Modular multiplication in the AMNS representation : Hardware Implementation

Louis Noyez, Nadia El Mrabet, Olivier Potin, Pascal Véron

Modular multiplication in the AMNS representation : Hardware Implementation

NOYEZ Louis[1], EL MRABET Nadia[1], POTIN Olivier[1], and VERON Pascal[2]

[1] Mines Saint-Etienne, CEA, Leti, Centre CMP, F - 13541 Gardanne France
[2] Laboratoire IMath, Université de Toulon, Toulon France

Abstract. This paper describes a hardware implementation of the modular multiplication using the Adapted Modular Number System (AMNS) representation of large integers. We propose a novel adaptation of the FIOS block Montgomery multiplication fitted to the AMNS representation. We explore multiple operations schedulings for the design of systolic architectures well suited to this FIOS algorithm. Our scalable implementation targets Ultrascale FPGA devices and takes full advantage of modern DSP48E2 Slices. We provide open-source, ready to use designs which are scalable to any width of the operands and a large range of AMNS parameters. Our designs can perform 256, 512, 1024, 2048 and 4096 bits modular multiplications in 0.178, 0.362, 0.764, 1.57 and 2.96 μs using 18, 35, 65, 125 and 245 DSP block respectively. They can allow for an improvement in computing speed and DSP AT (Digital Signal Processing block Area-Time product) of up to 17% and 13% respectively compared to state of the art implementations.

Keywords: AMNS · Montgomery multiplication · Hardware Implementation

1 Introduction

Modular multiplication modulo large numbers (notably primes) is at the core of the main public-key cryptography protocols in use today such as the RSA public-key cryptosystem [24] or the elliptic-curve based ECDSA [19] signature scheme. In order to guarantee the resilience of these protocols they must handle very large numbers (up to 4096 bits wide for RSA and 512 bits wide for ECDSA or the isogeny-based SQIsign post-quantum algorithm). Modular multiplication, which traditionally requires divisions is expensive in terms of computing time and memory/hardware resource cost. As such considerable efforts have been made to accelerate modular multiplication and increase its efficiency.

In 1985 Peter Montgomery introduced the Montgomery multiplication algorithm which performs modular multiplications by replacing trial divisions with simple binary right shifts which are easily implemented in software or hardware [21]. This algorithm is specifically suited to the binary nature of computers and is often used when multiple modular multiplications must be performed sequentially using the same modulus, as is the case in RSA (square and multiply

algorithm) or ECDSA (double and add algorithm). However very large integers cannot be directly handled by general purpose processors, which usually manipulate words of width 32 or 64 bits. Thus Koç et al. further refined Montgomery multiplication in 1996 by describing block variants of the algorithm [17]. They also provided a classification of these variants depending on the ways operands are parsed and the order in which elementary operations are performed.

Montgomery multiplication can be used regardless of the shape of the modulus, however different prime moduli can lead to different implementations of the modular multiplication. Specific classes of prime moduli leading to fast and efficient modular multiplications have been studied such as the Mersenne primes. Subsequently multiple other classes of prime moduli were explored such as the pseudo-Mersenne primes [25], the generalized Mersenne primes [26] and the more generalized Mersenne primes [9].

Instead of selecting a specific form of prime moduli, Bajard et al. overhauled classical modular multiplication entirely in 2004 by introducing the Polynomial Modular Number System (PMNS) [6]. The PMNS is an alternate polynomial representation of large integers intended to further accelerate modular multiplications by taking advantage of the parallelization of operations. It is also a redundant system where a single large integer can have multiple representatives in a PMNS. This redundancy and parallel execution has the potential to increase the security of cryptographic protocols and their resilience to hardware attacks such as side-channel attacks. The Adapted Modular Number System is a specific case of PMNS which allows for more efficient arithmetic operations. Modular multiplications in AMNS involve a polynomial multiplication step and a reduction step, the latter of which is critical for the overall performance of the process. Different methods for performing this reduction step have been explored. The so-called Montgomery-like algorithm [22] was devised for use with AMNS and further refined by Didier et al. [13]. As its name implies it follows the same principles that are behind the Montgomery multiplication algorithm. Didier et al. also devised refined generation processes for AMNS regardless of the primality of the target modulus granted it is odd. Software implementations of AMNS/PMNS have been proposed by Dosso et al. [14] and Coladon et al. [10], and a reference hardware implementation of AMNS multiplications exists in [8], which this work compares against.

Hardware acceleration of the Montgomery multiplication and its block variants has been explored thoroughly. Implementations involve trade-offs in terms of maximum operating frequency, computing time, memory/resource costs, result throughput, power consumption... [4, 16, 27, 28]. Namely some of these aspects and trade-offs were studied in [23] through systolic architectures implementing an adaptation of the FIOS block variant of Montgomery multiplication and targeting Ultrascale FPGAs by taking advantage of DSP48E2 arithmetic accelerator components capabilities.

Our main contribution consists in the development of a ready-to-use scalable design for the hardware acceleration of modular multiplications within AMNS. This implementation is based on the work from [23] on the classical represen-

tation of numbers and also targets Ultrascale FPGA devices. It is fully scalable to any width of the operands/AMNS parameters which makes it highly flexible and adaptable to multiple use cases. In order to develop this implementation we propose a novel block variant description of the Montgomery-like multiplication suited to AMNS representation. We discuss the impact of the scheduling of polynomial modular multiplications on the efficiency of our systolic architecture designed for the FIOS algorithm. Finally we compare our implementation results to hardware implementations of the Montgomery multiplication using the classical representation of numbers and the AMNS implementation of [8]. All of our work, source, verification systems and results are open and freely available at [3]. Furthermore we have developed easy to use python/sagemath tools for the generation and study of PMNS (available at [3], see Appendix A).

We analyze implementation results for cryptographic sizes (256, 512, 1024, 2048 and 4096 bits). The acceleration of modular multiplication is crucial to increasing the efficiency of cryptosystems such as Elliptic Curve Cryptography (ECDSA 256-512 bits). Legacy cryptosystems which use larger data width such as RSA (2048-4096 bits) are also likely to remain in use for the foreseeable future as institutions and companies transition to Post-Quantum Cryptography (PQC) algorithms. RSA-4096 is still widely used today by companies to secure traffic [2] and by government agencies for instance to provide root certificates [1]. Isogeny-based PQC cryptosystem like the SQIsign NIST candidate for standardisation (256-384-512 bits) [12] or the CSIDH cryptosystem (512 bits) [7] also stand to benefit from accelerating modular multiplications.

Organization of the paper: Section 1 of our paper constitutes this introduction. Section 2 introduces the AMNS representation of numbers, its operations, and the block Montgomery-like multiplication method we have devised. We describe our hardware implementation and discuss the impact of polynomial multiplication scheduling in Section 3. Section 4 highlights and compares our implementation results to state of the art hardware implementations. Finally Section 5 concludes this paper.

2 The Adapted Modular Number System

In this section, we describe the AMNS representation of numbers and its multiplication operations. We propose a novel FIOS block variant adaptation of the Montgomery-like multiplication algorithm. We do not delve into the generation of AMNS parameters and conversion operations from the classical representation of numbers to the AMNS representation which are both covered in [13]. For convenience we define the following conventions:

- Uppercase letters usually represent polynomial elements $A \in \mathbb{Z}[X]$ with $A = \sum_{i=0}^{N-1} A_i \cdot X^i$ while lowercase letters represent large integers $a \in \mathbb{Z}$
- $[p]$ represents modulo p operations while $[E]$ represents a polynomial modulo operation with reduction (see Figure 2)
- $||A||_\infty$ represents the infinite norm of polynomial A with $||A||_\infty = \max_{0 \leq i < N} |A_i|$

- $A \ggg w$ represents a w-bit signed arithmetic right shift of each coefficient of A (two's complement representations of signed integers are both shifted and sign-extended)

A PMNS $\mathcal{B} \subset \mathbb{Z}[X]$ is a subset of polynomials with integer coefficients described by the tuple (p, N, E, ρ, γ) where

- p is the prime modulus such that $\mathcal{B}$ represents $\mathbb{Z}/p\mathbb{Z}$
- N is the number of coefficients of polynomial elements of $\mathcal{B}$
- E is a monic polynomial of degree N called the external reduction polynomial
- ρ is the upper bound on the infinite norm of elements of $\mathcal{B}$
- γ is a root of E modulo p: $E(\gamma)[p] = 0$

Thus $A \in \mathcal{B} \implies \deg(A) \leq (N-1)$ and $||A||_\infty < \rho$.
$\mathcal{B}$ is said to be a PMNS if $\forall a \in \mathbb{Z}/p\mathbb{Z},\ \exists A \in \mathcal{B}$ such that $A(\gamma)[p] = a$.
Finally an AMNS is a PMNS such that $E(X) = X^N - \lambda$ where λ is a small number. In the remainder of this paper λ is taken to be 2 or a small power of 2. An example of AMNS representation with p of width 64 bits is highlighted Figure 1.

$$\bullet N = 4 \qquad \bullet E = X^4 - 2 \qquad \bullet \rho = 262144 \qquad \bullet \gamma = 13020125524669010305$$
$$p = 13157208063559315537$$
$$a = 10797837636805329088$$
$$\text{A representation of } a \text{ in } \mathcal{B} \text{ is}$$
$$A = 83086 + 7554 \cdot X + 34715 \cdot X^2 - 4780 \cdot X^3$$
$$\text{Indeed } A(\gamma)[p] = 10797837636805329088 = a \quad \text{and} \quad ||A||_\infty \leq 262144$$

Fig. 1. 64 bits example of an AMNS

2.1 AMNS multiplication

The straightforward multiplication of AMNS elements results in a polynomial element that no longer belongs to the AMNS, both because it has too high a degree and because its coefficients have too large an infinite norm as shown in Figure 2.

$$\begin{aligned}
E &= X^4 - 2 \\
A &= 83086 + \ 7554 \cdot X + 34715 \cdot X^2 - \ 4780 \cdot X^3 \\
B &= 80081 + 33377 \cdot X - \ 3680 \cdot X^2 + 25843 \cdot X^3 \\
A \cdot B = &\ 6653609966 \ + 3378093296 \cdot X \ + 2726385293 \cdot X^2 + 2895288153 \cdot X^3 \\
- &\ 92075238 \cdot X^4 + \ 914730145 \cdot X^5 - \ 123529540 \cdot X^6 \\
A \cdot B = &\ 6653609966 + 3378093296 \cdot X + 2726385293 \cdot X^2 + 2895288153 \cdot X^3 \\
&(- \ 92075238 + \ 914730145 \cdot X - \ 123529540 \cdot X^2) \cdot (\boldsymbol{E} + 2) \\
&\deg(A \cdot B) > N, \qquad ||A \cdot B||_\infty > \rho \quad \implies A \cdot B \notin \mathcal{B}
\end{aligned}$$

External Reduction:

Reduced mod $E(X) = X^N - \lambda$

$$A \cdot B[E] = \begin{pmatrix} 6653609966 \\ -2 \cdot 92075238 \end{pmatrix} + \begin{pmatrix} 3378093296 \\ +2 \cdot 914730145 \end{pmatrix} \cdot X + \begin{pmatrix} 2726385293 \\ -2 \cdot 123529540 \end{pmatrix} \cdot X^2 + 2895288153 \cdot X^3$$

$$A \cdot B[E] = 6469459490 + 5207553586 \cdot X + 2479326213 \cdot X^2 + 2895288153 \cdot X^3$$

$$\deg(A \cdot B[E]) \leq N-1, \quad ||A \cdot B[E]||_\infty > \rho \implies A \cdot B \notin \mathcal{B}$$

Internal Reduction:

$$C = \text{RedInt}(A \cdot B[E]) = 5419 + 19939 \cdot X + 12918 \cdot X^2 + 17941 \cdot X^3$$

$$\deg(C) \leq N-1, \qquad ||C||_\infty < \rho \implies C \in \mathcal{B} \quad \text{and} \quad C(\gamma)[p] = A(\gamma) \cdot B(\gamma)[p]$$

Fig. 2. AMNS multiplication

In order to derive an element that belongs to the AMNS we must first decrease the degree of the polynomial multiplication's result through an operation called the External Reduction (described Figure 3). It simply consists in performing polynomial multiplication operations modulo E. Indeed $A \cdot B = Q \cdot E + R$ where $R = A \cdot B[E]$ is the remainder of the euclidean division by E and $\deg(R) \leq N-1$ since $\deg(E) = N$. Finally $E(\gamma)[p] = 0 \implies R(\gamma)[p] = (A \cdot B)(\gamma)[p]$.

$$X^k[E] = \begin{cases} X^k & \text{if } k < N \\ \lambda \cdot X^{(k-N)} & \text{if } k \geq N \end{cases}$$

Fig. 3. External Reduction process

After the external reduction process the coefficients are still too large for the result to belong to the AMNS. In order to decrease the coefficient size an Internal Reduction process must be performed. The most common method is the Montgomery-like algorithm. It involves new parameters:

- For convenience, we set $t = (1 + |\lambda|(N-1))$ which is a multiplicative factor involved in computing the bound on the infinite norm of the result of polynomial multiplications modulo E [14].
- ϕ is a power of 2 such that $\phi > 2 \cdot t \cdot \rho$ for boundary consistency [14].
- The internal reduction polynomials M and M' such that $M(\gamma)[p] = 0$ and $M \cdot M'[E, \phi] = -1[\phi]$ with $M \in \mathcal{B}$. When the Montgomery-like algorithm is used such as described in Algorithm 1, ρ is taken such that $\rho > 2 \cdot t \cdot ||M||_\infty$.

Algorithm 1 Montgomery-like multiplication

Input: A, B, M, M', ϕ
Output: $RES \in \mathcal{B}$
$RES(\gamma)[p] = a \cdot b \cdot \phi^{-1}[p]$

1: $RES \leftarrow A \cdot B[E]$
2: $Q \leftarrow RES \cdot M'[E, \phi]$
3: $RES \leftarrow RES + Q \cdot M[E]$
4: $RES \leftarrow \dfrac{RES}{\phi}$
5: **return** RES

Similarly to the Montgomery multiplication using the classical representation of numbers, the Montgomery-like algorithm only involves multiplications and modulo/division operations by a power of two, which are easy to implement in hardware by selecting either the most or least significant bits of operands. An example of computation is given Figure 4.

• $\phi = 2^{24}$ • $M = -15681 + 51863 \cdot X + 416 \cdot X^2 - 6054 \cdot X^3$

• $E(X) = X^4 - 2$ • $M' = 5676967 + 132653 \cdot X + 15298711 \cdot X^2 + 13286439 \cdot X^3$

• $p = 13157208063559315537$ • $\gamma = 13020125524669010305$

$$
\begin{array}{rrrrr}
A = & 1\text{a}62\text{c}_h + & 1489\text{d}_h \cdot X + & 10\text{b}53_h \cdot X^2 + & \text{f}26\text{c}_h \cdot X^3 \\
B = & 22\text{de}4_h + & 148\text{e}0_h \cdot X + & 1065_h \cdot X^2 + & \text{f}41\text{e}_h \cdot X^3 \\
\text{step 1:} RES = & 89\text{bd}8547\text{a}_h + & 7075\text{db}400_h \cdot X + & 5\text{d}61\text{a}5\text{ef}8_h \cdot X^2 + & 50\text{f}5803\text{e}9_h \cdot X^3 \\
\text{step 2:} \quad Q = & \text{e}90\text{e}10_h + & 29\text{cde}0_h \cdot X + & \text{d}3\text{cbc}_h \cdot X^2 + & 32682\text{d}_h \cdot X^3 \\
\text{step 3:} RES = & 1912000000_h + & \text{b}3\text{a}7000000_h \cdot X + & 1\text{beb}000000_h \cdot X^2 - & 11\text{ca}000000_h \cdot X^3 \\
\text{step 4:} RES = & 1912_h + & \text{b}3\text{a}7_h \cdot X + & 1\text{beb}_h \cdot X^2 - & 11\text{ca}_h \cdot X^3
\end{array}
$$

Fig. 4. Montgomery-like multiplication Internal Reduction (hexadecimal notation) according to algorithm 1

2.2 Signed Montgomery-like FIOS multiplication

The initial description of the Montgomery multiplication algorithm from [21] for the classical representation of numbers was meant to handle data of arbitrary bit width. Since general purpose processors typically manipulate fewer data bits (32 or 64 bits) compared to cryptographic sizes, operands are sliced into multiple data blocks and finer descriptions of the Montgomery algorithm have been described. The seminal work on these block variants of the Montgomery multiplication is [17]. The authors propose a classification of block variants depending on the way operands are parsed, and the order in which multiplication and reduction operations are performed. The two most popular variants are the CIOS (Coarsely-Integrated Operand Scanning) and the FIOS (Finely-Integrated Operand Scanning) variants. In CIOS, one block of the first operand multiplies the full second operand before a reduction operation is performed while in FIOS, every block-by-block multiplication is followed by a reduction operation.

In this section we strive to adapt block variants of the Montgomery multiplication algorithm to the AMNS representation of numbers using "polynomial blocks" by analogy with the classical representation of numbers. We then adapt the methodology from [23] to develop a hardware accelerator for an AMNS version of the FIOS Montgomery multiplication block variant.

As far as we are aware, up until now there were no block variants of this algorithm specifically suited to the AMNS. As such software implementations of AMNS multiplications usually require polynomial elements to have coefficients which can be held within a single 32 or 64 bits wide variable (depending on the processor architecture). We introduce the following parameters:

- Let w be the word width that a processing unit can handle, and $W = 2^w$.
- Let s be the number of w-bit blocks required to hold coefficients such that $\phi = 2^{sw}$.

An additional constraint is the use of negative coefficients in the A, B, and M polynomials. To the contrary, in the classical representation the Montgomery

multiplication algorithm and its block variants only handle positive integers. Signed arithmetic is usually performed using two's complement representation of negative numbers. Performing arithmetic operations using two's complement representation requires sign extending operands to the maximum bit width of the result and keeping the least significant bits of the final result (see Figure 5). The notation ${}_{sw}\overline{a}$ denotes the two's complement representation of a over $s \cdot w$ bits.

$$ {}_{sw}\overline{a} = \begin{cases} a & \text{if } a > 0 \\ 2^{sw} - |a| & \text{if } a < 0 \end{cases} \qquad {}_{2sw}\overline{a \cdot b} = {}_{2sw}\,\overline{a} \cdot {}_{2sw}\overline{b} \quad [2^{2sw}] $$

$$ a = 7 \qquad {}_{4}\overline{a} = 0111_b \xrightarrow[\text{extension}]{\text{sign}} {}_{8}\overline{a} = 00000111_b $$

$$ b = -5 \qquad {}_{4}\overline{b} = 1011_b \xrightarrow[\text{extension}]{\text{sign}} {}_{8}\overline{b} = 11111011_b $$

$$ {}_{8}\overline{a \cdot b} = {}_{8}\,\overline{a} \cdot {}_{8}\overline{b}[2^8] = 00000111_b \cdot 11111011_b[2^8] = 0000011011011101_b[2^8] $$

$$ {}_{8}\overline{a \cdot b} = 11011101_b \implies a \cdot b = -35 $$

Fig. 5. Example of two's complement signed multiplication

In the example of Figure 5 operands a and b are first represented on 4 bits using two's complement. Their most significant bit is then replicated so as to perform a sign extension over 8 bits prior to performing a signed multiplication. Finally the multiplication is performed and the result is truncated to the least significant 8 bits to obtain the actual result of the signed multiplication.

2.3 Our new polynomial block variant of FIOS

In order to adapt the FIOS block variant of the Montgomery multiplication inspired from [23] to the AMNS representation of numbers we must fit the representation to computing hardware (as illustrated Figure 6). We have to:

- Represent coefficients with two's complement: ${}_{sw}\overline{A} = \sum_{k=0}^{N-1} {}_{sw}\overline{A_k} \cdot X^k$. For simplicity ${}_{sw}\overline{A} \iff \overline{A}$ in the remainder of this section and the signed nature of arithmetic operations is left implicit unless specified otherwise (notably in Algorithm 3). We also write $\overline{A}_k = \sum_{i=0}^{s-1} \overline{A}_{ki} \cdot (2^w)^i$ in base 2^w.
- Slice AMNS elements into "block polynomials" whose coefficients have bit-width w : $\overline{A} = \sum_{i=0}^{s-1} \overline{A}_{\bullet i} \cdot 2^{iw}$ and $||\overline{A}_{\bullet i}||_\infty < 2^w$. Thus $\overline{A}_{\bullet i} = \sum_{k=0}^{N-1} \overline{A}_{ki} X^k$.

Having to sign-extend the operands to the bit width of the result and retrieving the least significant bits of the result is a mathematical view of signed arithmetic. Computing hardware capable of handling arbitrarily large unsigned data and whose elementary operations are the polynomial multiplication and addition modulo E could directly implement a signed version of the Montgomery-like Algorithm 1 and would indeed require sign-extending operands to a bit-width of $\lceil \log_2(t \cdot \phi^2) \rceil$. Implementing these computations directly using common general

purpose processors would be very costly in terms of performance and resource. In practice processors have dedicated hardware/instructions meant to perform signed arithmetic without having to alter the representation of operands thusly.

- $p = 609751$ • $\gamma = 410669$ • $\rho = 128$ • $E = X^4 - 2$
- $N = 4$ • $w = 3$ • $s = 4$ • $\phi = 2^{12}$
- $a = 573030$ • $A = 59 - 13 \cdot X + 3 \cdot X^2 + 52 \cdot X^3$
- In Figure 6, $\overline{A} = \sum_{i=0}^{3} \overline{A}_{\bullet i} \cdot \left(2^3\right)^i$ and $\overline{A}(X) = \sum_{i=0}^{3} \overline{A}_i \cdot X^i$

Polynomials within the blue dashed boxes are the "block polynomials" $\overline{A}_{\bullet i}$.

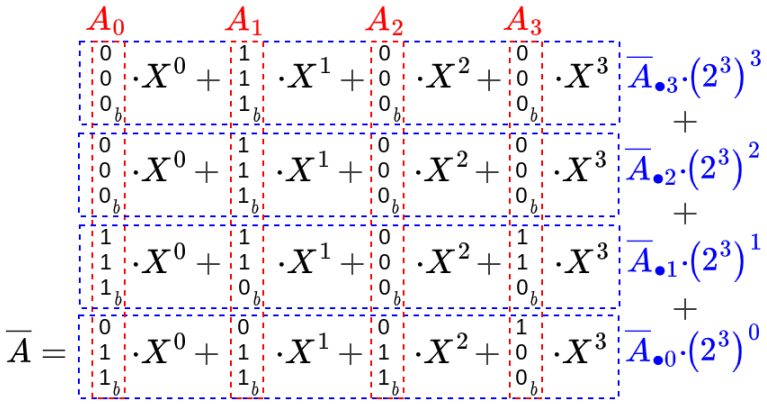

Fig. 6. Example of polynomial slicing for block Montgomery-like algorithm

Instead of directly performing polynomial multiplications modulo E we can scan block polynomials of $\overline{A}$ one by one.

Algorithm 2 $\overline{A}$**-scan Montgomery-like multiplication**

Input: $\overline{A}, \overline{B}, \overline{M}, M'_{\bullet 0}$

Output: $\overline{RES} = \sum_{j=0}^{s-1} \overline{RES}_{\bullet j} \cdot 2^{jw}$ such that $RES(\gamma)[p] \equiv a \cdot b \cdot \phi^{-1}[p]$

1: $\overline{RES} \leftarrow 0$
2: **for** $i = 0$ to $s - 1$ **do**
3: $\quad \overline{RES} \leftarrow \overline{RES} + \overline{A}_{\bullet i} \cdot \overline{B}[E]$
4: $\quad Q_i \leftarrow \overline{RES} \cdot M'_{\bullet 0}[E, W]$
5: $\quad \overline{RES} \leftarrow \overline{RES} + Q_i \cdot \overline{M}[E]$
6: $\quad \overline{RES} \leftarrow \overline{RES} \ggg w$
7: **end for**
8: **return** $\overline{RES} = \sum_{j=0}^{s-1} \overline{RES}_{\bullet j} \cdot 2^{jw}$

This leads to Algorithm 2 which is an intermediate description between a full polynomial and a completely block polynomial version of the Montgomery-like algorithm. In that description all block polynomials $\overline{A}_{\bullet i}$ are considered unsigned except for $\overline{A}_{\bullet s-1}$. Note that $RES(\gamma)[p] = a \cdot b \cdot \phi^{-1}[p]$. Indeed Montgomery multiplications are usually carried out in the Montgomery domain such that $A(\gamma)[p] = a \cdot \phi[p]$ and $RES(\gamma)[p] = a \cdot b \cdot \phi[p]$. Multiplications by M' are replaced with polynomial block multiplications by $M'_{\bullet 0}$ which simplifies operations [15].

Such an algorithm could be implemented directly using computing hardware capable of performing a polynomial multiplication between a full polynomial and a block polynomial whose coefficients are signed on $w+1$ bits.

Finally Algorithm 2 can be further refined by scanning operands $\overline{B}$ and $\overline{M}$ polynomial blocks by polynomial blocks and finely interleaving multiplication steps in a way akin to the FIOS variant of the classical Montgomery multiplication (see [17] and [23]). A functional python/sagemath implementation of Algorithm 3 is available at [3].

Algorithm 3 FIOS Montgomery-like

Input: $\overline{A}, \overline{B}, \overline{M}, M'_{\bullet 0}$

Output: $\overline{RES} = \sum_{j=0}^{s-1} \overline{RES}_{\bullet j} \cdot 2^{jw}$

such that $RES(\gamma)[p] \equiv a \cdot b \cdot \phi^{-1}[p]$

$\overline{RES} \leftarrow 0$
for $i = 0$ to $s-1$ **do**
  $\overline{RES}_{\bullet 0} \leftarrow \overline{RES}_{\bullet 0} + \overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 0}[E]$
  $Q_i \leftarrow \overline{RES}_{\bullet 0} \cdot M'_{\bullet 0}[E, W]$
  $\overline{RES}_{\bullet 0} \leftarrow \overline{RES}_0 + Q_i \cdot \overline{M}_{\bullet 0}[E]$
  $\overline{RES}_{\bullet 0} \leftarrow \overline{RES}_{\bullet 0} \ggg w$
  for $j = 1$ to $s-1$ **do**
    $\overline{RES}_{\bullet j-1} \leftarrow \overline{RES}_{\bullet j-1} + \overline{A}_{\bullet i} \cdot \overline{B}_{\bullet j}[E] + \overline{RES}_{\bullet j}$
    $\overline{RES}_{\bullet j-1} \leftarrow \overline{RES}_{\bullet j-1} + Q_i \cdot \overline{M}_{\bullet j}[E]$
    $\overline{RES}_{\bullet j} \leftarrow \overline{RES}_{\bullet j-1} \ggg w$
    $\overline{RES}_{\bullet j-1} \leftarrow \overline{RES}_{\bullet j-1}[W]$
  end for
  $\overline{RES}_{\bullet s-1} \leftarrow \overline{RES}_{\bullet s-1}[W]$
end for
return $\overline{RES} = \sum_{j=0}^{s-1} \overline{RES}_{\bullet j} \cdot 2^{jw}$

Like its classical representation counterpart, Algorithm 3 uses an outer loop to scan block polynomials of $\overline{A}$, generate the reduction parameter Q_i and reduce the least significant block. An inner loop is used to scan blocks of $\overline{B}$ and perform the remaining computation of $\overline{A}_{\bullet i} \cdot \overline{B}$ and $Q_i \cdot \overline{M}$, with multiplication and reduction steps finely interleaved one after the other. All block operands in Algorithm 3 are considered to be unsigned except during the last iteration of the inner loop and outer loop, where $\overline{A}_{\bullet s-1}$, $\overline{B}_{\bullet s-1}$ and $\overline{M}_{\bullet s-1}$ carry sign data and signed arithmetic is performed.

Algorithm 3 could be directly implemented using computing hardware capable of performing polynomial multiplications and additions modulo E between **two block polynomials** whose coefficients are signed using $w+1$ bits.

3 Hardware implementation

This section describes the target device of our hardware implementation and its arithmetic components as well as the structure of our systolic architecture. We then discuss multiple schedulings of operations for the modular polynomial multiplication operation and their impact on performance/resource cost. Finally we describe the scheduling of operations used to implement Algorithm 3.

3.1 Ultrascale FPGA and DSP48E2

FPGAs are matrices of elementary components which can be used to quickly prototype and implement a wide range of electronic circuits in conjunction with specific hardware description languages, as well as synthesis and routing development tools. In 2013, Xilinx unveiled the Ultrascale family of FPGA devices which include powerful DSP48E2 arithmetic accelerator components. We chose to develop a fine hardware description of this technology which has seldom been used to implement modular multiplication accelerators. DSP48E2 components, which are also used in [23], have capabilities suited to the implementation of block Montgomery algorithms as described Figure 7. These components feature

- A 27x18 bits asymmetric signed multiplier;
- A 3-input 48-bit adder which can be used to add the result of a multiplication, accumulate data and add external data in a single clock cycle;
- The ability to perform a native 17-bit right shift of accumulator data without requiring additional external resource or computing time;
- Cascade capabilities to quickly share data between two DSP blocks;
- Three optional levels of registers on the multiplier path (AB_{reg}, M_{reg}, P_{reg});
- An optional C_{reg} register on the external pathway to the adder.

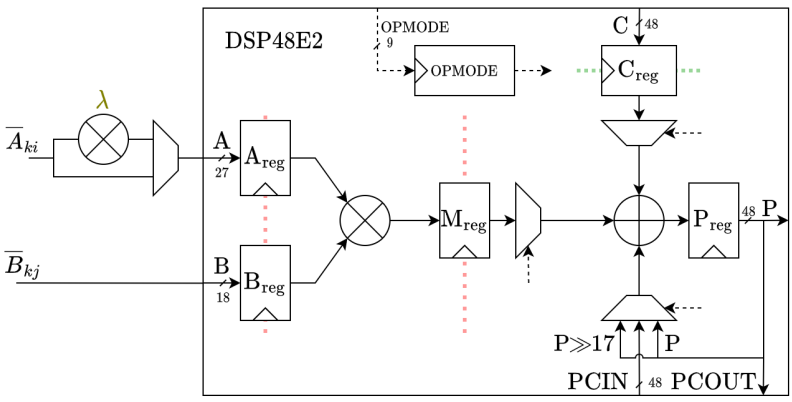

Fig. 7. Diagram of a DSP48E2 block and λ multiplication path

Like [23] we set the word width of our operand slicing to $w = 17$, namely to use the DSP native 17-bit right shift efficiently and sign-extend inputs to the DSP block when signed multiplication is required. This slicing also lets us take advantage of the asymmetric 27 bits input to the DSP multiplier to perform a λ multiplication of coefficients prior to feeding them to the DSP when necessary. This imposes a boundary condition: $\lceil \log_2(\lambda) \rceil + w + 1 \leq 27$. λ is usually be taken to be 2 or a small power of 2 for performance reasons since λ multiplication then merely requires bit-shifting the input.

Using these components we designed a systolic architecture to implement Algorithm 3. A systolic architecture is a matrix of Processing Elements (PEs) which operate in lockstep and can communicate data with one another. The structure of our systolic array is illustrated Figure 8. In [23], the authors use a linear systolic array where processing elements start their operations one after the other in a single column. Contrarily, it seems natural for an AMNS implementation to use N columns of processing elements in a 2D systolic array, each column dedicated to the computation of a single coefficient of the result. These columns can process data in parallel (see scheduling Subsection 3.2). Each line of processing elements computes operations related to a single iteration of the outer loop of Algorithm 3 and can start their subroutine as soon as they are provided data by the previous line of processing elements.

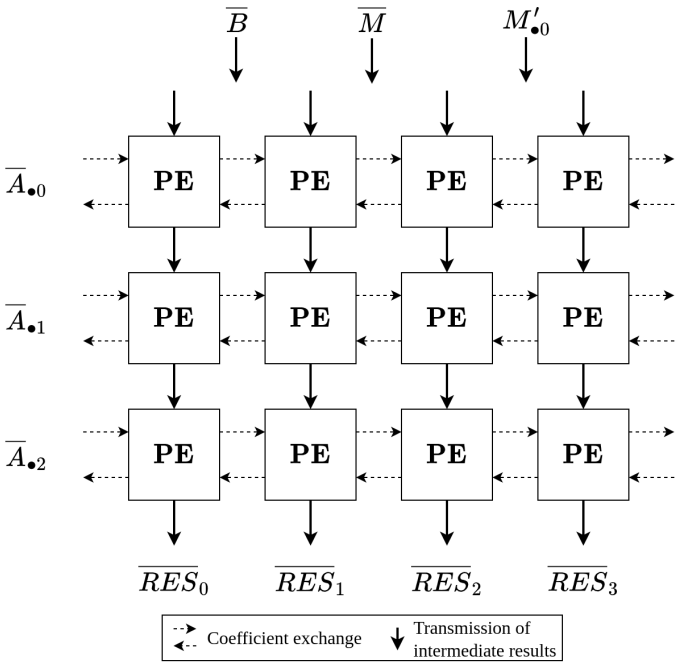

Fig. 8. Illustration of our systolic architecture ($N = 4, s = 3$)

In Figure 8, coefficients required for the computations of a single line of PEs are exchanged between PEs on the same line. Data required for a line of PEs to proceed with its computations is provided by the previous line of PEs.

The configuration of DSP48E2 blocks in processing elements is fixed. We do not use the cascade capabilities of DSP blocks, which could lead to congestion and decreased maximum operating frequency as our processing elements are more complex and numerous than those used in [23]. This also promotes scalability as it relaxes constraints on our synthesizer and routing tool, which are no longer required to use DSP blocks that are directly next to one another within the FPGA. We chose to use the AB_{reg}, P_{reg} and C_{reg} registers, which provide a good compromise between the maximum operating frequency of our circuit and the total number of clock cycles required for computation.

3.2 Scheduling of polynomial modular multiplication

There are multiple ways of scheduling a polynomial modular multiplication, which have different impacts on the performance and resource cost of the implementation. In this section we explore two extreme cases of scheduling and their consequences given N parallel processing elements which can perform a signed multiplication of two w-bit coefficients (potentially a λ multiplication) and accumulation of the results. For the remainder of this section, $\tilde{A}$ and $\tilde{B}$ represent polynomials whose coefficients have width w bits and which can be handled by a single line of processing elements.

Target	Cycle	Coefficients				
		$\boldsymbol{X^0}$	$\boldsymbol{X^1}$	$\boldsymbol{X^2}$	$\boldsymbol{X^3}$	$\boldsymbol{X^4}$
$\tilde{A}$		$\tilde{A}_{\mathbf{0}}$	$\tilde{A}_{\mathbf{1}}$	$\tilde{A}_{\mathbf{2}}$	$\tilde{A}_{\mathbf{3}}$	$\tilde{A}_{\mathbf{4}}$
$\tilde{B}$		$\tilde{B}_{\mathbf{0}}$	$\tilde{B}_{\mathbf{1}}$	$\tilde{B}_{\mathbf{2}}$	$\tilde{B}_{\mathbf{3}}$	$\tilde{B}_{\mathbf{4}}$
$\tilde{A} \cdot \tilde{B}[E]$	1	$\tilde{A}_{\mathbf{0}}\tilde{B}_{\mathbf{0}}$	$\tilde{A}_{\mathbf{0}}\tilde{B}_{\mathbf{1}}$	$\tilde{A}_{\mathbf{0}}\tilde{B}_{\mathbf{2}}$	$\tilde{A}_{\mathbf{0}}\tilde{B}_{\mathbf{3}}$	$\tilde{A}_{\mathbf{0}}\tilde{B}_{\mathbf{4}}$
	2	$+\lambda\tilde{A}_{\mathbf{1}}\tilde{B}_{\mathbf{4}}$	$+\tilde{A}_{\mathbf{1}}\tilde{B}_{\mathbf{0}}$	$+\tilde{A}_{\mathbf{1}}\tilde{B}_{\mathbf{1}}$	$+\tilde{A}_{\mathbf{1}}\tilde{B}_{\mathbf{2}}$	$+\tilde{A}_{\mathbf{1}}\tilde{B}_{\mathbf{3}}$
	3	$+\lambda\tilde{A}_{\mathbf{2}}\tilde{B}_{\mathbf{3}}$	$+\lambda\tilde{A}_{\mathbf{2}}\tilde{B}_{\mathbf{4}}$	$+\tilde{A}_{\mathbf{2}}\tilde{B}_{\mathbf{0}}$	$+\tilde{A}_{\mathbf{2}}\tilde{B}_{\mathbf{1}}$	$+\tilde{A}_{\mathbf{2}}\tilde{B}_{\mathbf{2}}$
	4	$+\lambda\tilde{A}_{\mathbf{3}}\tilde{B}_{\mathbf{2}}$	$+\lambda\tilde{A}_{\mathbf{3}}\tilde{B}_{\mathbf{3}}$	$+\lambda\tilde{A}_{\mathbf{3}}\tilde{B}_{\mathbf{4}}$	$+\tilde{A}_{\mathbf{3}}\tilde{B}_{\mathbf{0}}$	$+\tilde{A}_{\mathbf{3}}\tilde{B}_{\mathbf{1}}$
	5	$+\lambda\tilde{A}_{\mathbf{4}}\tilde{B}_{\mathbf{1}}$	$+\lambda\tilde{A}_{\mathbf{4}}\tilde{B}_{\mathbf{2}}$	$+\lambda\tilde{A}_{\mathbf{4}}\tilde{B}_{\mathbf{3}}$	$+\lambda\tilde{A}_{\mathbf{4}}\tilde{B}_{\mathbf{4}}$	$+\tilde{A}_{\mathbf{4}}\tilde{B}_{\mathbf{0}}$

Table 1. Congested Scheduling of polynomial modular multiplication ($N = 5$)

The first scheduling we suggest is highlighted in Table 1. Each computing step corresponds to a different clock cycle. In this scheduling a single coefficient of $\tilde{A}$ is used by all processing elements during each clock cycle, while a different coefficient of $\tilde{B}$ is used by each processing element at any given time. Subsequent computations involve feeding the next coefficient of $\tilde{A}$ to all processing elements, and rotating coefficients of $\tilde{B}$ between processing elements on the

same line. The issue with this scheduling is that if coefficients of $\tilde{A}$ are stored in a single source register, feeding a single coefficient to all PEs will generate a high fanout/density of signals especially as N increases which is detrimental to scalability. Consequently high congestion will decrease the maximum frequency of our implementation and increase resource cost as our development tools struggle to route all signals as is illustrated Figure 9.

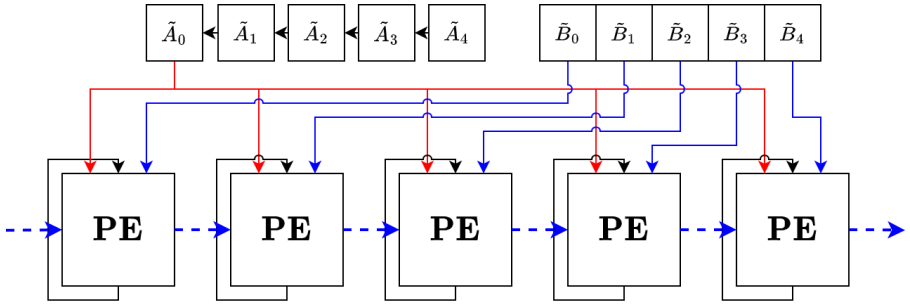

Fig. 9. Congested PE line

A possible solution to this issue is to provide N copies of $\tilde{A}$ in multiple registers. However this would cost a lot of hardware resource and additional clock cycles to fill up these registers. To remedy these issues, we propose a relaxed scheduling (see Table 2).

Target	Cycle	Coefficients				
		$\boldsymbol{X^0}$	$\boldsymbol{X^1}$	$\boldsymbol{X^2}$	$\boldsymbol{X^3}$	$\boldsymbol{X^4}$
$\tilde{A} \cdot \tilde{B}[E]$	1	$\tilde{A}_0\tilde{B}_0$	$\lambda\tilde{A}_3\tilde{B}_3$	$\tilde{A}_1\tilde{B}_1$	$\lambda\tilde{A}_4\tilde{B}_4$	$\tilde{A}_2\tilde{B}_2$
	2	$+\lambda\tilde{A}_2\tilde{B}_3$	$+\tilde{A}_0\tilde{B}_1$	$+\lambda\tilde{A}_3\tilde{B}_4$	$+\tilde{A}_1\tilde{B}_2$	$+\tilde{A}_4\tilde{B}_0$
	3	$+\lambda\tilde{A}_4\tilde{B}_1$	$+\lambda\tilde{A}_2\tilde{B}_4$	$+\tilde{A}_0\tilde{B}_2$	$+\tilde{A}_3\tilde{B}_0$	$+\tilde{A}_1\tilde{B}_3$
	4	$+\lambda\tilde{A}_1\tilde{B}_4$	$+\lambda\tilde{A}_4\tilde{B}_2$	$+\tilde{A}_2\tilde{B}_0$	$+\tilde{A}_0\tilde{B}_3$	$+\tilde{A}_3\tilde{B}_1$
	5	$\lambda\tilde{A}_3\tilde{B}_2$	$+\tilde{A}_1\tilde{B}_0$	$+\lambda\tilde{A}_4\tilde{B}_3$	$+\tilde{A}_2\tilde{B}_1$	$+\tilde{A}_0\tilde{B}_4$

Table 2. Relaxed Scheduling of polynomial modular multiplication ($N = 5$)

In this scheduling, the congestion issue is avoided as is shown Figure 10 by making sure that no coefficient of A or B are used more than once by PEs at any given time. Subsequent computations involve a right rotation of coefficients of A and a left rotation of coefficients of B between PEs. The fact that these transformations are of the same nature also simplifies the circuit compared to the congested scheduling. Although the pattern of λ multiplications and the routing of signals might seem more complex, synthesizer and routing tools can accom-

modate these changes. The drawback of this scheduling is that it is only possible for N an odd number of coefficients contrary to the congested scheduling, which is valid regardless of the value of N. The relevance of this choice is explored in Subsection 4.1.

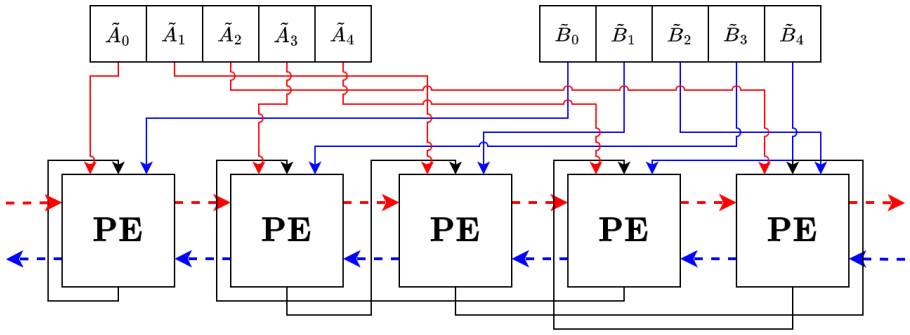

Fig. 10. Relaxed PE line

3.3 FIOS Montgomery-like scheduling

The implementation of the FIOS Montgomery-like algorithm using our systolic architecture is fairly straightforward.

start cycle	end cycle	target variable	operation
1	1		$\text{LOAD}(\text{AB}_{\text{reg}})$
2	$N+1$	$\overline{RES}_{\bullet 0}$	$\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 0}[E] + \overline{RES}_{\bullet 0}$
$N+2$	$N+2$	TEMP	$\text{FIRST}(\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 1}[E])$
$N+3$	$2N+2$	Q_i	$\overline{RES}_{\bullet 0} \cdot M'_{\bullet 0}[E, W]$
$2N+3$	$2N+3$	TEMP	$\text{SECOND}(\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 1}[E]) + \text{TEMP}$
$2N+4$	$3N+3$	$\overline{RES}_{\bullet 0}$	$Q_i \cdot \overline{M}_{\bullet 0}[E] + \overline{RES}_{\bullet 0}$
$3N+4$	$4N+1$	$\overline{RES}_{\bullet 0}$	$\text{REMAINING}(\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 1}[E]) + \text{TEMP}$ $+\overline{RES}_{\bullet 0} \ggg w$
$4N+2$	$5N+1$	$\overline{RES}_{\bullet 0}$	$Q_i \cdot \overline{M}_{\bullet 1}[E] + \overline{RES}_{\bullet 0} + \overline{RES}_{\bullet 1}$
$5N+2$	$6N+1$	$\overline{RES}_{\bullet 1}$	$\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 2}[E] + \overline{RES}_{\bullet 0} \ggg w$
$6N+2$	$7N+1$	$\overline{RES}_{\bullet 1}$	$Q_i \cdot \overline{M}_{\bullet 2}[E] + \overline{RES}_{\bullet 1} + \overline{RES}_{\bullet 2}$
$7N+2$	$8N+1$	$\overline{RES}_{\bullet 2}$	$\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 3}[E] + \overline{RES}_{\bullet 1} \ggg w$
$8N+2$	$9N+1$	$\overline{RES}_{\bullet 2}$	$Q_i \cdot \overline{M}_{\bullet 3}[E] + \overline{RES}_{\bullet 2} + \overline{RES}_{\bullet 3}$
		$\vdots$	
$(2s-1)N+2$	$2sN+1$	$\overline{RES}_{\bullet s-2}$	$\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet s-1}[E] + \overline{RES}_{\bullet s-3} \ggg w$
$2sN+2$	$2sN+N+1$	$\overline{RES}_{\bullet s-2}$	$Q_i \cdot \overline{M}_{\bullet s-1}[E] + \overline{RES}_{\bullet s-2} + \overline{RES}_{\bullet s-1}$
$2sN+N+2$	$2sN+N+2$	$\overline{RES}_{\bullet s-1}$	$\overline{RES}_{\bullet s-2} \ggg w$

Table 3. Scheduling of the FIOS Montgomery-like algorithm for the i^{th} line of PEs

Table 3 describes the scheduling of operations for iteration i of the outer loop of Algorithm 3 handled by the i^{th} line of processing elements. Initial values of $\overline{RES}_{\bullet j}$ are assumed to be provided by the previous $(i-1)^{\text{th}}$ line of processing elements. Polynomial multiplications modulo E are performed according to the chosen scheduling as described in Subsection 3.2. The three input adder of DSP block is used every time two additions must be performed during the same clock cycle, and the native 17 bits right shift of data is taken advantage when $\overline{RES}_{\bullet j-1} \ggg w$ is computed.

The first operation performed consists in the initial loading of data into the AB_{reg} registers of the DSP block. Subsequently $\overline{RES}_{\bullet 0}$ is computed as described by line 3 of Algorithm 3. Data from $\overline{RES}_{\bullet 0}$ must be reused by the DSP block to compute the Q_i parameter but it is not immediately available since it must pass through the AB_{reg} registers. Thus in order not to waste computation cycles we start computing the FIRST operation of $\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 1}[E]$. The same is true while the Q_i parameter is being computed and the SECOND operation of $\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 1}[E]$ is

performed. The final value of $\overline{RES}_{\bullet 0}$ used in the i^{th} iteration of the outer loop is then calculated (as well as the REMAINING operations of $\overline{A}_{\bullet i} \cdot \overline{B}_{\bullet 1}[E]$). The next, $(i+1)^{\text{th}}$ line of processing elements can start its operation $4N+2$ clock cycles after the start of the i^{th} line of PEs as the final value of $\overline{RES}_{\bullet 0}$ will be available just in time to be used by the $(i+1)^{\text{th}}$ PE line. Further scheduling assumes a regular behavior and performs operations described by the inner loop of Algorithm 3 until the end of an FIOS Montgomery-like outer loop iteration. Each PE line of PEs requires $2 \cdot s \cdot N + N + 2$ clock cycles to perform an outer loop iteration. Since s PE lines are used in the systolic architecture and PE lines start operations every $4N+2$ clock cycles, the complete FIOS Montgomery-like algorithm is performed in a total of $(4 \cdot N + 2) \cdot (s-1) + 2 \cdot s \cdot N + N + 2$ clock cycles.

4 Implementation results

In this section we first study the relevance of our choice of scheduling for the polynomial multiplication modulo E and its theoretical impact on performance, DSP resource cost, and DSP-wise efficiency. We then observe its influence on the actual scalability of our implementation. Finally we compare our implementation results with results from the state of the art. λ is set to 2.

4.1 Choice of polynomial modular multiplication scheduling

Although we have chosen the relaxed scheduling of polynomial multiplication modulo E as described in Subsection 3.2, it constrains the use of an odd number of coefficients N for the representation of AMNS elements in contrast with the congested scheduling, which allows the use of any value for N. We can evaluate the relevance of this choice for cryptographic sizes of p given that the total number of clock cycles and processing elements required is independent of the choice of polynomial modular multiplication scheduling. We first approximate s the number of polynomial blocks/PE lines as a function of N.

- Let $t = (1 + |\lambda| \cdot (N-1))$ and $w = 17$.
- Let width $= \lceil \log_2(p) \rceil$.
- Let $||M||_\infty \approx p^{\frac{1}{N}}$ by virtue of the generation process of AMNS (not detailled here, see [13]). Thus $\lceil \log_2(||M||_\infty) \rceil \approx \left\lceil \frac{\text{width}}{N} \right\rceil$.
- Let $\rho(N)$ be a power of two such that $\rho(N) > 2 \cdot t \cdot ||M||_\infty$ namely $\log_2(\rho(N)) = \left\lceil \log_2(2 \cdot t) + \frac{\text{width}}{N} \right\rceil$.
  $\rho(N)$ is taken to be a power of two to simplify conversions from the classical representation to the AMNS representation.
- Let $s(N) = \left\lceil \frac{\log_2(2 \cdot t) + \log_2(\rho(N)) + 1}{w} \right\rceil$. Note that $\phi = 2^{sw}$.

Figures 11, 12 and 13 from Annex B present respectively the total number of clock cycles, of processing elements, and the PEs/clock cycles product as a measure of efficiency (the lower the better) for cryptographic sizes of p ranging from 256 to 4096 bits and N ranging from 3 to 20. Increasing N beyond 20 has no further impact on s and thus decreases performance/increases resource cost. For each of these metrics the best theoretical N is annotated. When the best N is even, the next best odd N is also annotated. Regardless of the metric, the difference between the best even N and best odd N is less than 0.6% of the best solution in terms of metric. Furthermore using the best solution with N odd usually results in a smaller N, which tends to provide better maximum operating frequency in implementations. Additionally the relaxed scheduling corresponds to the least congested system possible for our implementation and will thus tend to provide better maximum frequency as highlighted in Table 4. This study leads us to believe that the constraint of using N odd has a negligible negative impact compared with its benefits.

Scheduling	N	s	Frequency (MHz)
Congested	5	4	625
Congested	11	2	550
Relaxed	5	4	625
Relaxed	11	2	600

Table 4. Illustration of scalability depending on scheduling

Table 4 illustrates the difference in scalability between implementations using either the congested or relaxed scheduling of polynomial modular multiplication and justifies our choice of the relaxed scheduling. Indeed as N increases the congested scheduling incurs a sharper drop in the maximum frequency of the system while the relaxed scheduling remains stable at around 600 MHz for the same values of s.

4.2 State of the Art comparison

In Table 5 we compare the implementation results of our AMNS design with the equivalent CA0D2C1E design from [23] which uses the classical representation of numbers for multiple cryptographic sizes. We also compare against the work from [8], which implements AMNS with a fixed number of 4 coefficients for multiple cryptographic sizes. Area-Time product (AT) is computed for the three main types of resource used (DSP/LUT/FF) and is meant to be a measure of efficiency (the lower the better). While implementations from [8] provide the fastest performance owing to their low number of clock cycles, they require a very large number of DSP blocks (up to 6.7 times more than our most conservative implementation) which leads to a poor DSP AT. Although they are still

Design parameters	Freq (MHz)	Latency (cc)	DSP/LUT/FF	Time (μs)	DSP/LUT/FF AT (resource.μs)
width = 256					
CA0D2C1E [23]	625	140	16/1759/3365	0.224	3.58/394/754
[8]	200	33	120/2728/-	0.165	19.8/450/-
[8]	194	47	91/1718/-	0.242	22.0/415/-
[29]	-	-	289/87622/2952	0.01755	5.072/1537/51
[5]	-	-	248/9450/-	0.0852	21.13/805/-
[18]	86.79	29	120/6688/5163	0.334	40.1/2234/1724
[20]	68	-	0/187900/-	0.0147	0/2762/-
$N = 3$, $s = 6$	**625**	**111**	**18/4156/5145**	**0.178**	**3.20/738/914**
$N = 5$, $s = 4$	625	113	20/4824/5541	0.181	3.62/872/1000
$N = 7$, $s = 3$	625	111	21/5109/5637	0.178	3.73/907/1000
$N = 11$, $s = 2$	600	103	22/5421/5528	0.172	3.78/931/949
width = 512					
CA0D2C1E [23]	625	275	31/3443/6602	0.440	13.6/1510/2900
[8]	162	33	188/29985/-	0.204	38.4/6120/-
[8]	182	47	176/37138/-	0.258	45.4/9580/-
[29]	-	-	1089/329868/4143	0.02568	27.97/8471/106
[18]	61.3	29	560/23511/11307	0.472	264.3/11097/5336
$N = 5$, $s = 7$	525	209	35/8044/10510	0.398	13.9/3200/4180
$N = 7$, $s = 5$	**550**	**199**	**35/8124/10128**	**0.362**	**12.7/2940/3660**
$N = 11$, $s = 4$	525	239	44/10498/12350	0.455	20.0/4780/5620
$N = 13$, $s = 3$	500	201	39/9268/11033	0.365	14.3/3390/4030
width = 1024					
CA0D2C1E [23]	625	545	61/6785/12108	0.872	53.2/5920/10600
[11]	244	1372	9/5003/-	5.63	50.7/28166/-
$N = 5$, $s = 13$	**525**	**401**	**65/14999/19272**	**0.764**	**49.6/11500/14700**
$N = 11$, $s = 7$	500	443	77/18282/21598	0.886	68.2/16200/19100
width = 2048					
CA0D2C1E [23]	625	1085	121/13487/22602	1.74	210/23400/39000
[11]	246	2945	9/5964/-	11.99	107.9/71508/-
$N = 5$, $s = 25$	**500**	**785**	**125/29182/35008**	**1.57**	**196/45800/54900**
width = 4096					
CA0D2C1E [23]	625	2174	242/26978/44806	3.48	842/93800/156000
[11]	175	2945	27/11898/-	16.82	454.1/200124/-
$N = 5$, $s = 25$	**525**	**1553**	**245/58161/66740**	**2.96**	**725/172000/197000**

Table 5. Comparison of implementation results

competitively efficient for LUT AT when width = 256, their LUT cost explodes for width = 512, leading to poor LUT AT. They also have a fairly low maximum frequency compared to our implementations, which may be due to congestion issues and the fact they are implemented on older Xilinx Artix 7 technology compared to our use of Ultrascale devices.

When comparing with the implementation of [23], our most conservative designs require slightly more DSP blocks (between 2 or 4 additional blocks). However they also require up to 20% fewer clock cycles (up to 27% in the case of 2048 bits), leading to faster computations (up to 17% in the case of 512 bits). This allows us to propose designs with a comparable or better DSP AT (up to 13% in the case of 4096 bits). Our implementations require twice as many LUTs and FFs, leading to poorer LUT and FF AT.

We also compare against other methods based on the classical representation of numbers. [29] also uses an Ultrascale+ target device for data widths 256 and 512 bits. Despite its implementations being 10 times faster than ours, it uses more LUTs and up to 31 times more DSP blocks, hence a worse DSP and LUT AT. It does use fewer FFs, likely due to registers within DSP blocks being used instead. Likewise the 256 bits implementation from [5] which exploits the RNS representation of numbers on an Ultrascale+ target uses vastly more DSP blocks than ours. Although it is faster than our implementation, it is still slower compared to [29] which leads it to have a worse LUT AT and a 7 times higher DSP AT. Implementations using the Karatsuba method for integer multiplication have also been presented in [18] and [20]. Interestingly the 256 bits implementation from [20] uses no DSP blocks in exchange for an extremely high number of LUTs which causes it to have a worse LUT AT despite it being the fastest 256 bits implementation. Implementations from [18] have worse DSP, LUT and FF AT than ours because of their higher resource cost and slower execution speed. This might not be indicative of the performance or efficiency of Karatsuba based Montgomery multiplication implementations since methods such as Karatsuba typically target data with larger bit-width. For instance the FFT-based implementations from [11], although slower and having a worse LUT AT, uses much fewer resource than ours. Thus it has a comparable DSP efficiency to ours for width 1024 bits but much better efficiency for widths 2048 and 4096 bits where it boasts almost half as much DSP AT than we do.

Actual implementation results of our designs confirm the predictions from Figure 11 as far as performance is concerned but do not quite follow the DSP AT predictions from Figure 13 because of the slight decrease in maximum operating frequency incurred by the increase in number of coefficients N and number of polynomial blocks s.

5 Conclusion and perspectives

This work has presented a complete FPGA hardware implementation of the modular multiplication using the AMNS representation of large numbers. It is meant to serve as a building block for the acceleration of cryptography protocols involving modular multiplications such as RSA, ECDSA or the isogeny-based post-quantum algorithm SQIsign. We have introduced and developed a novel polynomial block description of the Montgomery-like algorithm used in AMNS multiplication. Our methodology can be used to adapt any block variant of the Montgomery multiplication algorithm previously intended for the classical representation of numbers to the AMNS representation. We have subsequently implemented a hardware accelerator of the FIOS Montgomery-like algorithm through a 2D systolic architecture. This architecture takes full advantage of the features available in the modern DSP48E2 arithmetic components of Ultrascale FPGAs. Our design is highly flexible and adaptable to any number of coefficients, size of coefficients and other AMNS parameters which makes it versatile and suited to a number of different applications. We have studied multiple different scheduling of operations for the polynomial modular multiplication and explored their impact on the performance, resource cost and efficiency of the system. We also demonstrated that the constraint we imposed of using an odd number of coefficients N had negligible impact on the performance and resource cost of our AMNS designs. In order to validate our design we have developed software python/sagemath tools which can be used to effortlessly explore and manipulate AMNS representations. All of our work, source, verification utilities and results are open and available at [3] as a ready-to-use design. Our AMNS design are competitive in terms of performance, DSP cost and DSP efficiency. They can perform 256, 512, 1024, 2048 and 4096 bits modular multiplications in 0.178, 0.362, 0.764, 1.57 and 2.96 μs using 18, 35, 65, 125 and 245 DSP block respectively. They allow for an improvement in computing speed and DSP AT of up to 17% and 13% respectively compared to [23].

Perspectives for this work include further software implementation of our novel FIOS Montgomery-like algorithm as well as the study of other block variants of the Montgomery multiplication applied to AMNS. Indeed besides hardware acceleration the FIOS Montgomery-like algorithm is also intended to accelerate software implementations using general purpose processors. Finally we intend to explore the resilience of hardware implementations based on AMNS against Side-Channel attacks.

References

1. https://cyber.gouv.fr/igca (2022)
2. https://community.teamviewer.com/French/discussion/105303/amelioration-de-la-securite-cle-rsa-passe-de-2048-bits-a-4096-bits (2023)
3. FIOS DSP Montgomery Multiplier. https://github.com/LOUISNOYEZ/AMNS_MM (2024)
4. Abd-Elkader, A.A.H., Rashdan, M., Hasaneen, E.S.A.M., Hamed, H.F.A.: Fpga-based optimized design of montgomery modular multiplier. IEEE Transactions on Circuits and Systems II: Express Briefs **68**(6), 2137–2141 (2021). https://doi.org/10.1109/TCSII.2020.3040665
5. Ahsan, J., Esmaeildoust, M., Kaabi, A., Zarei, V.: Efficient fpga implementation of rns montgomery multiplication using balanced rns bases. Integration **84**, 72–83 (2022). https://doi.org/https://doi.org/10.1016/j.vlsi.2021.12.006, https://www.sciencedirect.com/science/article/pii/S0167926021001322
6. Bajard, J.C., Imbert, L., Plantard, T.: Modular number systems: Beyond the mersenne family. In: Handschuh, H., Hasan, M.A. (eds.) Selected Areas in Cryptography, vol. 3357, pp. 159–169. Springer Berlin Heidelberg (2004), http://link.springer.com/10.1007/978-3-540-30564-4_11, series Title: Lecture Notes in Computer Science
7. Castryck, W., Lange, T., Martindale, C., Panny, L., Renes, J.: Csidh: an efficient post-quantum commutative group action. In: Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part III 24. pp. 395–427. Springer (2018)
8. Chaouch, A., Didier, L.S., Dosso, F.Y., El Mrabet, N., Bouallegue, B., Ouni, B.: Two hardware implementations for modular multiplication in the amns: Sequential and semi-parallel. Journal of Information Security and Applications **58**, 102770 (2021), https://www.sciencedirect.com/science/article/pii/S2214212621000193
9. Chung, J., Hasan, A.: More generalized mersenne numbers. In: Matsui, M., Zuccherato, R.J. (eds.) Selected Areas in Cryptography, vol. 3006, pp. 335–347. Springer Berlin Heidelberg (2004), http://link.springer.com/10.1007/978-3-540-24654-1_24, series Title: Lecture Notes in Computer Science
10. Coladon, T., Elbaz-Vincent, P., Hugounenq, C.: Mphell: A fast and robust library with unified and versatile arithmetics for elliptic curves cryptography. In: 2021 IEEE 28th Symposium on Computer Arithmetic (ARITH). pp. 78–85 (2021). https://doi.org/10.1109/ARITH51176.2021.00026
11. Dai, W., Chen, D.D., Cheung, R.C.C., Koç, .: Area-time efficient architecture of fft-based montgomery multiplication. IEEE Transactions on Computers **66**(3), 375–388 (2017). https://doi.org/10.1109/TC.2016.2601334
12. De Feo, L., Kohel, D., Leroux, A., Petit, C., Wesolowski, B.: Sqisign: compact post-quantum signatures from quaternions and isogenies. In: Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part I 26. pp. 64–93. Springer (2020)
13. Didier, L.S., Dosso, F.Y., Véron, P.: Efficient modular operations using the adapted modular number system **10**(2), 111–133 (2020), http://link.springer.com/10.1007/s13389-019-00221-7, journal of Cryptographic Engineering

14. Dosso, F.Y., Robert, J.M., Véron, P.: Pmns for efficient arithmetic and small memory cost. IEEE Transactions on Emerging Topics in Computing **10**(3), 1263–1277 (2022). https://doi.org/10.1109/TETC.2022.3187786
15. Dussé, S.R., Kaliski, B.S.: A cryptographic library for the motorola dsp56000. In: Damgård, I.B. (ed.) Advances in Cryptology — EUROCRYPT '90. pp. 230–244. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
16. Huang, M., Gaj, K., El-Ghazawi, T.: New hardware architectures for montgomery modular multiplication algorithm. IEEE Transactions on Computers **60**(7), 923–936 (2011). https://doi.org/10.1109/TC.2010.247
17. Kaya Koc, C., Acar, T., Kaliski, B.: Analyzing and comparing montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996). https://doi.org/10.1109/40.502403
18. Khan, S., Javeed, K., Shah, Y.A.: High-speed fpga implementation of full-word montgomery multiplier for ecc applications. Microprocessors and Microsystems **62**, 91–101 (2018). https://doi.org/https://doi.org/10.1016/j.micpro.2018.07.005, https://www.sciencedirect.com/science/article/pii/S0141933117302843
19. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**(177), 203–209 (Jan 1987)
20. Liu, R., Li, S.: A design and implementation of montgomery modular multiplier. In: 2019 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1–4 (2019). https://doi.org/10.1109/ISCAS.2019.8702684
21. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**, 519–521 (1985)
22. Negre, C., Plantard, T.: Efficient modular arithmetic in adapted modular number system using lagrange representation. In: Mu, Y., Susilo, W., Seberry, J. (eds.) Information Security and Privacy. pp. 463–477. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
23. Noyez, L., El Mrabet, N., Potin, O., Veron, P.: Montgomery multiplication scalable systolic designs optimized for DSP48e2 **17**(1), 1–31 (2024), https://dl.acm.org/doi/10.1145/3624571, ACM Transactions on Reconfigurable Technology and Systems
24. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (feb 1978), https://doi.org/10.1145/359340.359342
25. Solinas, J.: Pseudo-Mersenne Prime, pp. 482–483. Springer US, Boston, MA (2005), https://doi.org/10.1007/0-387-23483-7_325, encyclopedia of Cryptography and Security
26. Solinas, J.A.: Generalized mersenne prime. In: Van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, pp. 509–510. Springer US (2011), http://link.springer.com/10.1007/978-1-4419-5906-5_32
27. Tenca, A.F., Çetin Kaya Koç: A scalable architecture for montgomery multiplication. In: Workshop on Cryptographic Hardware and Embedded Systems (1999)
28. Tenca, A.F., Todorov, G., Koç, Ç.K.: High-radix design of a scalable modular multiplier. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems — CHES 2001. pp. 185–201. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
29. Öztürk, E.: Design and implementation of a low-latency modular multiplication algorithm. IEEE Transactions on Circuits and Systems I: Regular Papers **67**(6), 1902–1911 (2020). https://doi.org/10.1109/TCSI.2020.2966755

A AMNS sagemath toolchain example

PMNS_sage_example

April 17, 2024

[25]:
```
load("PMNS.sage")
import random
```

[26]:
```
width = 256

p = random_prime(2**width-1, False, 2**(width-1))

w = 17

N = 5

LAMBDA = 5

AMNS_inst = PMNS(p, f"x**{N}-{LAMBDA}", phi_word_width = w)
```

[27]:
```
print("rho: ",AMNS_inst.rho)
print("phi: ", AMNS_inst.phi)
```

```
rho:  18014398509481984
phi:  295147905179352825856
```

[28]:
```
a = random.randrange(2**(width-1), p)
b = random.randrange(2**(width-1), p)
print("a: ", a)
print("b: ", b)
```

```
a:
62577267816865040374513551895071381207230729393424714378763245797825372801858
b:
59019625754205588799238007112607496435520975214689516963400202087396311124454
```

[29]:
```
A = AMNS_inst(a)
B = AMNS_inst(b)
print("A: ", A)
print("B: ", B)
```

```
A:  -14534195664711727*x^4 - 8975346308491892*x^3 - 17266101555795343*x^2 -
38044210229715083*x - 63296665373787115
```

```
B:  -13839353480681752*x^4 - 10102002257426769*x^3 - 20338179628853391*x^2 -
38033682203030272*x - 54656749766418919
```

[30]:
```
print("a: ", ZZ(A))
print("b: ", ZZ(B))
```

```
a:
625772678168650403745135518950713812072307293934247143787632457978253728018 58
b:
590196257542055887992380071126074964355209752146895169634002020873963111244 54
```

[32]:
```
print("C: ", A*B)
print("a*b[p]: ", a*b % p)
print("C_conv: ", (A*B).conv_out())
```

```
C:  -1682210683506286*x^4 - 2049296723737705*x^3 - 3278128968715415*x^2 -
2586983793371265*x - 7338291625272665
a*b[p]:
155446106597317452687272414515480909409866175648688485364449887394152070798 53
C_conv:
155446106597317452687272414515480909409866175648688485364449887394152070798 53
```

[35]:
```
print("a^25[p]: ", power_mod(a, 25, p))
print("A^25: ", A**25)
print("A^25_conv: ", (A**25).conv_out())
```

```
a^25[p]:
604274711860103149684527020778533305922636508331261629849331560181002255820 42
A^25:  -1893350003871704*x^4 - 98327871230488*x^3 - 4843428483133068*x^2 -
5536477699931177*x - 4855191540622047
A^25_conv:
604274711860103149684527020778533305922636508331261629849331560181002255820 42
```

B Estimation of time/resource cost graphs

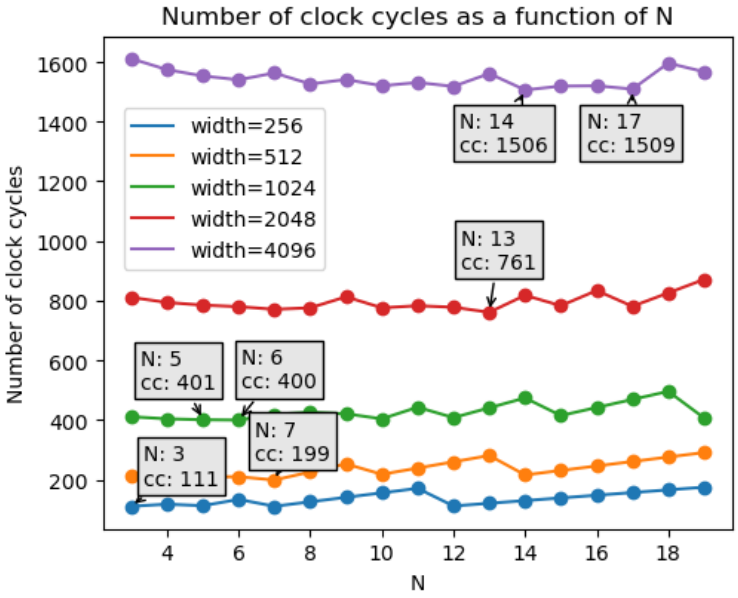

Fig. 11. Approximate number of clock cycles as a function of N

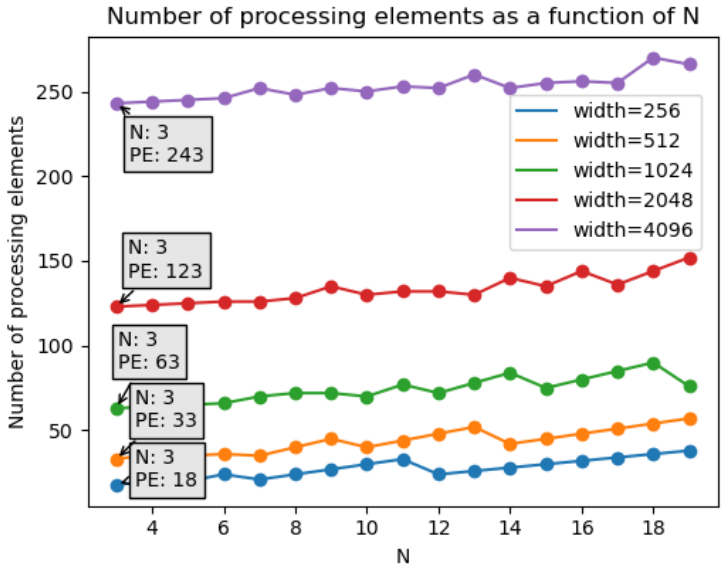

Fig. 12. Approximate number of processing elements as a function of N

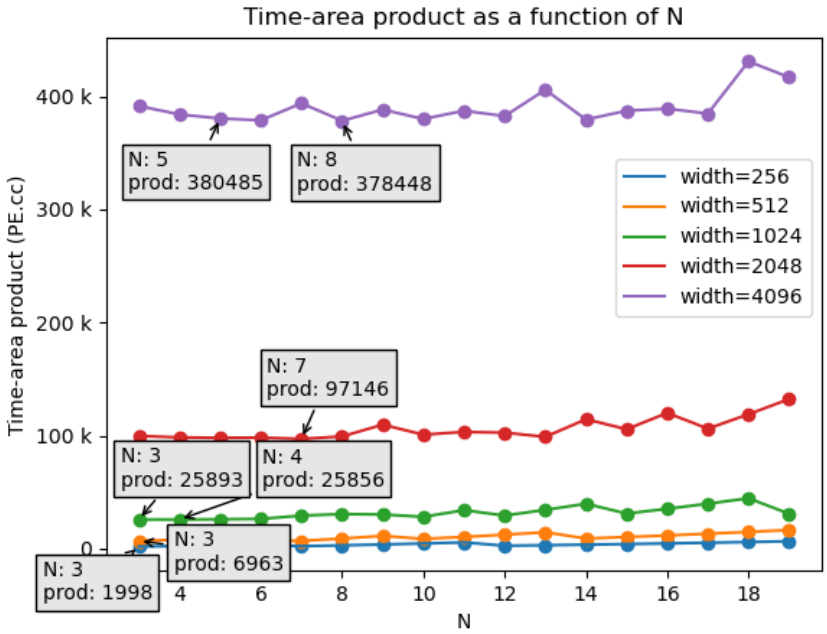

Fig. 13. Time-area product of PEs and clock cycles as a function of N