP), version 5.0.1 [41]. GMP is a library for multi-precision arithmetic that is highly optimized for several commercial CPUs, including the one in Table 10. In this implementation the available parallelism of the CPU cores is exploited using OpenMP [42], which is an API for programming shared memory parallel systems. Nevertheless, for the latency result, only one thread is running the application since no cooperation between threads exists while computing the EC point multiplication. This last implementation is the one with closer characteristics regarding the presented GPU implementation (same EC point multiplication algorithm, projective coordinates, and curve). Hence, it is the one that better allows for a relative assessment with the work herein proposed: despite a higher latency, the GPU implementation herein proposed provides more than 2.6 times more throughput than the result in Table 9.

TABLE 10. Experimental setup summary for the OpenCL measurements.

Implementation	Characteristics
CPU	AMD Ph. II X4 945 4-Core (3.0 GHz) OpenSUSE 11.0 Operating System 2 × 2 GB Memory (DDR3 1333) 64-bit datapath ATIstream OpenCL support

5.3. Generalization for other architectures

In Section 5, we presented the RNS-based algorithms for EC point multiplication and experimental evaluation on a GPU. The presented analysis allows pointing out the best direction toward an efficient implementation of the EC point multiplication algorithm: (i) a smaller dynamic range is desired since a huge parallelization of a single multiplication should not pay off the increase of the base extension complexity; (ii) it is worthwhile to precompute constants (e.g. the μ result) in the device instead of transferring them from memory, taking advantage of the enhanced computational power of the device regarding the memory access efficiency; and (iii) the Kawamura method (Version K) suggests more interesting performance, in terms of throughput and latency, than the Shenoy method (Version S). Following the aforementioned guidelines, in this Section we generalize the obtained results for other devices. Also, we develop an analysis of the scalability of the proposed algorithms for ECs supported on differently sized underlying finite fields. The presented analysis is based only on the versions S and K implementations, since these are the ones that suggest better performance.

In order to obtain a generic implementation of the proposed algorithms that target different devices, we programmed the Versions S and K of the algorithms with OpenCL [20]. In this generalization effort, two different devices are targeted: (i) a GPU and (ii) a multi-core CPU. Table 10 describes the experimental setup for the CPU. For the GPU, the experimental setup is the same as in Table 7.

Figure 7 depicts the latency figures for both the CPU and the GPU, as well as for several standard curves based on underlying fields with different sizes [7]. In these figures we also included the performance figures of the Montgomery Ladder Algorithm 1 implemented with the GMP Library and OpenMP reported in Table 9. In Fig. 7 we observe that Version K offers more interesting latency metrics than Version S, which confirms the results obtained for the CUDA implementation in Table 6. We do not intend to compare the CPU with GPU implementation since different technologies, aimed at different applications,

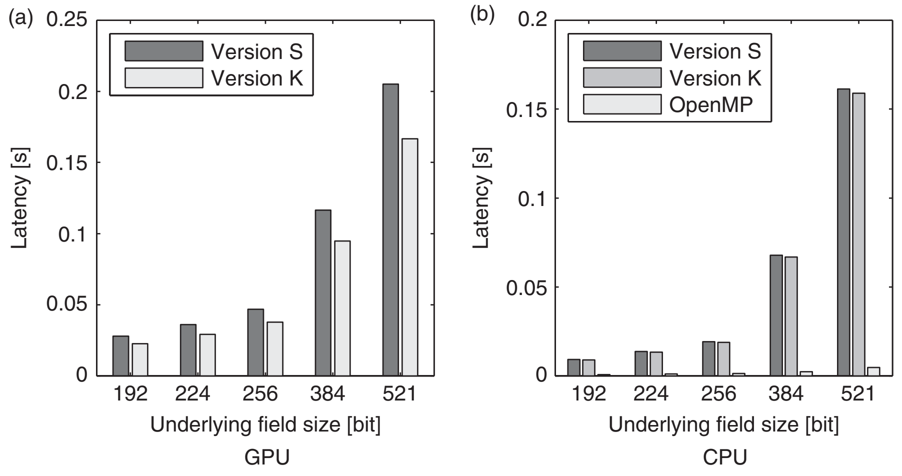


FIGURE 7. Latency figures of the OpenCL implementations for both CPU and GPU devices.

are used. Nevertheless, from Fig. 7, we observe that, for the Version K and the smaller field sizes, the CPU latency is up to 2.5 smaller than that of the GPUs, but for the larger field sizes, the results converge and the CPU latency is only 1.05 smaller. This suggests that the computation time does not increase with the field size in the GPU as fast as in the CPU. This effect may also be related with the thread switching in the CPU that introduces more penalty than in the GPU, where, for only one point multiplication per multiprocessor, there are also very few conflicts between threads. The commutation between threads can also be a reason for the high latency of the RNS approach regarding the OpenMP implementation in the CPU (up to 32 times higher for the 521-bit configuration).

The OpenCL (and OpenMP) maximum throughput metrics are presented in Fig. 8. The throughput values were obtained by increasing the number of point multiplications per OpenCL thread group (thread block in the CUDA terminology), while the available core's local memory (shared memory in CUDA) is enough to fit the required variables. The number of groups (blocks in CUDA) is the same as the number of available cores in the device (30 in the GPU and 4 in the CPU). The results in Fig. 8 suggest that the Version K in both CPU and GPU devices provides higher throughput (up to 1.19 and 1.02 more throughput for the GPU and CPU, respectively). Also, regarding the CUDA implementation, the OpenCL implementation provides 1.31 less throughput for the 224-bit configuration. These results also suggest that the GPU can exploit more the RNS capabilities providing a throughput

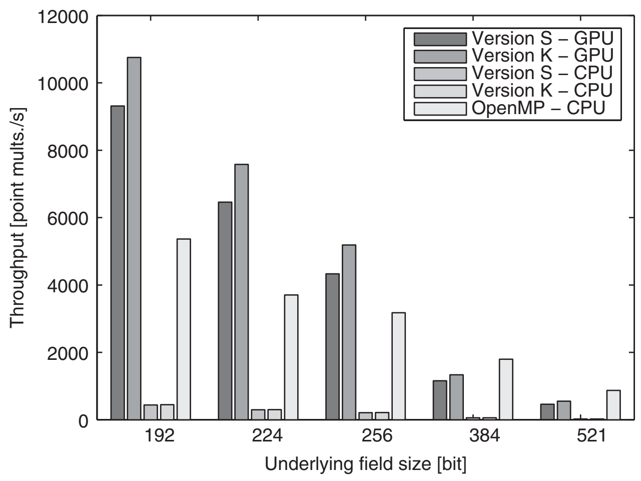


FIGURE 8. Throughput figures of the OpenCL implementations for both CPU and GPU devices.

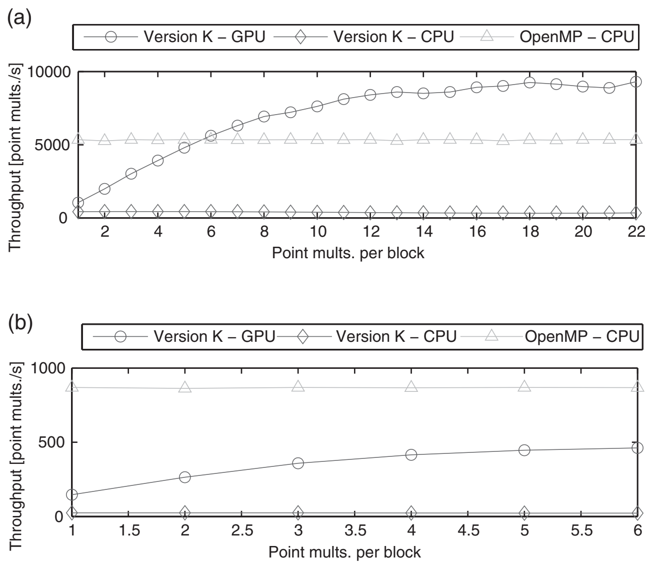


FIGURE 9. Throughput figures of the OpenCL implementations for both CPU and GPU devices, and several EC point multiplications per block. (**a**) 192-bit underlying field and (**b**) 521-bit underlying field.

more than an order of magnitude higher than that of the CPU's. It can also be observed that the GPU implementation provides up to two times more throughput for the smaller field sizes and up to 1.5 times less throughput for the larger field sizes when compared with the OpenMP implementation. This result suggests the existence of a break point from which the parallelization obtained by the RNS approach does not pay off the increase of the base extension complexity. This effect is the same behind the discussion in Section 5. This discussion concludes that the type I algorithm is not worthwhile regarding the type II algorithm approaches. The results in Fig. 8 also suggest that the CPU is not a suitable platform to take advantage of the RNS approach, providing up to 32 times less throughput for OpenCL implementations regarding the OpenMP approach, since it does not provide enough parallelism. In Fig. 9 we can observe the throughput metric behavior when the multiplications per OpenCL group increase for the smallest and largest underlying finite field employed in our tests. These results show that the increasing of the number of multiplications per block only affects the GPU implementation in both field sizes, since for the CPU implementations the behavior resembles a horizontal line. These results acknowledge the lack of parallelism in the CPU, which does not allow the RNS approach to be profitable.

6. CONCLUSIONS

In this paper, we have proposed parallel algorithms for EC point multiplication on a GPU by adopting a new RNS approach. This RNS approach achieves a higher level of parallelism, and thus higher performance in the massive parallel architecture of the GPU. We tested different implementation versions, which attempt to exploit different properties of the GPU platform.

Experimental results suggest a maximum throughput of 9827 EC point multiplications per second and minimum latency of 29.2 ms, using an Nvidia 285 GTX GPU for an EC curve supported on a 224-bit underlying prime field. Moreover, up to an order of magnitude of reduction in latency and up to 122% throughput improvement are obtained regarding the results in the related art. The gains of the proposed implementation result from the higher utilization of the multiprocessor cores, by computing up to 20 simultaneous EC point multiplications in each GPU multiprocessor.

We also analyzed the scalability and the results for other architectures, programming the algorithms with OpenCL. Results suggest that the employed RNS approach is more advantageous for lower size underlying fields. Also, comparison of the different architectures' allowed concluding that a commercial CPU (with 4 cores) does not provide enough parallelism/thread commutation efficiency for a worthwhile implementation of the RNS approach. However, the number of cores is expected to significantly increase in general purpose processors, thus making the proposed algorithms also suitable for this type of multi-core processors.

ACKNOWLEDGEMENTS

The authors would like to thank Prof. Pedro Trancoso from the University of Cyprus for his availability in testing the proposed implementation in the Nvidia 8800 GTS GPU.

FUNDING

This work was supported by FCT (Fundação para a Ciência e Tecnologia) through the PIDDAC Program funds (INESC-ID multi annual funding) and Pessoa PHC in Portugal, and by Égide in France.

REFERENCES

[1] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J. and Phillips, J. (2008) GPU Computing. *Proc. IEEE*, **96**, 879–899.

[2] Preis, T., Virnau, P., Paul, W. and Schneider, J.J. (2009) GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *J. Comput. Phys.*, **228**, 4468–4477.

[3] Takahashi, T. and Hamada, T. (2009) GPU-accelerated boundary element method for Helmholtz' equation in three dimensions. *Int. J. Numer. Methods Eng.*, **80**, 1295–1321.

[4] Manavski, S. and Valle, G. (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment. *BMC Bioinf.*, **9**, 1295–1321.

[5] Manavski, S. (2007) CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. *Proc. IEEE Int. Conf. Signal Processing and Communications 2007—ICSPC 2007*, Dubai, United Arab Emirates, November 24–27, pp. 65–68. IEEE.

[6] Szerwinski, R. and Güneysu, T. (2008) Exploiting the Power of GPUs for Asymmetric Cryptography. In Oswald, E. and Rohatgi, P. (eds), *Proc. Workshop on Cryptographic Hardware and Embedded Systems 2008 – CHES 2008*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

[7] FIPS PUB 186-3 (2009) *Federal Information Processing Standards Publication 186-3: Digital Signature Standard*. National Institute of Standards and Technology. Gaithersburg, MD, USA.

[8] Bajard, J.-C., Didier, L.-S. and Kornerup, P. (2001) Modular Multiplication and Base Extensions in Residue Number Systems. *Proc. 15th IEEE Symp. Computer Arithmetic 2001—ARITH 2001*, Vail, CO, USA, June 11–17, pp. 59 –65. IEEE.

[9] Blake, I., Seroussi, G., Smart, N. and Cassels, J.W.S. (2005) *Advances in Elliptic Curve Cryptography*, London Mathematical Society Lecture Note Series. Cambridge University Press, New York, NY, USA.

[10] Bajard, J., Duquesne, S. and Ercegovac, M. (2010) Combining leak–resistant arithmetic for elliptic curves defined over F_p and RNS representation. *IACR Cryptology ePrint Archive*, **311**, 1–25.

[11] Szabo, N. and Tanaka, R. (1967) *Residue Arithmetic and its Applications to Computer Technology*. McGraw-Hill, New York, NY, USA.

[12] Yassine, H. and Moore, W. (1991) Improved mixed-radix conversion for residue number system architectures. *IEE Proc.-G: Circuits Devices Syst.*, **138**, 120–124.

[13] Shenoy, A. and Kumaresan, R. (1989) Fast base extension using a redundant modulus in RNS. *IEEE Trans. Comput.*, **38**, 292 –297.

[14] Kawamura, S., Koike, M., Sano, F. and Shimbo, A. (2000) Cox-Rower Architecture for Fast Parallel Montgomery Multiplication. In Preneel, B. (ed.), *Proc. EUROCRYPT 2000—Advances in Cryptology*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

[15] Kaya Koç, Ç., Acar, T. and Kaliski, J., B.S. (1996) Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, **16**, 26–33.

[16] NVIDIA CUDA C Programming Guide (2010) *CUDA Toolkit 3.1*. NVIDIA Corporation. Santa Clara, CA, USA.

[17] Bernstein, D. (2008) Curve25519: New Diffie–Hellman Speed Records. In Yung, M., Dodis, Y., Kiayias, A. and Malkin, T. (eds), *Public Key Cryptography—PKC 2006*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

[18] Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J. (2008) NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, **28**, 39–55.

[19] Giorgi, P., Izard, T. and Tisserand, A. (2010) Comparison of Modular Arithmetic Algorithms on GPUs. In Chapman, B., Desprez, F., Joubert, G.R., Lichnewsky, A., Peters, F. and Priol, T. (eds), *Parallel Computing: From Multicores and GPU's to Petascale—Advances in Parallel Computing*. IOS Press, Amsterdam, The Netherlands.

[20] OpenCL web page - Khronos Group. Available online: http://www.khronos.org/opencl/.

[21] OpenCL Specification (2009) *The OpenCL Specification—Version 1.0.48*. Khronos OpenCL Working Group. Available online.

[22] OpenCL Programming Guide for the CUDA Architecture (2010) *CUDA Toolkit 3.1*. NVIDIA Corporation. Santa Clara, CA, USA.

[23] ATI Stream Technology. Available online: http://www.amd.com/stream.

[24] Bajard, J., Meloni, N. and Plantard, T. (2005) Efficient RNS Bases for Cryptography. *Proc. 17th IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation*, Paris, France, July 11–15, pp. 1–7. IMACS.

[25] Guillermin, N. (2010) A High Speed Coprocessor for Elliptic Curve Scalar Multiplication over Fp. In Mangard, S. and Standaert, F.-X. (eds), *Proc. Workshop on Cryptographic Hardware and Embedded Systems 2010—CHES 2010*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

[26] Antão, S., Bajard, J.-C. and Sousa, L. (2010) Elliptic Curve Point Multiplication on GPUs. *Proc. 21st IEEE Int. Conf. Application-Specific Systems Architectures and Processors—ASAP 2010*, Rennes, France, July 7–9, pp. 192–199. IEEE.

[27] Bernstein, D., Chen, T.-R., Cheng, C.-M., Lange, T. and Yang, B.-Y. (2009) ECM on Graphics Cards. In Joux, A. (ed.), *Proc. EUROCRYPT 2009—Advances in Cryptology*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

[28] Bernstein, D., Chen, H., Chen, M., Cheng, C., Hsiao, C., Lange, T., Lin, Z. and Yang, B. (2009) The Billion-Mulmod-Per-Second PC. *Proc. Special-Purpose Hardware for Attacking Cryptographic Systems Workshop 2010- SHARKS 2010*, Lousanne, Switzerland, September 9–10, pp. 131–144. SHARKS.

[29] Longa, P. and Gebotys, C. (2010) Analysis of efficient techniques for fast elliptic curve cryptography on x86-64 based processors. *IACR Cryptology ePrint Archive*, **335**, 1–34.

[30] Gaudry, P. and Thomé, E. (2007) The mpFq Library and Implementing Curve-Based Key Exchanges. *Proc. Software Performance Enhancement for Encryption and Decryption Meeting—SPEED 2007*, Amsterdam, The Netherlands, June 11–12, pp. 49–64. ECRYPT.

[31] Schinianakis, D., Fournaris, A., Michail, H., Kakarountas, A. and Stouraitis, T. (2009) An RNS implementation of an F_p elliptic curve point multiplier. *IEEE Trans. Circuits Syst. I: Regular Papers*, **56**, 1202–1213.

[32] Zhang, X. and Li, S. (2007) A High Performance ASIC Based Elliptic Curve Cryptographic Processor over GF(p). *Proc. 2nd Int. Design and Test Workshop—IDT 2007*, Cairo, Egypt, December 16–18, pp. 182–186. IEEE.

[33] Chen, G., Bai, G. and Chen, H. (2007) A high-performance elliptic curve cryptographic processor for general curves over GF(p) based on a systolic arithmetic unit. *IEEE Trans. Circuits Syst. II: Express Briefs*, **54**, 412 –416.

[34] Güneysu, T. and Paar, C. (2008) Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In Oswald, E. and Rohatgi, P. (eds), *Proc. Workshop on Cryptographic Hardware and Embedded Systems 2008—CHES 2008*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

[35] Costigan, N. and Schwabe, P. (2009) Fast Elliptic-Curve Cryptography on the Cell Broadband Engine. In Preneel, B. (ed.), *Proc. AFRICACRYPT 2009—Progress in Cryptology*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

[36] Yan, H., Shi, Z. and Fei, Y. (2009) Efficient Implementation of Elliptic Curve Cryptography on DSP for Underwater Sensor Networks. *Proc. Workshop on Optimizations for DSP and Embedded Systems—ODES 2009*, Chamonix, France, April 2, pp. 1–9. ODES.

[37] Liu, A. and Ning, P. (2008) TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. *Proc. 7th Int. Conf. Information Processing in Sensor Networks—IPSN 2008*, St. Louis, MO, USA, April 22–24, pp. 245–256. IEEE Computer Society.

[38] Crypto++ Library. Available online: http://www.cryptopp.com/.

[39] Multiprecision Integer and Rational Arithmetic C/C++ Library—MIRACL. Available online: http://www.shamus.ie/.

[40] ECRYPT Benchmarking of Cryptographic Systems. Available online: http://bench.cr.yp.to/.

[41] The GNU Multiple Precision Arithmetic Library. http://gmplib.org/.

[42] The OpenMP API specification for parallel programming. Available online: http://openmp.org.