Multi-Precision Fast Modular Multiplication

Ido Atlas
ido@ingonyama.com

Tomer Solberg
tomer@ingonyama.com

Yuval Domb
yuval@ingonyama.com

Abstract

The Barrett-Domb modular multiplication is a low-complexity, hardware-friendly modular multiplication algorithm based on the novel Barrett modular reduction scheme. In this work, we introduce a multi-precision, GPU/CPU friendly, version of the Barrett-Domb algorithm. We show that our scheme is competitive with the widely used Montgomery modular multiplication, potentially retiring the need for the cumbersome Montgomery conversions.

1 Barrett-Domb Revisited

Let us consider the problem of calculating the product of two integers a, b modulo an integer s

$$r = ab \bmod s \tag{1}$$

where $0 \leq a, b < s$. Barrett's reduction [Bar86] calculates the quotient l such that

$$ab = ls + r \tag{2}$$

with $0 \leq r < s$, and then subtracts ls from ab. Note that $l < s$ since $ab < s^2$. The Barrett-Domb reduction [Dom22] is a method to estimate ls by breaking it into two distinct stages, as depicted in Figure 1. The reader is referred to [Dom22] for a detailed definition of the single-scalar method.

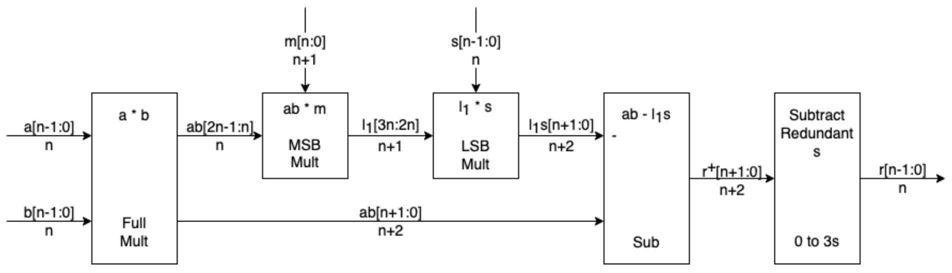

Figure 1: Original Barrett-Domb Scheme

We revisit [Dom22] providing somewhat more accurate treatment to the following aspects:

1. Correct the error bound on l_1
2. Show that $n = \lceil \log_2 s \rceil$ bits suffice for representing l_1
3. Slightly modify the MSB multiplier in preparation for the multi-precision scheme

1.1 Corrected error bound on l_1

Begin with Barrett's estimation of s^{-1}

$$m(n) \equiv \left\lfloor \frac{2^{2n}}{s} \right\rfloor \tag{3}$$

$$\widehat{s^{-1}} \equiv \frac{m(n)}{2^{2n}} < s^{-1} \tag{4}$$

$$e(\widehat{s^{-1}}) \equiv s^{-1} - \widehat{s^{-1}} < 2^{-2n} \tag{5}$$

and use it to estimate l as

$$l_0 \equiv ab \cdot \widehat{s^{-1}} = \frac{ab \cdot m(n)}{2^{2n}} \tag{6}$$

$$e(l_0) \equiv ab \cdot s^{-1} - ab \cdot \widehat{s^{-1}} = ab \cdot e(\widehat{s^{-1}}) < 2^{2n} \cdot 2^{-2n} = 1 \tag{7}$$

Remove the n least significant bits from ab at the cost of increased error

$$\begin{aligned}
l_* &\equiv [ab]_{\mathbf{msb}} \cdot \widehat{s^{-1}} && (8)\\
&= \frac{ab[2n-1:n] \cdot m(n)}{2^n} && (9)\\
&= l_0 - \frac{ab[n-1:0] \cdot m(n)}{2^{2n}} && (10)\\
&< l_0 - \frac{2^n \cdot 2^{n+1}}{2^{2n}} && (11)\\
&= l_0 - 2 && (12)\\
l_1 &\equiv \lfloor l_* \rfloor && (13)
\end{aligned}$$

and note that

$$e(l_1) < 4 \tag{14}$$

correcting the bound in Equation (17) of [Dom22] since $e(l_*) < 3$.

1.2 Minimal bit representation of l_1

In the original paper, l_1 was represented using $n+1$ bits where only n bits are actually required, since

$$l_1 < l_0 < l < s < 2^n \tag{15}$$

1.3 MSB multiplier modification

In the original scheme, the MSB multiplier takes as inputs $n \times (n+1)$ bits which can be reduced to $n \times n$ at the cost of a single addition. Observe that the most significant bit of m is always 1 since

$$2^n = \left\lfloor \frac{2^{2n}}{2^n} \right\rfloor < m(n) = \left\lfloor \frac{2^{2n}}{s} \right\rfloor < \left\lfloor \frac{2^{2n}}{2^{n-1}} \right\rfloor = 2^{n+1} \tag{16}$$

This leads to the following implementation

$$(ab[2n-1:0] \cdot m[n:0])[3n-1:2n] = (ab[2n-1:0] \cdot m[n-1:0])[3n-1:2n] + ab[2n-1:0] \tag{17}$$

which is a single $n \times n \to n_{\mathbf{msb}}$ multiplier followed by a single adder with no carry bit required.

1.4 The modified scheme

We conclude this section with the modified Barrett-Domb scheme depicted in Figure 2.

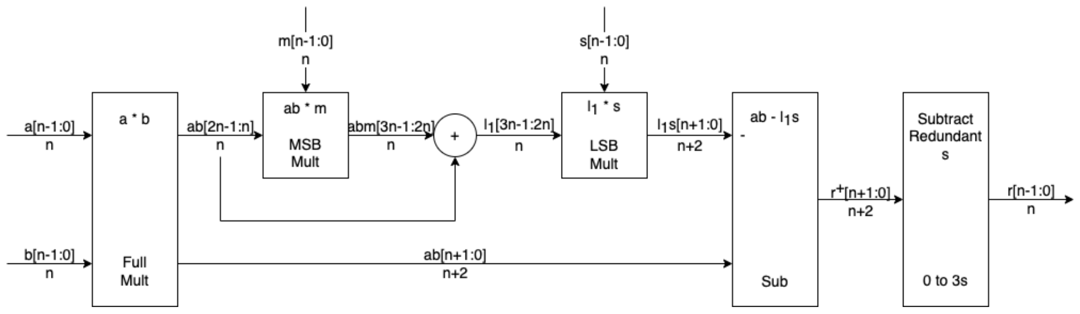

Figure 2: Modified Barrett-Domb Scheme

2 Multi-Precision Multipliers

Assume now that our calculation is being performed on a machine with a digit size of w bits. This machine has basic building blocks which can perform $w \times w \to 2w$ bit multiplications, $2w + 2w \to 2w + 1$ bit additions, and bit-wise shifts. We consider the case where $n = \lceil \log_2(s) \rceil > w$, requiring multiple digits to represent a single operand. This is identical to working with digits in a numerical system base $B = 2^w$.

Initially assume, for simplicity, that $n = wk$ where $k \in \mathbf{N}$. This restriction will be lifted later on. In order to generalize to the multi-precision scheme let us replace each single-precision block with its multi-precision counterpart.

2.1 Multi-precision full multiplier

Given unsigned integers: $X, Y < 2^n$ let us compute the product $X \cdot Y$, on a processor with digit size w bits. To do so, split X and Y into digits

$$X = \sum_{i=0}^{k-1} x_i \cdot B^i, \qquad 0 \le x_i \le B - 1 \tag{18}$$

$$Y = \sum_{i=x0}^{k-1} y_i \cdot B^i, \qquad 0 \le y_i \le B - 1 \tag{19}$$

and then sum the products of digits using

$$X \cdot Y = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} x_i \cdot y_j \cdot B^{i+j} \tag{20}$$

Note that $X \cdot Y$ is a $2k$-digit number. This multiplication requires k^2 machine (single-digit) multiplications and $k^2 - 1$ machine (double-digit) additions. It is worth noting that this is the naive way of multiplying multi-digit numbers and there are methods to reduce the number of digit multiplications which is beyond the scope of this paper.

2.2 Multi-precision LSB multiplier

The k least significant digits of the product $X \cdot Y$ are equivalent to the $n = kw$ least significant bits of the product. This can be represented as the following sum of digit products

$$[X \cdot Y]_{LSB} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} x_i \cdot y_j \cdot B^{i+j} \bmod B^k \tag{21}$$

It helps to visualize the required digit products as a matrix where the entry $[i, j]$ represents the product $x_i \cdot y_j \cdot B^{i+j}$. Taking for example $k = 4$ leads to

$$\begin{bmatrix} x_0y_3B^3 & x_0y_2B^2 & x_0y_1B & x_0y_0 \\ x_1y_3B^4 & x_1y_2B^3 & x_1y_1B^2 & x_1y_0B \\ x_2y_3B^5 & x_2y_2B^4 & x_2y_1B^3 & x_2y_0B^2 \\ x_3y_3B^6 & x_3y_2B^5 & x_3y_1B^4 & x_3y_0B^3 \end{bmatrix} \tag{22}$$

where the upper triangle clearly consists of all necessary products. We use this to reorder the summation in (20)

$$X \cdot Y = \sum_{d=0}^{k-1} \sum_{i=0}^{d} x_i \cdot y_{d-i} \cdot B^d \tag{23}$$

$$+ \sum_{d=k}^{2k-2} \sum_{i=d^*}^{k-1} x_i \cdot y_{d-i} \cdot B^d \tag{24}$$

where the first and second terms correspond to the upper triangle and lower triangle without the diagonal of (22) and $d^* \equiv d-(k-1)$. Thus, the number of digit multiplications required for the LSB multiplier, corresponding to the upper triangle of (22), is

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2} \tag{25}$$

Let us note that in our scheme, the LSB multiplier is expected to provide a few extra output bits. That is, we would like to have $n \times n \to n + b$ rather than $n \times n \to n$ where b is small (typically 2). There are a few ways to handle this:

1. The simplest method is to calculate another full diagonal of (22). Assuming that $b \leq w$ this is sufficient. However, this increases the cost of digit multiplications to

$$\frac{k(k+1)}{2} + (k-1) = \frac{1}{2}(k^2 + 3k - 2) \tag{26}$$

2. Since the typical case only requires very few extra bits (i.e. $b \ll w$), this can be handled efficiently using specialized or dedicated hardware.

3. Often $n < k \cdot w$ and the most significant digit is not fully utilized. When $n \leq k \cdot w + b$, the redundant b bits can be handled with no extra cost incurred. This is, in fact, the case for many cryptographic applications such as Elliptic Curve Cryptography (ECC) using curves BN254, BLS12-377, BLS12-381 [Wan, Ark, Edg], where the field arithmetic only requires 254, 377, and 381 bits, leaving 2, 7, and 3 free bits, respectively. This is further discussed in Section 3.1.

2.3 Multi-precision MSB multiplier

The MSB multiplier analysis is similar to the LSB. However, it requires the most significant k digits, which lie on the lower triangle of the matrix (22). Let us rewrite equation (23) with the diagonal moved to the lower triangle summation as

$$X \cdot Y = \sum_{d=0}^{k-2} \sum_{i=0}^{d} x_i \cdot y_{d-i} \cdot B^d \tag{27}$$

$$+ \sum_{d=k-1}^{2k-2} \sum_{i=d^*}^{k-1} x_i \cdot y_{d-i} \cdot B^d \tag{28}$$

Computing only the second term in this expression introduces an error. Since the result of this multiplier is an estimation of l, this error can be compensated for at the final subtraction stage. Let us upper bound the error as

$$e_{\mathbf{msb}} = \sum_{d=0}^{k-2} \sum_{i=0}^{d} x_i \cdot y_{d-i} \cdot B^{d-k} \tag{29}$$

$$< \sum_{d=0}^{k-2} \sum_{i=0}^{d} B^{d-k+2} \tag{30}$$

$$= \sum_{d=0}^{k-2} (d+1) B^{d-k+2} \tag{31}$$

$$= (k-1) + (k-2)B^{-1} + ... + B^{-k+2} \tag{32}$$

Assuming reasonably that $B > k$ leads to

$$e_{\mathbf{msb}} < k \tag{33}$$

Note that this upper error bound can be reduced to 1 by calculating another full diagonal of (22). However, the number of multiplications resulting from that would increase to $\frac{1}{2}(k^2 + 3k - 2)$, instead of $\frac{1}{2}k(k+1)$ as shown in (26).

One should note that a higher error requires an increase in the number of output bits for the LSB multiplier, from 2 to $\log(4 + k)$. Typical values of k for ECC calculations are between 8 and 16, adding up to 4 more bits. This does not change the essence of the three possible solutions for handling those bits (see Section 2.2). However, it may change the specific solution of choice, as discussed later on.

3 The Multi-Precision Scheme

Replacing the blocks in the Barrett-Domb scheme of Figure 2 with their multi-precision counterparts results in the scheme presented in Figure 3. Note that this scheme is limited to the precise bit-lengths case where $n = wk$.

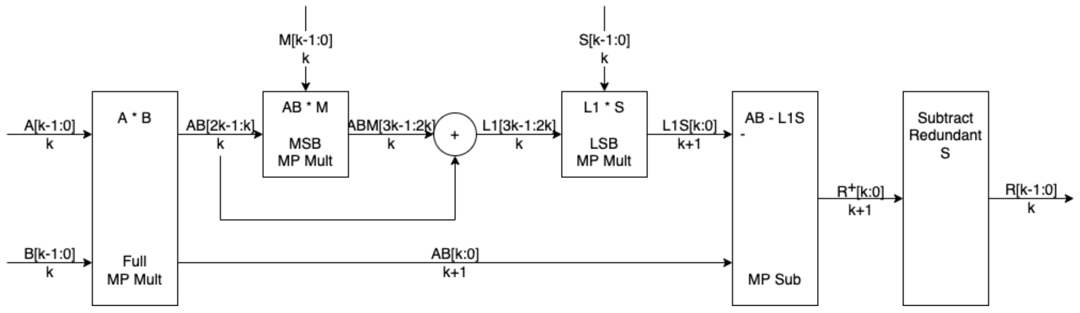

Figure 3: Precise Multi-Precision Barrett-Domb Scheme

3.1 Imprecise bit-lengths

Until now, we assumed that $n = wk$. When this does not apply, more imprecision is added to the MSB multiplier, since now the upper k digits of the product ab no longer contain its n upper bits. This can be corrected by taking $ab[2k-1:k-1]$, the top $k+1$ digits of ab, and bit-shifting them to the left by $2z$, where $z = wk - n$. The output of this operation is by definition exactly k digits. This also requires using $m(n + z/2)$ rather than $m(n)$ that was used in the single-precision scheme (see Equation (3)). The output of the MSB multiplier now requires a right bit-shift of z bits. The imprecise multi-precision scheme is depiced in Figure 4.

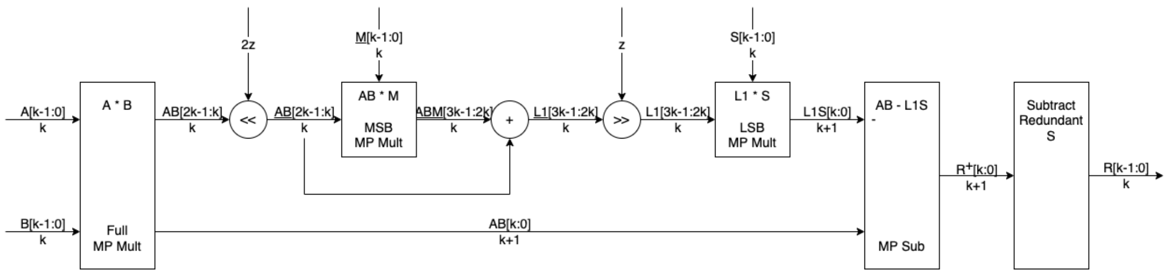

Figure 4: Imprecise Multi-Precision Barrett-Domb Scheme

Note that the suggested shifting scheme adds z extra bits on top of the required (missing) z bits. These additional bits improve the precision of the MSB multiplier and thus reduce the overall error of the LSB multiplier output to $\log(4 + k/2^z)$. The LSB multiplier then also enjoys the added precision of z bitswhich means that we can use the minimal number of multipliers $k(k+1)/2$ for both the MSB and the LSB multipliers, given that the following condition holds

$$z \geq \log(4 + k/2^z) \qquad (34)$$

Consider a few typical examples, assuming $w = 32$:

- For BN254 curve, $k = 256/32 = 8$ and $z = 256 - 254 = 2$. $\log(4 + k/2^z) = \log 6 > 2$, so the above condition does not hold and the LSB multiplier will need $\lceil \log 6 - 2 \rceil = 1$ extra bit. This should either be fixed with dedicated logic or by adding a full digit to the calculation, adding $k - 1$ single-digit multipliers.
- For BLS12-381 curve, $k = 384/32 = 12$ and $z = 384 - 381 = 3$. $\log(4 + k/2^z) = \log 5.5 < 3$, so the condition holds and we need the minimal number of multipliers.
- For BLS12-377 curve, $k = 384/32 = 12$ and $z = 384 - 377 = 7$. $\log(4 + k/2^z) \approx \log 4 < 7$, so the condition holds and we need the minimal number of multipliers.

Finally, note that in systems where bit-shifts are costly, the scheme can be reduced to a single left bit-shift on ab prior to the MSB multiplier. This may increase the number of required multiplications, in a similar manner to our previous discussions, and is left to the reader to consider for specific cases.

3.2 The multi-precision Barrett-Domb pseudo code

Algorithm 1 Barrett-Domb modular multiplication

```
1: function MP-BARRETT-DOMB-MULT($A, B, S$)
2:     $z \leftarrow km - n$
3:     $n \leftarrow \lceil log_2(S) \rceil$
4:     $\underline{M} \leftarrow \lfloor (2^{2n+z}/s) \rfloor$
5:     $AB \leftarrow A \cdot B$
6:     $\underline{AB} \leftarrow AB[2k-1:k-1] \cdot 2^{2z}$
7:     $\underline{L_1} \leftarrow [\underline{AB}[2k-1:k] \cdot \underline{M}]_{MSB}$
8:     $\underline{L_1} \leftarrow \underline{L_1} + \underline{AB}[2k-1:k]$
9:     $L_1 \leftarrow \lfloor \underline{L_1} \cdot 2^{-z} \rfloor$
10:    $LS \leftarrow [L_1 \cdot S]_{LSB}$
11:    $R \leftarrow AB[k-1:0] - LS$
12:    while $R < S$ do
13:        $R \leftarrow R - S$
14:    end while
15:    return $R$
16: end function
```

Note that the subtraction while loop should be replaced with a standard log-ladder subtractor.

4 Comparison with Montgomery Reduction

The Montgomery reduction [Mon85] is the main method used today for modular multiplication. As with our method, it starts with a full multiplication of the inputs and follows with a reduction scheme. Unlike our scheme, the Montgomery method requires that the numbers are first converted into a special form, called the Montgomery form, in which all calculations take place. At the end of the calculations, the results need to be converted back into the usual representation. The Mongomery multiplication algorithm takes two numbers A, B in the Montgomery form, and computes their product modulo another number s. In a multi-precision scheme, we define $r = 2^w$ where w is the digit size, and the numbers A, B and the modulus s are given as length-k arrays of digits, for example

$$A = \sum_{i=0}^{n-1} a_i r^i \tag{35}$$

The Montgomery multiplication also requires as input the number $\mu = -s^{-1} \bmod r$ which can be precomputed, it then produces as output $C = ABr^{-n} \bmod s$ which is the Montgomery form of the desired reduced product. The Montgomery multiplication can be described by Algorithm 2 [BM17], in which only the number A is explicitly broken into digits, and multiplications are presented as either digit-number (d-n) or digit-digit (d-d) multiplications.

Algorithm 2 Montgomery modular multiplication

```
 1: function MontgomeryMult($a_i, B, s$)            ▷ $i \in \{0, 1, \dots, k-1\}$
 2:     $R \leftarrow 0$                              ▷ $R$ has $k+1$ digits
 3:     for $i = 0$ to $k-1$ do
 4:         $R \leftarrow R + a_i B$                  ▷ d-n multiplication
 5:         $q \leftarrow \mu R \bmod r$              ▷ d-d multiplication
 6:         $R \leftarrow (R + qs)/r$                 ▷ d-n multiplication
 7:     end for
 8:     if $R \geq s$ then
 9:         $R \leftarrow R - s$
10:     end if
11:     return $R$
12: end function
```

Counting the operations we see that each iteration of the for loop contains $2k + 1$ d-d multiplications. The entire algorithm then contains $2k^2 + k$ d-d multiplications. Discounting the AB multiplication which requires k d-d multiplications for both Montgomery and Barrett-Domb, we are left with $k^2 + k$ d-d multiplications for the Montgomery reduction. We can now compare this to the Barrett-Domb multiplication presented here. We saw that there are 3 options for its implementation:

1. Minimal - both LSB and MSB mults contain $k(k+1)/2$ d-d multiplications or in total $k^2 + k$ d-d multipliers. This is exactly the same as the Montgomery multiplication, and our method gains a distinct advantage as it eliminates the need for conversions.

2. Intermediate - either the LSB or MSB mults require an added diagonal worth of multiplications, adding up to a total of $k^2 + 2k - 1$ d-d multiplications, or around k more than Montgomery.

3. Maximal - both the LSB and MSB mults require an added diagonal worth of multiplications, adding up to a total of $k^2 + 3k - 2$ d-d multiplications, or around $2k$ more than Montgomery.

In the latter two options, one can ask which implementation has a higher computational overhead - Barrett-Domb with its extra multiplications, or Montgomery with the added conversions? This is of course dependent on the application, so we will compare by example.

4.1 NTT comparison

Consider for example the computation of an NTT with N samples. This is a calculation that takes N numbers as input, performs approximately $N \log N$ multiplication, and returns N numbers as outputs. Converting into the Montgomery form requires applying the Montgomery multiplication for every input, requiring $2k^2 + k$ d-d multipliers. Converting back to the usual form requires applying the Montgomery reduction (without the first multiplier) for every output, requiring $k^2 + k$ d-d multipliers. Thus, a Montgomery implementation consists of $(k^2 + k)N \log N + (3k^2 + 2k)N$ d-d multiplications. A minimal Barrett-Domb implementation (where applicable) consists of $(k^2 + k)N \log N$ d-d multiplications, while an intermediate one consists of $(k^2 + 2k - 1)N \log N$ d-d multiplications, and a maximal consists of $(k^2 + 3k - 2)N \log N$. The number of extra multiplications that are required to be performed by a Montgomery implementation versus each one of the above Barrett-Domb schemes is as follows:

$$D_{\mathbf{min}} = (3k^2 + 2k)N \tag{36}$$
$$D_{\mathbf{int}} \approx kN(3k - \log N) \tag{37}$$
$$D_{\mathbf{max}} \approx kN(3k - 2\log N) \tag{38}$$

where one must note that $D_{\mathbf{int}}$ and $D_{\mathbf{max}}$ may be negative for the cases where Montgomery is advantageous. We see that the intermediate Barrett-Domb implementation gains an advantage if $k > 1/3 \log N$. Typical values of k are 8, 12, 16 (for a digit size of 32 bits and number sizes of 256, 384, 512 bits). For these values, the intermediate Barrett-Domb will have the upper hand for NTT sizes smaller than $2^{24}, 2^{36}, 2^{48}$ respectively. The maximal Barrett-Domb will similarly have the upper hand for NTT sizes smaller than $2^{12}, 2^{18}, 2^{36}$. These results are summarized in Table 1.

Implementation:	Minimal	Intermediate	Maximal
$k = 8$	Always	$N < 2^{24}$	$N < 2^{12}$
$k = 12$	Always	$N < 2^{36}$	$N < 2^{18}$
$k = 16$	Always	$N < 2^{48}$	$N < 2^{24}$

Table 1: Looking at the three possible implementations of the Barrett-Domb multiplier, and three typical values of k, each cell in the table gives the condition for which the Barrett-Domb multiplier has an advantage due to Montgomery's conversions, for $w = 32$

4.2 Other comparisons

In general, as the ratio between the number of elements and the number of multiplication increases, so does the advantage of the Barrett-Domb multiplier (for non-minimal implementations) over Montgomery. So while the minimal implementation will always have the advantage, in applications with few conversions and a large number of multiplications Montgomery would have a clear advantage over the intermediate and maximal implementations. In contrast, in applications with a similar number of conversions and multiplications, the Barrett-Domb will generally perform better. Several such examples:

- Modular exponentiation takes $O(1)$ elements and performs $O(N)$ multiplications, so Montgomery will be advantageous.

- General linear transformations (general matrix acting on a vector) take $O(N)$ elements and perform $O(N^2)$ multiplications, so Montgomery will be advantageous.
- Hadamard product (element by element) performs the same number of conversions and multiplications (typically $O(N)$ or $O(N^2)$ for both), so Barrett-Domb will be advantageous.
- Scalar product saves on the output conversions, but input conversions are still the same as the Hadamard product, so Barrett-Domb will be advantageous.

Recall that in all cases, the minimal Barrett-Domb, if feasible (depending on the bit length of s), will be advantageous.

5 Implementation and Future Work

A reference code written in Python can be found in [Ing]. A natural next step is to adapt and integrate the code into leading GPU libraries that support finite field arithmetic of large integers. Concretely, Sppark [Spp], which implements NTT based on optimized Montgomery for GPU [ELWW16] is a good first candidate.

Other interesting directions include examining Karatsuba optimizations and performance over SIMD systems.

References

[Ark] Arkworks. Bls12-377 arkworks documentation. `https://docs.rs/ark-bls12-377/latest/ark_bls12_377/`.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BM17] Joppe W. Bos and Peter L. Montgomery. Montgomery arithmetic from a software perspective. *IACR Cryptol. ePrint Arch.*, 2017:1057, 2017. `https://eprint.iacr.org/2017/1057`.

[Dom22] Yuval Domb. Fast modular multiplication. 2022. `https://www.ingonyama.com/blogs/fast-modular-multiplication`.

[Edg] Ben Edgington. Bls12-381 for the rest of us. `https://hackmd.io/@benjaminion/bls12-381`.

[ELWW16] Niall Emmart, Justin Luitjens, Charles Weems, and Cliff Woolley. Optimizing modular multiplication for nvidia's maxwell gpus. In *2016 IEEE 23nd symposium on computer arithmetic (ARITH)*, pages 47–54. IEEE, 2016.

[Ing] Ingonyama. Multi precision barrett-domb python implementation. `https://github.com/ingonyama-zk/modular_multiplication`.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[Spp] Sppark. Zero-knowledge template library. `https://github.com/supranational/sppark`.

[Wan] Jonathan Wang. Bn254 for the rest of us. `https://hackmd.io/@jpw/bn254`.