An Efficient Many-Core Architecture for Elliptic Curve Cryptography Security Assessment

Marco Indaco[2], Fabio Lauri[2], Andrea Miele[1], Pascal Trotta[2]

[1] LACAL, EPFL, Lausanne, Switzerland
[2] Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy

Abstract. Elliptic Curve Cryptography (ECC) is a popular tool to construct public-key crypto-systems. The security of ECC is based on the hardness of the elliptic curve discrete logarithm problem (ECDLP). Implementing and analyzing the performance of the best known methods to solve the ECDLP is useful to assess the security of ECC and choose security parameters in practice. We present a novel many-core hardware architecture implementing the parallel version of Pollard's rho algorithm to solve the ECDLP. This architecture results in a speed-up of almost 300% compared to the state of the art and we use it to estimate the monetary cost of solving the Certicom ECCp-131 challenge using FPGAs.

1 Introduction

Elliptic Curve Cryptography (ECC) was introduced in the mid 1980s [11, 13] by Koblitz and Miller and over the last decades it has become a standardized [16] tool to build public-key crypto-systems. For instance ECC is employed for secure communication in popular applications like Bitcoin, TLS and SSH [3]. The security of ECC is based on the difficulty of solving the elliptic curve discrete logarithm problem (ECDLP). The best publicly known algorithm to attack this problem in practice is the parallel Pollard rho [17].

Analyzing the performance of Pollard rho in practice and solving large instances of the ECDLP is useful to estimate the security of ECC and choose the parameters of deployed crypto-systems appropriately. The Certicom challenges [5] have been published with the aim of providing a public litmus test for assessing the performance of ECDLP attacks.

Our work targets elliptic curves defined over "generic" prime fields $\mathbf{F}_p$ where the prime p is assumed to have no special form. Both hardware [9, 10] and software [4, 1] implementations of Pollard rho for the ECDLP on prime fields have been proposed in the literature. The architecture proposed in [9] has been implemented on Xilinx Spartan-3 FPGAs and elliptic curve prime group sizes ranging from 64 to 160 bits have been considered to assess its performance. The implementation proposed in [10] targets the secp112r1 curve from Certicom defined over a prime field of a special form. The architecture is based on a modular multiplication unit optimized to be efficiently mapped on embedded DSP resources of a Xilinx Virtex-5 FPGA. These works have demonstrated that Field Programmable Gate Arrays (FPGAs) are suitable accelerators for Pollard rho.

We present a novel pipelined many-core architecture implementing the parallel version of Pollard rho for elliptic curves over generic prime fields using the negation map speed-up and fruitless cycle handling [17]. The size of the prime field is configurable at synthesis time and the implementation does not rely on a specific target device architecture. We analyze the performance of our architecture when implemented on different FPGA families. Compared to the state of the art we obtain a speed-up of almost 300%. We also provide cost estimates for solving the Certicom challenge ECCp-131 using FPGA clusters.

This paper is organized as follows. Section 2 introduces elliptic curves and Pollard rho. Section 3 describes the proposed architecture and its optimization. Section 4 describes implementation details and the experimental results. Section 5 presents the conclusion.

2 Preliminaries

In this section we introduce some basic notions about elliptic curves and we describe the parallel version of Pollard's rho algorithm with the negation map and fruitless cycle handling used in our implementation.

2.1 Elliptic curves over prime fields

We focus on finite fields whose number of elements is a prime number larger than 3. We denote with F_p the finite field containing p elements where $p \in \mathbf{Z}_{>3}$ is prime. We denote with k the bit-size of p, i.e., $k = \lfloor \log_2 p \rfloor + 1$.

As described in [12], a pair $(a, b) \in F_p^2$, for which $4a^3 + 27b^2 \neq 0$, defines an *elliptic curve* E over F_p. We denote by $E(F_p)$ *set of points* of E and by O the point at infinity. The set $E(F_p)$ has the structure of an *abelian group* with the group law defined as follows:

- *Identity*: $O + P = P + O = P$ for all $P \in E(F_p)$.
- *Negative*: Given $P = (x_1, y_1) \neq O$ and $Q = (x_2, y_2) \neq O$ we have that $P+Q = O$ if and only if $x_1 = x_2$ and $y_1 = -y_2$; thus $-(x, y) = (x, -y)$.

- *Addition*: Given $P \neq Q$ and $\lambda \in F_p$ with $\lambda = (y_1 - y_2)/(x_1 - x_2)$then $P + Q = R = (x_3, y_3)$ with $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = \lambda(x_1 - x_3) - y_1$.

The arithmetic operations in the above formulae are addition, subtraction, multiplication and inversion modulo p.

2.2 Parallel Pollard rho for the ECDLP

We focus on prime order subgroups of $E(F_p)$. We will denote such a subgroup having prime order q and generator $P = (x, y) \in E(F_p)$ by $\langle P \rangle$. Given some $Q \in \langle P \rangle$, the *elliptic curve discrete logarithm problem* (ECDLP) is to find $h \in \mathbf{Z}/q\mathbf{Z}$ such that $Q = hP$.

We use the parallel version of Pollard rho with *r-adding walks* (where r is assumed to be a power of 2), *distinguished points* and the *negation map* [17, 18, 7]. A distinguished point, is a point in $\langle P \rangle$ having the least significant d bits of the x coordinate all equal to zero for a small positive integer d. An r-adding walk is defined as follows. Precompute r points $F_j = c_j P + d_j Q = (x_j, y_j)$, for random non-zero $c_j, d_j \in \mathbf{Z}/q\mathbf{Z}$ and $0 \leq j < r$ (store each point F_j and its coefficients c_j, d_j in a look-up table). The first point in the walk is selected as $P_0 = a_0 P + b_0 Q$ for random (but known) integers $a_0, b_0 \in [1, q - 1]$ and at step $i \geq 0$ the next point is computed as $P_{i+1} = f(P_i)$ with the following iteration function:

$$f(P_i) = P_i + F_{\ell(P_i)}, \tag{1}$$

where $\ell \in [0, r - 1]$ is the non negative integer represented by the $\log_2 r$ least significant bits of the x coordinate of P_i. It is easy to also keep track of two integer multipliers $a, b \in \mathbf{Z}/q\mathbf{Z}$ such that $P_i = aP + bQ$. Parallel walks start from different random initial points but all of them use the same look-up table. This choice implies that once two independent walks reach the same point then they will eventually hit the same distinguished point. If at step $i \geq 0$ one walk hits a distinguished point P_i, it reports P_i to a central processor. Once the central processor has received the same distinguished point twice, it finds a collision, i.e., four integers $a, b, a', b' \in \mathbf{Z}/q\mathbf{Z}$ such that $aP + bQ = a'P + b'Q$ and $b' - b \not\equiv 0 \bmod q$, then the value $(a - a')/(b' - b) \bmod q$ is a solution of the ECDLP.

The iteration function (1) can be modified to use the negation map and reduce the number of expected steps to find a collision by a factor $\sqrt{2}$. When using the negation map additive walks can get stuck in a *fruitless* cycle in which the same sequence of points is produced indefinitely preventing the walk from performing useful steps [18, 7]. Fruitless cycles of all lengths that are a multiple of 2 can occur. The occurrence of the most frequent fruitless cycles can be reduced using a look-ahead technique, but as fruitless cycles will eventually occur mechanisms of detection and escape are usually needed [18].

We use the negation map and the 2-cycle reduction technique presented in [2], which requires an additional look-up table containing r points $F'_j = c'_j P + d'_j Q = (x'_j, y'_j)$ for random non-zero $c'_j, d'_j \in \mathbf{Z}/q\mathbf{Z}$ and $0 \leq j < r$. This technique reduces the probability of entering a 2-cycle from $1/(2r)$ to $1/(2r^3)$ and this makes 4-cycles the most likely to occur with probability $(r-1)/(4r^3)$ (i.e., a 4-cycle appears on average every $4r^3/(r-1)$ steps). We do not implement cycle detection and escape in our FPGA architecture as it would add significant architectural complexity. We assume that cycle detection and escape is performed periodically on the host system (for instance the processor embedded in most FPGAs) every w iterations (see Section 4 for an explanation how the value w is selected following the approach from [2]), after which the current point of each walk is updated accordingly (see subsection 3.1 for the practical details).

The Pollard rho iteration we implement follows from the above description. Each walk repeats the iteration composed of the following steps, until a collision is found:

1. Given $P_i = (x_i, y_i)$ and $\ell = x_i \bmod r$, set the point $S = (x_s, y_s)$ equal to $F_\ell = (x_l, y_l)$. Or set the point S equal to $F'_\ell = (x'_l, y'_l)$ if the second table was enabled at the previous iteration. Set the values a_s and b_s equal to c_ℓ and d_ℓ or c'_ℓ and d'_ℓ accordingly. Compute $P_{i+1} = P_i + S = (x_{i+1}, y_{i+1})$ (addition formula in Section 2.1). Given the two integer multipliers a, b such that $P_i = aP + bQ$, compute $a \leftarrow a + a_s \bmod q$ and $b \leftarrow b + b_s \bmod q$ so that $P_{i+1} = aP + bQ$ (recall that $P_0 = a_0 P + b_0 Q$).
2. (negation map)
   (a) Compute $-y_{i+1} = p - y_{i+1} \bmod p$.
   (b) If y_{i+1} is even set $P_{i+1} \leftarrow -P_{i+1} = (x_{i+1}, -y_{i+1})$ and set $a \leftarrow -a \bmod q$ and $b \leftarrow -b \bmod q$.
3. (reduction) If the second table is not enabled then if $\ell(P_i) = \ell(P_{i+1})$ set $P_{i+1} \leftarrow P_i$ and enable the second table for the next iteration. Otherwise if the second table is enabled the current step is skipped.
4. If $x_{i+1} \bmod 2^d = 0$ report the (distinguished) point P_{i+1} to the central processor.

   If w iterations have been run, report current point to central processor for cycle detection and escape.

3 Proposed architecture

The proposed many-core architecture relies on a pipelined core implementing the parallel version of Pollard rho. In this section we discuss design and implementation of a single core and of the final many-core architecture.

3.1 Single pipeline multi walk core

The architecture of a *single pipeline multi walk* (SPMW) core is depicted in Figure 1. Although each walk exhibits an iterative behavior, the parallel version of Pollard's rho algorithm runs independent walks. We exploit this behavior by interleaving the execution of several independent walks in the same hardware pipeline.

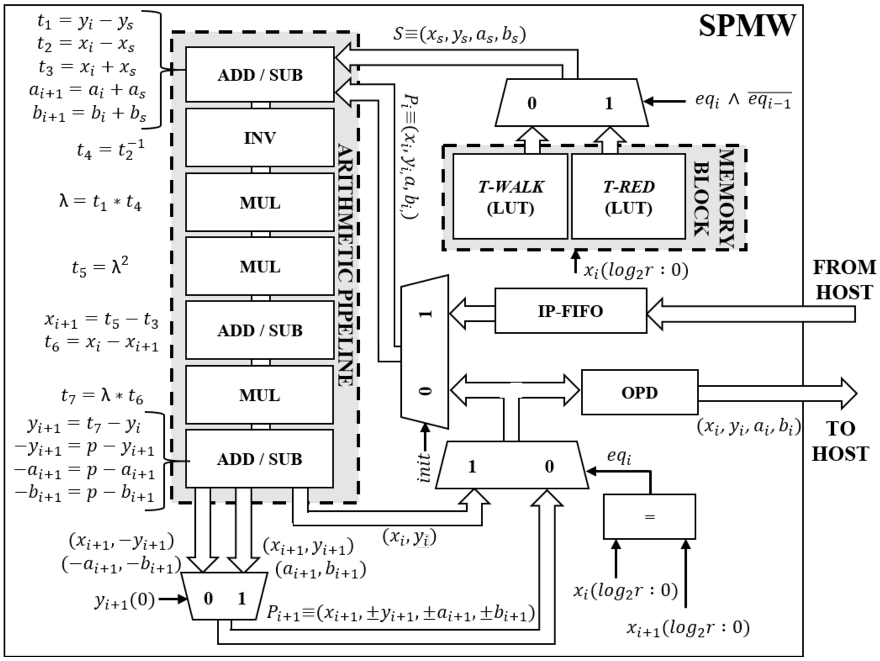
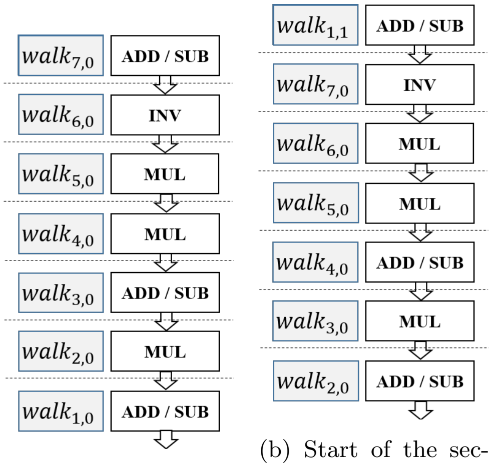


Fig. 1: High-level view of the SPMW core.

An SPMW core contains an arithmetic pipeline performing steps 1 and 2(a) from Section 2.2, an *initial points FIFO* (IP-FIFO) to hold the initial point $P_0 = (x_0, y_0)$ ($2k$ bits) and the multipliers a_0, b_0 ($2k$ bits) for each walk, two lookup tables ($4rk$ bits each), i.e., T-WALK defining the r-adding walk and T-RED for the reduction technique, three 2-to-1 multiplexers and a comparator implementing negation map and reduction (steps 2(b) and 3), and an *output point dispatcher* (OPD) for step 4. The arithmetic pipeline is composed of addition/subtraction modules, Montgomery multiplication modules [14] and an inversion module implementing a modified Kaliski inversion algorithm as in [9].

At the start-up the host loads the initial random points $P_0 = (x_0, y_0)$ and the multipliers a_0, b_0 for each walk to be started into the IP-FIFO. As mentioned above, we iteratively run two sets of walks, with only one set active at a time. Before the execution of the current set of walks is suspended because of cycle detection and escape, the host loads a fresh set of updated initial points P_0 into the IP-FIFO. A counter inside the OPD asserts the *init* signal in Figure 1 controlling the multiplexer that allows one set of walks to start and also triggers the OPD itself to send the current point of each walk in the active set to the host for cycle detection and escape.

The pipeline can be fully filled by interleaving the execution of multiple walks as shown in Figure 2, where we denote by $walk_{i,j}$ the operation performed by the i-th walk at the j-th iteration. At the beginning, $walk_{1,0}$ enters the pipeline. When the first stage completes, the output of $walk_{1,0}$



(a) Pipeline filled with 7 parallel walks.

(b) Start of the second iteration of the first walk.

Fig. 2: Single-Pipe Multi-Walks approach.

is passed to the second stage. At the same time, a new walk (i.e., $walk_{2,0}$) is started, filling the first stage. New walks can be launched until all pipeline stages are filled (Figure 2a). Once a walk completes an iteration, it re-enters the first stage to start the following iteration (e.g., $walk_{1,1}$ in Figure 2b).

As cycle detection and escape is performed periodically on the central processor (host) we use two independent sets of walks. Only one of these two sets is active and running at a time. As soon as cycle detection and escape has to be performed by sending the current points to the host, each core switches the execution to the other set of walks by simply loading updated points from the *IP-FIFO* buffer. In this way there is no performance loss due to the communication with the host for cycle detection and escape. Obviously, the time frame between consecutive cycle detection and escape runs must be large enough for the host to generate and store updated points (for the currently inactive set of walks) in the *IP-FIFO*.

The performance is limited by the different latencies of pipeline stages. A walk can move forward only when the stage having the highest latency completes its computation.

Table 1 shows the latency in terms of clock cycles of each module composing the pipeline as a function of k, assuming that the adders, Montgomery multipliers and inversion units are implemented as reported in [6] and [8].

Table 1: Latencies of the modules composing the pipeline.

Add/Sub	Montgomery multiplication	Inversion
1	k	$2k$

As shown in Table 1 the inversion module has the highest latency, i.e., $2k$. Therefore, the throughput is equal to $1/(2k + 1)$ (an additional clock cycle is required to transfer the result to the next pipeline stage). The throughput can be increased by splitting the computation of the most costly operations, namely inversion and Montgomery multiplication, across multiple pipeline stages (*pipeline unrolling*).

3.2 Pipeline unrolling

Pipeline unrolling consists in splitting the computation performed by the stages having the highest latency across multiple stages having lower latency. In our case we focus on the Montgomery multiplication module and the inversion module, with latencies equal to k and $2k$ respectively. We modify inversion and Montgomery multiplication modules so their internal state (i.e., content of their registers) can be pre-loaded (e.g., the state reached by another instance of the same module can be used as the preloaded value). With this modification a module can perform just a subset of the steps required by the entire operation and its state can be transferred to another instance of the same module. Several identical modules can be combined (in a "cascade fashion") to compute a full operation. Even though this approach implies area penalty, each module "replica" in the chain implements a new pipeline stage having lower latency and this increases the number of walks concurrently running in the pipeline.

As a first step we replicate the inversion unit to split inversion stage into two pipeline stages, each one characterized by a latency of k clock-cycles, as shown in Figure 3. The throughput becomes $TP = 1/(k+1)$, however the hardware resources needed for the inversion operation have doubled.

To further increase the throughput of the pipeline, the aforementioned approach can be recursively applied to all stages currently having maximum latency $t_{\max} = k$, namely all Montgomery multiplication and inversion stages, based on equations (2) and (3):

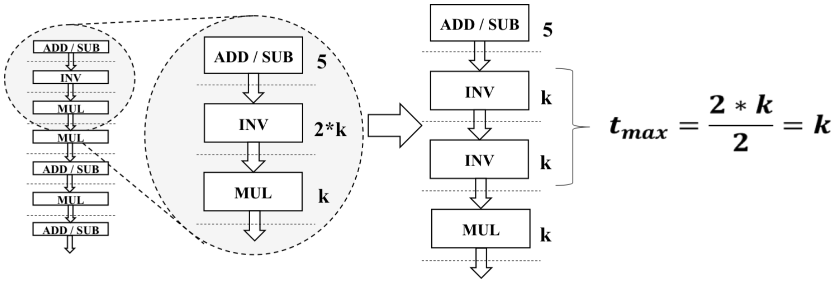


Fig. 3: Replicated inversion module (the total number of stages has increased from 7 to 8).

$$TP = \frac{1}{\lceil k/u \rceil + 1}, \tag{2}$$

$$N_s = 3 + 5 \cdot u. \tag{3}$$

Equation (2) models the SPMW core throughput with respect to k and the *unrolling factor* u. The unrolling factor denotes how many times inversion and multiplication modules are replicated, assuming as starting point the architecture depicted in Figure 3. Equation (3) computes the number of stages composing the pipeline after unrolling. The 3 addition/subtraction stages are not replicated because of their low latency, whereas the other 5 stages (i.e., 2 inversion stages and 3 multiplication stages) are replicated u times. As the value N_s equals the number of walks that can be interleaved and executed in parallel in a single pipeline, it also represents the number of points to be stored in the IP-FIFO and thus determines its size.

The unrolling factor is limited by the availability of hardware resources to accommodate the module replicas. We combine two approaches to maximize the throughput under hardware resource (area and memory) constraints:

1. Increase the unrolling factor until the area constraint is violated; this approach alone leads to a single SPMW core that, in some cases, does not utilize in the most efficient way the hardware resources available in the target device. Incrementing the unrolling factor by one causes a δ_{area} increase of the area (for instance in our case 2 inversion units and 3 multipliers must be added). This may leave hardware resources unused when the overall area is not a multiple of δ_{area}.
2. Replicate SPMW cores to build a many-core architecture, as described in subsection 3.3.

The total device area is denoted by $A_{\max}$ and the total device memory to accommodate look-up tables and the IP-FIFO is denoted by $M_{\max}$. As aforementioned, incrementing the unrolling factor by one causes a δ_{area} increase of the area. We denote by A_0 the area required to implement an SPMW core with $u = 1$. The values A_0 and δ_{area} depend both on the device technology and k. The area occupied by one SPMW core A_{SPMW} is defined by equation (4). The number of cores we can instantiate N_{SPMW} is defined by equation (5). The minimum number N_{tables} of pairs of look-up tables T-WALK and T-RED necessary to sustain the bandwidth needed by N_{SPMW} (see subsection 3.3 for the details) is defined by equation (6), while the amount of memory needed by the IP-FIFO M_{FIFO} is defined by equation (7). The maximum number $\text{NMAX}_{\text{tables}}$ of pairs of T-WALK and T-RED look-up tables that can fit the target device is defined by equation (8).

The optimal values for the unrolling factor u and the number of SPMW cores N_{SPMW}, given k, $A_{\max}$, $M_{\max}$ and the current $t_{\max}$, are found by maximizing the many-core throughput TP_{MC} defined by equation (9) under the constraints defined by equation (10). The first constraint is imposed to make sure we can accommodate enough look-up table pairs to serve all cores (see subsection 3.3 for details on how the look-up tables can be shared by multiple cores).

$$A_{\mathrm{SPMW}} = A_0 + (u-1) \cdot \delta_{\mathrm{area}}. \tag{4}$$

$$N_{\mathrm{SPMW}} = \left\lfloor \frac{A_{\max}}{A_{\mathrm{SPMW}}} \right\rfloor. \tag{5}$$

$$N_{\mathrm{tables}} = \lceil N_{\mathrm{SPMW}}/(t_{\max}+1) \rceil. \tag{6}$$

$$M_{\mathrm{FIFO}} = N_{\mathrm{SPMW}} 4kN_s. \tag{7}$$

$$\mathrm{NMAX}_{\mathrm{tables}} = \lfloor (M_{\max} - M_{\mathrm{IPFIFO}})/(8kr) \rfloor. \tag{8}$$

$$TP_{\mathrm{MC}} = N_{\mathrm{SPMW}} \cdot \frac{1}{\lceil t_{\max}/u \rceil + 1}. \tag{9}$$

$$\begin{aligned} &N_{\mathrm{SPMW}} \leq \mathrm{NMAX}_{\mathrm{tables}} \cdot (t_{\max}+1), \\ &N_{\mathrm{SPMW}} \cdot A_{\mathrm{SPMW}} \leq A_{\max}. \end{aligned} \tag{10}$$

In the following we describe the details of the inversion and Montgomery multiplication modules with state pre-loading.

Inversion module with state pre-loading The architecture of the inversion module with state pre-loading is depicted in Figure 4. We extend the input/output interface of the basic module, implementing the algorithm reported in [6], with additional input signals (i.e., $u_{\mathrm{in}}, v_{\mathrm{in}}, r_{\mathrm{in}}, s_{\mathrm{in}}$ and f_{in}) and output signals (i.e., $u_{\mathrm{out}}, v_{\mathrm{out}}, r_{\mathrm{out}}, s_{\mathrm{out}}$ and f_{out}). We add 5 multiplexers (controlled by the signal $\mathrm{sel}_{\mathrm{in}}$) to allow the internal state of the module (registers u, v, r, s and the FSM in Figure 4) to be pre-loaded from an external source through the additional input signals. The additional output signals propagate the state of the module. Several inversion modules with state pre-loading can be connected sequentially by mapping the additional output signals of one module to the additional input signals of the following one to perform a full operation. Figure 3 shows how our pipeline changes by adding one replica of the inversion module to reduce $t_{\max}$ from $2k$ to k.

The output signal r_{out} of the last module will hold the final result. Notice that several input/output signals are unused by some modules in the sequence, for instance the primary input signal a is used only by the first module. All the unused signals are automatically removed when by synthesis tools.

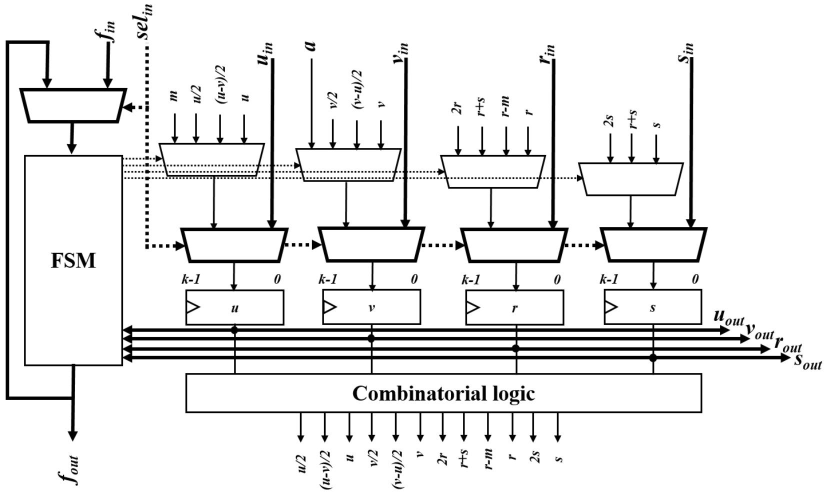


Fig. 4: Inversion module with state pre-loading. Changes to the reference architecture [8] are highlighted in bold.

Montgomery multiplier with state pre-loading The architecture of the Montgomery multiplication module with state pre-loading is depicted in Figure 5.

We follow the same strategy used above. We add output and input signals (ACC_{in} and ACC_{out}) to allow pre-loading and propagation of the state (i.e., the register ACC). Additional input and output signals ($X_{\text{in}}, Y_{\text{in}}$ and $X_{\text{out}}, Y_{\text{out}}$) are needed to pre-load and propagate the content of the registers storing X_i and Y_i. We finally add a multiplexer (controlled by the signal sel_{in}) to allow the internal state of the module (register ACC) to be pre-loaded from an external source through the additional input signals ACC_{in}.

As for the inversion module, several Montgomery multiplication units with state pre-loading can be connected sequentially to perform a full operation. The output signal P of the last module will contain the final result $P = XY2^{-k} \bmod M$. The right part of the module (see Figure 5) produces the final result P (reducing P_t modulo M). As it is used only by the last module, it can be removed from all the other replicas.

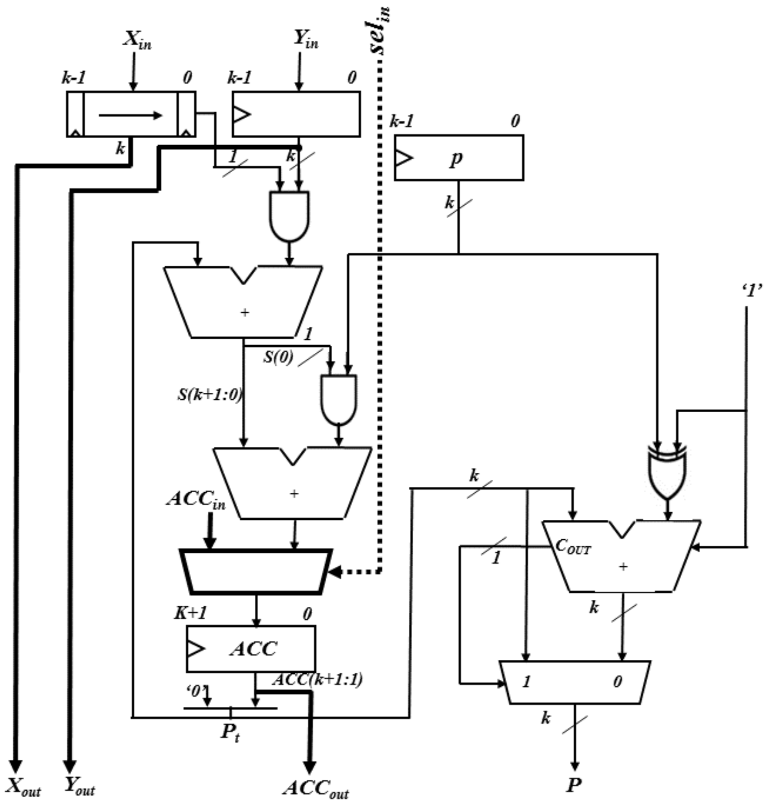


Fig. 5: Montgomery multiplier with state pre-loading. Changes to the reference architecture [6] are highlighted in bold.

3.3 System level architecture

Figure 6 shows the proposed many-core system level architecture, where the *host* communicates with an FPGA on which several instances of the SPMW core are implemented.

Each SPMW core has its own IP-FIFO, whereas the look-up tables T-WALK and T-RED can be shared by several cores as long as this is compatible with the bandwidth required by each core (as mentioned at the end of Section 3.2). More precisely, an SPMW core accesses T-WALK (or T-RED) for one cycle every $t_{\max} + 1$ cycles. Therefore, the look-up tables can be shared among $t_{\max}+1$ SPMW cores by making the execution of each core shifted by one clock cycle.

The architecture, denoted by multi-SPMW (MSPMW) in Figure 6, can be replicated if the total bandwidth needed by all cores exceeds the

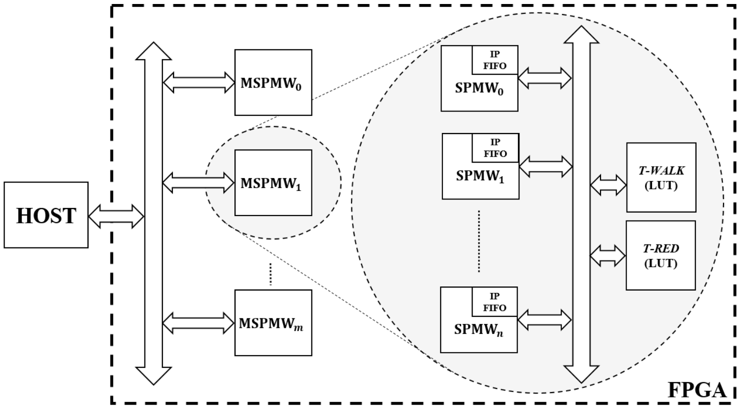


Fig. 6: System level architecture.

maximum bandwidth sustainable by the look-up tables. As will be demonstrated in Section 4, since the overall bandwidth required for the communication between the host and the FPGA is very limited, a simple interface can be employed, leading to a negligible area overhead. We analyze bandwidth requirements and other implementation details in the next section where we optimize and implement our architecture on different FPGAs.

4 Experimental results

In this section we analyze the parameters choice for our implementation and show the experimental results.

We have selected the Certicom ECCp-131 challenge as the case study. It defines an ECDLP instance on a prime order elliptic curve over a 131-bit generic prime field and it is the smallest unsolved Certicom challenge over prime fields [5]. We denote the prime order of the group of points by q.

We optimized our architecture for a *Virtex 7-xc7v2000t* FPGA [19] using the parameters reported in Table 2 and obtained $N_s = 78$ (number of stages), $N_{\text{tables}} = 2$, $N_{\text{SPMW}} = 11$ and $t_{\max} = 9$ (see Section 3.2). We have performed synthesis and place-and-route with *Xilinx ISE Design Suite 14.7*. The resulting operating frequency is $F = 192$ Mhz.

As mentioned in Section 2.2 a walk is expected to get into a fruitless 4-cycle after roughly $\alpha = 4r^3/(r-1) \approx 10.7 \cdot 10^8$ iterations. We run one set of walks for w iterations before sending the current points to the host system for cycle detection/escape and switching the execution to

Table 2: Optimization parameters for Virtex-7-xc7v2000t FPGAs. Area figures are in number of *slices*.

$\boldsymbol{k}$	$\boldsymbol{A_0}$	δ_{area}	$\boldsymbol{A_{max}}$	$\boldsymbol{M_{max}}$	$\boldsymbol{r}$	$\boldsymbol{d}$
131	3121	1561	287076 ($\approx$ 90%)	40.9Mbit 1188 BRAMs ($\approx$ 90%)	2^{14}	30

the second (suspended) set of walks (by reading updated points from the IP-FIFO). Denote by $w' \leq w$ the number of fruitless iterations a walk performs due to fruitless cycles. As in [2] we want $w'/w < 0.1$ and this results in $w = \alpha/50$ (using equation (1) in [2]).

We set $d = 30$, thus a walk is expected to hit a distinguished point every 2^{30} iterations. To apply Equation (9), we consider 90% of the available hardware resources to make sure the design will fit on the FPGA after *place and route*.

We have run *post* place-and-route simulations using *Modelsim SE 10.0c* and used *Xilinx XPower Analyzer* to estimate the power consumption, namely 26.9W.

The system generates $D = 211.2 \cdot 10^6 \cdot 2^{-30} \approx 0.2$ distinguished points per second. Each distinguished point consists of x and y coordinates and the two multipliers a and b, plus one bit to differentiate distinguished points and points sent to the host for cycle detection and escape. In total each distinguished point is represented by an h-bit string with $h = 4k + 1 = 525$. The current set of walks is suspended after $c = (wN_s(t_{\max} + 1))/F \approx 87s$ (the current points are sent to the host for cycle detection/escape and the second set of walks is re-started by reading points from the IP-FIFO) and the host has a time frame of 87 seconds to generate and store the updated points for the suspended set of walks into the IP-FIFO. A time frame of 87s is large enough to allow a regular CPU based host to serve several FPGAs. The number of IP-FIFOs equals N_{SPMW} (see Section 3.3). Each IP-FIFO contains N_s $4k$-bit points. Then the total required bandwidth is $hD + (4kN_sN_{\text{SPMW}})/c = 5.26$ Kbits/s.

Look-up tables T-WALK and T-RED and IP-FIFOs are built from 36 Kbit BRAMs configured as 512x72-bit memory blocks. To read one point (four 131-bit values) from T-WALK or T-RED in one clock cycle, each point is stored across 8 BRAMs connected in parallel, for a total of 1024 BRAMs ($N_{\text{tables}} = 2$) out of the 1292 available. The IP-FIFOs are implemented with 88 BRAMs (8 BRAMs per IP-FIFO).

Table 3 reports the overall equipment cost in dollars to solve the ECCp-131 Certicom challenge in one year on various FPGAs. The equipment cost for the Virtex UltraScale FPGA is not available yet. The Rivyera V7 is a computer hosting up to 40 Virtex-7 v2000t FPGAs [15].

We have estimated that the size of the hash table to store the distinguished points on the host should be a few TeraBytes. It can be further reduced by increasing the value of d.

Table 3: Solving ECCp-131 in one year on (a cluster of) different FPGAs. Number of points to compute: $\approx \sqrt{q\pi/4}$.

Tech	**Device**	**FPGA price**	**Points/s**	**Cost**
65 nm	Virtex-5 vlx330t	8.4 K$	20.5M	453 M$
40 nm	Virtex-6 vlx760	12.6 K$	67.3M	207 M$
28 nm	Virtex-7 v2000t	17.4 K$	211.2M	91 M$
28 nm	RIVYERA V7	500 K$	8448M	65 M$
20 nm	Virtex UltraScale 440	-	738M	-

Using the estimated power consumption of 26.9W on a Virtex-7 v2000t FPGA we can estimate the overall electricity cost in the case of the third row of Table 3, where 2610 devices are needed. Assuming that the electricity cost is 0.21$ per KWh we obtain 223K$ as the overall cost for one year, which is negligible compared to the equipment cost. It is arguably unfeasible to solve the ECCp-131 challenge on FPGAs in reasonable time as shown in Table 3, however the technology scaling could make it possible in the near future.

We have implemented our solution on a Xilinx Virtex-5 ($k = 112$) and a Xilinx Spartan-3 FPGAs ($k = 160$) to compare with the current state of the art [10] (Table 4), [9] (Table 5). We have implemented our solution using both the basic SPMW core with no unrolling and the SPMW core optimized with unrolling (SPMWopt in Tables 4 and 5).

Our solution requires more slices compared to [10]. However unlike the latter, it does not rely on DSP blocks and we achieve a speed-up of of 380%. Notice that the speed-up is computed taking into account the resources available on a single device.This is a pessimistic comparison due to fact that the prime used in [10] has a special form allowing fast reduction.

With respect to the architecture from [9], which targets generic prime fields, we achieve a speed-up of 293%.

Table 4: Comparison with [10] on a Xilinx Virtex-5 vsx240t.

	[10]	**SPMW**	**SPMWopt**
Frequency	100 Mhz	125 Mhz	125 Mhz
Points/cycle	1/114	1/225	1/14
Slices/core	5,229 (14.0%)	3,070 (8.2%)	16,386 (43.8%)
DSPs/core	130 (12.3%)	-	-
BRAMs/core	8 (1.5%)	8 (1.5%)	8 (1.5%)
BRAMs for T-WALK, T-RED	-	256 ($r = 2^{13}$) (49.6%)	256 ($r = 2^{13}$) (49.6%)
#Cores/device	6	11 ($N_s = 7$)	2 ($N_s = 48$, $t_{max} = 13$)
Prime type	special form	Any	Any
negation map	No	Yes	Yes
Years to solve secp112r1 (112-bit)	50.4	30.7	10.5
Speed-up	-	64%	380%

Table 5: Comparison with [9] on a Xilinx Spartan-3 xc3s5000.

	[9]	**SPMW**	**SPMWopt**
Frequency	40 Mhz	51 Mhz	48 Mhz
Points/cycle	1/855	1/321	1/41
Slices/core	3,230 (9.7%)	9,380 (28.2%)	29,390 (88.3%)
DSPs/core	-	-	-
BRAMs/core	15 (14.4%)	18 (17.3%)	18 (17.3%)
BRAMs for T-WALK, T-RED	-	36 ($r = 2^9$) (34.6%)	72 ($r = 2^{10}$) (69.2%)
#Cores/device	9	2 ($N_s = 7$)	1 ($N_s = 23$, $t_{max} = 40$
Prime type	Any	Any	Any
negation map	No	Yes	Yes
Years to solve ECDLP (160-bit)	$3.6 \cdot 10^{18}$	$3.6 \cdot 10^{18}$	$9.1 \cdot 10^{17}$
Speed-up	-	-	293%

5 Conclusion

We presented a many-core hardware architecture implementing the parallel version of Pollard's rho algorithm with the negation map for the ECDLP on elliptic curves defined over generic prime fields. On FPGAs our architecture outperforms the state of the art by providing a speed-up of almost 300%. The optimization methodology we presented can be applied to similar hardware designs implementing embarrassingly parallel algorithms. As a case study we estimated the monetary cost to solve the Certicom ECCp-131. In the near future we plan to explore the optimization of our architecture for specific devices like low-cost Xilinx Zynq programmable SoCs.

References

1. D. J. Bernstein, T. Lange, and P. Schwabe. On the correct use of the negation map in the Pollard rho method. In D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, editors, *Public Key Cryptography – PKC 2011*, volume 6571 of *Lecture Notes in Computer Science*, pages 128–146. Springer, Heidelberg, 2011.
2. J. W. Bos, C. Costello, and A. Miele. Elliptic and hyperelliptic curves: A practical security analysis. In *Public Key Cryptography*, pages 203–220, 2014.
3. J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow. Elliptic curve cryptography in practice. Cryptology ePrint Archive, Report 2013/734, 2013. `http://eprint.iacr.org/`.
4. J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112–bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *International Journal of Applied Cryptography*, 2(3):212–228, 2012.
5. Certicom. Certicom ECC Challenge. `https://www.certicom.com/index.php/the-certicom-ecc-challenge`, 1997.
6. J.-P. Deschamps. *Hardware Implementation of Finite-Field Arithmetic*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2009.
7. I. M. Duursma, P. Gaudry, and F. Morain. Speeding up the discrete log computation on curves with automorphisms. In K.-Y. Lam, E. Okamoto, and C. Xing, editors, *Asiacrypt 1999*, volume 1716 of *Lecture Notes in Computer Science*, pages 103–121. Springer, Heidelberg, 1999.
8. T. Güneysu. *Efficient hardware architectures for solving the discrete logarithm problem on elliptic curves*. PhD thesis, Horst Görtz Institute, Ruhr University of Bochum, 2006.
9. T. Güneysu, C. Paar, and J. Pelzl. Special-purpose hardware for solving the elliptic curve discrete logarithm problem. *ACM Trans. Reconfigurable Technol. Syst.*, (2):8:1–8:21, June.
10. L. Judge, S. Mane, and P. Schaumont. A hardware-accelerated ecdlp with high-performance modular multiplication. *International Journal of Reconfigurable Computing*, 2012:7, 2012.
11. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.
12. H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.
13. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Crypto 1985*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, Heidelberg, 1986.
14. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.
15. SciEngines. Rivyera V7-2000T. `http://www.sciengines.com/products/computers-and-clusters/v72000t.html`, 2014.
16. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-4, 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.

17. P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
18. M. J. Wiener and R. J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In S. Tavares and H. Meijer, editors, *Selected Areas in Cryptography – (SAC) 1998*, volume 1556 of *Lecture Notes in Computer Science*, pages 190–200. Springer NY, 1999.
19. Xilinx, Inc. *7 Series FPGAs Overview - DS180*, 2014.