Use of SIMD Features to Speed up Eta Pairing

Anup Kumar Bhattacharya[1], Sabyasachi Karati[1], Abhijit Das[1], Dipanwita Roychowdhury[1], Bhargav Bellur[2] and Aravind Iyer[2]

[1]Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur, India

[2]General Motors Technical Centre India
India Science Lab, Bangalore, India

[1]anup, skarati, abhij, drc@cse.iitkgp.ernet.in
[2]bhargav_bellur@yahoo.com, aravind.iyer@gm.com

Abstract. Eta pairing over supersingular elliptic curves is widely used in designing many cryptographic protocols. Because of efficiency considerations, curves over finite fields of small characteristics are preferred. In this paper, we report several of our implementations of eta pairing over finite fields of characteristics two and three. We exploit SIMD features available in Intel processors to speed up eta-pairing computations. We study two ways of vectorizing the computations: horizontal (intra-pairing) and vertical (inter-pairing). We report our experimental results using SSE2 and AVX2 features supported by the Haswell microarchitecture. Our implementations use two popular curves. Recently proposed discrete-logarithm algorithms make these curves less secure than previously thought. We discuss the implications of these developments in the context of our implementations.

Keywords: Supersingular elliptic curves, eta pairing, software implementation, SIMD, SSE intrinsics, AVX intrinsics.

1 Introduction

Pairing over algebraic curves are extensively used [11, 12, 23] in designing cryptographic protocols. There are two advantages of using pairing in these protocols. Some new functions are realized using pairing [11, 23]. Many other protocols [12] achieve small signature sizes at the same security level.

Miller's algorithm [31] is an efficient way to compute pairing. Tate and Weil are two main variants of pairing functions on elliptic curves, with Tate pairing computation being significantly faster than Weil pairing for small fields. In the last few years, many variants of Tate pairing [8, 20, 28] are proposed to reduce the computation complexity of Tate pairing substantially. Eta pairing [8] is one such variant defined for supersingular curves. Some pairing-friendly families [15] of curves are defined over prime fields and over fields of characteristics two and three. Vercauteren [38] proposes the concept of optimal pairing which gives lower bounds on the number of Miller iterations required to compute pairing.

There have been many attempts to compute pairing faster. Barreto et al. [9] propose many simplifications of Tate-pairing algorithms. Final exponentiation is one such time-consuming step in pairing computation. Scott et al. [35] propose elegant methods to reduce the complexity of final exponentiation. Ahmadi et al. [3] and Granger et al. [18] describe efficient implementations of arithmetic in fields of characteristic three for faster pairing computation. Multi-core implementations of Tate pairing are reported in [5, 10]. Beuchat et al. [10] provide an estimate on the optimal number of cores needed to compute pairing in a multi-core environment. GPU-based implementations of eta pairing are reported in [13, 25].

Many low-end processors are released with SIMD facilities which provide the scope of parallelization in resource-constrained applications. SIMD-based implementations of pairing are reported in [5, 10, 19]. All these data-parallel implementations vectorize individual pairing computations, and vary in their approaches to exploit different SIMD intrinsics in order to speed up the underlying field arithmetic. This technique is known as horizontal vectorization.

The other SIMD-based vectorization technique, vertical vectorization, has also been used for efficient implementation purposes. Montgomery [32] applies vertical vectorization to Elliptic Curve Method to factor integers. For RSA implementations using SSE2 intrinsics, Page and Smart [33] use two SIMD-based techniques called inter-operation and intra-operation parallelisms. Grabher et al. [17] propose digit slicing to reduce carry-handling overhead in the implementation of ate pairing over Barreto-Naerhig curves defined over prime fields. Implementation results with both inter-pairing and intra-pairing parallelism techniques are provided and a number of implementation strategies are discussed.

Intuitively, so long as different instances of some computation follow fairly the same sequence of basic CPU operations, parallelizing multiple instances (vertical vectorization) would be more effective than parallelizing each such instance individually (horizontal vectorization). Computation of eta pairing on curves over fields of small characteristics appears to be an good setting for vertical vectorization. This is particularly relevant because all the parties in a standard elliptic-curve-based protocol typically use the same curve and the same base point (unlike in RSA where different entities use different public moduli).

Each of the two vectorization models (horizontal and vertical) has its private domains of applicability. Even in the case of pairing computation, vertical vectorization does not outperform horizontal vectorization in every step. For example, comb-based multiplication [29] of field elements is expected to be more efficient under vertical vectorization than under horizontal vectorization. On the contrary, modular reduction using polynomial division seems to favor horizontal vectorization more than vertical vectorization, since the number of steps in the division loop and also the shift amounts depend heavily on the operands. This problem can be bypassed by using defining polynomials with a small number of non-zero coefficients (like trinomials or pentanomials). However, computing inverse by the extended polynomial gcd algorithm cannot be similarly tackled. Moreover, vertical vectorization is prone to encounter more cache misses compared to horizontal vectorization and even to non-SIMD implementation. The

effects of cache misses are rather pronounced for algorithms based upon lookup tables (like comb methods).

Despite all these potential challenges, vertical vectorization may be helpful in certain cryptographic operations. Our experimentation with SSE2 and AVX2 intrinsics reveals that this is the case for eta pairing on supersingular curves over fields of characteristics two and three. More precisely, horizontal vectorization leads to speedup of up to 30% over non-SIMD implementation. Vertical vectorization, on the other hand, yields speedup in the range 25–55%. In short, the validation of the effectiveness of vertical vectorization in pairing computations is the main technical contribution of this paper.

We take two popular supersingular elliptic curves defined over the fields $\mathbb{F}_{2^{1223}}$ and $\mathbb{F}_{3^{509}}$. At the time we started this work, eta pairing over these curves were believed to offer 128-bit security. Recently proposed finite-field discrete-logarithm algorithms [6, 24] indicate that their security guarantees are much less. For example, Adj et al. [2] estimate that the curve over $\mathbb{F}_{3^{509}}$ offers slightly more than 80-bit security. As a result, we have to use curves over much larger fields in order to restore the security level to 128 bits. In another paper, Adj et al. [1] demonstrate that a curve defined over $\mathbb{F}_{2^{3041}}$ provides 129-bit security. All our SIMD-based techniques can be ported *mutatis mutandis* to curves defined over fields larger than what we study in this paper. However, larger fields imply slower implementations of eta pairing, and that in turn highlights the necessity of achieving better speedup figures. We expect that our SIMD-based implementations of eta pairing appropriately address this necessity.

The rest of the paper is organized as follows. Section 2 reviews the notion of pairing, and lists the algorithms used to implement field and curve arithmetic. Section 3 describes horizontal and vertical vectorization models. We intuitively explain which of the basic operations are likely to benefit more from vertical vectorization than from horizontal vectorization. We give special attention to field multiplication. Our experimental results are tabulated in Section 4. We conclude the paper in Section 5 after highlighting some potential areas of future research.

2 Background on Eta Pairing

In this section, we briefly describe standard algorithms that we have used for implementing arithmetic in extension fields of characteristics two and three. We subsequently state Miller's algorithm for the computation of eta pairing on supersingular curves over these fields.

2.1 Eta Pairing in a Field of Characteristic Two

We implemented eta pairing over the supersingular elliptic curve $y^2 + y = x^3 + x$ defined over the binary field $\mathbb{F}_{2^{1223}}$ represented as an extension of $\mathbb{F}_2$ by the irreducible polynomial $x^{1223} + x^{255} + 1$. An element of $\mathbb{F}_{2^{1223}}$ is packed into an array of 64-bit words. The basic operations on such elements are done as follows.

- Addition: We perform word-level XOR to add multiple coefficients together.

– Multiplication: Computing products $c = ab$ in the field is costly, but needed most often in Miller's algorithm. Comb-based multiplication [29] with four-bit windows is used in our implementations.
– Inverse: We use the extended Euclidean gcd algorithm for polynomials to compute the inverse of an element in the binary field.
– Square: We use a precomputed table of square values for all possible 8-bit inputs.
– Square Root: The input element is written as $a(x^2) + xb(x^2)$. Its square root is computed as $a(x) + x^{1/2}b(x)$, where $x^{1/2} = x^{612} + x^{128}$.
– Reduction: Since the irreducible polynomial defining $\mathbb{F}_{2^{1223}}$ has only a few non-zero coefficients, we use a fast reduction algorithm (as in [34]) for computing remainders modulo this polynomial.

The embedding degree for the supersingular curve stated above is four. So we need to work in the field $\mathbb{F}_{(2^{1223})^4}$. This field is represented as a tower of two quadratic extensions over $\mathbb{F}_{2^{1223}}$. The basis for this extension is given by $(1, u, v, uv)$, where $g(u) = u^2 + u + 1$ is the irreducible polynomial for the first extension, and $h(v) = v^2 + v + u$ defines the second extension. The distortion map is given by $\psi(x, y) = (x + u^2, y + xu + v)$.

Addition in $\mathbb{F}_{(2^{1223})^4}$ uses the standard word-wise XOR operation on elements of $\mathbb{F}_{2^{1223}}$. Multiplication in $\mathbb{F}_{(2^{1223})^4}$ can be computed by six multiplications in the field $\mathbb{F}_{2^{1223}}$ [19].

Algorithm 1 describes the computation of eta pairing η_T. This is an implementation [19] of Miller's algorithm for the supersingular curve $E_2 : y^2 + y = x^3 + x$ under the above representation of $\mathbb{F}_{2^{1223}}$ and $\mathbb{F}_{(2^{1223})^4}$. Here, the point $P \in E_2(\mathbb{F}_{2^{1223}})$ on the curve has prime order r. Q too is a point with both coordinates from $\mathbb{F}_{2^{1223}}$. The distortion map is applied to Q. Algorithm 1 does not explicitly show this map. The output of the algorithm is an element of μ_r, the order-r subgroup of $\mathbb{F}^*_{(2^{1223})^4}$.

Algorithm 1 Eta Pairing Algorithm for a Field of Characteristic Two

Input: $P = (x_1, y_1), Q = (x_2, y_2) \in E(\mathbb{F}_{2^{1223}})[r]$
Output: $\eta_T(P, Q) \in \mu_r$

$T \leftarrow x_1 + 1$
$f \leftarrow T \cdot (x_1 + x_2 + 1) + y_1 + y_2 + (T + x_2)u + v$

for $i = 1$ **to** 612 **do**
  $T \leftarrow x_1$
  $x_1 \leftarrow \sqrt{x_1}, y_1 \leftarrow \sqrt{y_1}$
  $g \leftarrow T \cdot (x_1 + x_2) + y_1 + y_2 + x_1 + 1 + (T + x_2)u + v$
  $f \leftarrow f \cdot g$
  $x_2 \leftarrow x_2^2, y_2 \leftarrow y_2^2$
end for

return $f^{(q^2-1)(q-\sqrt{2q}+1)}$, where $q = 2^{1223}$.

The complexity of Algorithm 1 is dominated by the 612 iterations (called Miller iterations), and exponentiation to the power $(q^2-1)(q-\sqrt{2q}+1)$ (referred to as the final exponentiation). In each Miller iteration, two square roots, two squares, and seven multiplications are performed in the field $\mathbb{F}_{2^{1223}}$. In the entire Miller loop, 1224 square roots and 1224 squares are computed, and the number of multiplications is 4284. Evidently, the computation of the large number of multiplications occupies the major portion of the total computation time. Each multiplication of $\mathbb{F}_{(2^{1223})^4}$ (computation of $f \cdot g$) is carried out by six multiplications in $\mathbb{F}_{2^{1223}}$. In these six multiplications, three variables appear as one of the two operands. Therefore only three precomputations (instead of six) are sufficient for performing all these six multiplications by the Lopez-Dahab method. For characteristic-three fields, such a trick is proposed in [37]. Using Frobenius endomorphism [19, 35], the final exponentiation is computed, so this operation takes only a small fraction of the total computation time.

2.2 Eta Pairing in a Field of Characteristic Three

The irreducible polynomial $x^{509}-x^{318}-x^{192}+x^{127}+1$ defines the extension field $\mathbb{F}_{3^{509}}$. The curve $y^2 = x^3 - x + 1$ defined over this field is used. Each element of the extension field is represented using two bit vectors [36]. The basic operations on these elements are implemented as follows.

- Addition and subtraction: We use the formulas given in [26].
- Multiplication: Comb-based multiplication [3] with two-bit windows is used in our implementations.
- Inverse: We use the extended Euclidean gcd algorithm for polynomials to compute the inverse of an element in the field.
- Cube: We use a precomputed table of cube values for all possible 8-bit inputs.
- Cube Root: The input element is first written as $a(x^3) + xb(x^3) + x^2c(x^3)$. Its cube root is computed as $a(x)+x^{1/3}b(x)+x^{2/3}c(x)$, where $x^{1/3} = x^{467} + x^{361} - x^{276} + x^{255} + x^{170} + x^{85}$, and $x^{2/3} = -x^{234} + x^{128} - x^{43}$ [3, 7]. We have not used the cube-root-friendly representation of $\mathbb{F}_{3^{509}}$ prescribed in [4].
- Reduction: We use a fast reduction algorithm [34] for computing remainders modulo the irreducible polynomial.

The embedding degree in this case is six, so we need to work in the field $\mathbb{F}_{(3^{509})^6}$. A tower of extensions over $\mathbb{F}_{3^{509}}$ is again used to represent $\mathbb{F}_{(3^{509})^6}$. The first extension is cubic, and is defined by the irreducible polynomial $u^3 - u - 1$. The second extension is quadratic, and is defined by $v^2 + 1$. The basis of $\mathbb{F}_{(3^{509})^6}$ over $\mathbb{F}_{3^{509}}$ is therefore $(1, u, u^2, v, uv, u^2v)$. The distortion map in this case is $\psi(x, y) = (u - x, yv)$.

For multiplying two elements of $\mathbb{F}_{(3^{509})^6}$, we have used 18 multiplications in $\mathbb{F}_{3^{509}}$ [27]. The method reported in [16], which uses only 15 such multiplications, is not implemented.

Algorithm 2 describes the computation of eta pairing [10] in the case of characteristic three. P and Q are points with both coordinates from $\mathbb{F}_{3^{509}}$. The

Algorithm 2 Eta Pairing Algorithm for a Field of Characteristic Three

Input: $P = (x_P, y_P), Q = (x_Q, y_Q) \in E(\mathbb{F}_{3^{509}})[r]$
Output: $\eta_T(P, Q) \in \mu_r$

$x_P \leftarrow \sqrt[3]{x_P} + 1$
$y_P \leftarrow -\sqrt[3]{y_P}$
$t \leftarrow x_P + x_Q$
$R \leftarrow -(y_P t - y_Q v - y_P u)(-t^2 + y_P y_Q v - tu - u^2)$
$X_P[0] \leftarrow x_P, Y_P[0] \leftarrow y_P$
$X_Q[0] \leftarrow x_Q, Y_Q[0] \leftarrow y_Q$

for $i = 1$ **to** 254 **do**
  $X_P[i] \leftarrow \sqrt[3]{X_P[i-1]}$
  $X_Q[i] \leftarrow X_Q^3[i-1]$
  $Y_P[i] \leftarrow \sqrt[3]{Y_P[i-1]}$
  $Y_Q[i] \leftarrow Y_Q^3[i-1]$
end for

for $i = 1$ **to** 127 **do**
  $t \leftarrow X_P[2i-1] + X_Q[2i-1]$
  $w \leftarrow Y_P[2i-1]Y_Q[2i-1]$
  $t' \leftarrow X_P[2i] + X_Q[2i]$
  $w' \leftarrow Y_P[2i]Y_Q[2i]$
  $S \leftarrow (-t^2 + wv - tu - u^2)(-t'^2 + w'v - t'u - u^2)$
  $R \leftarrow R \cdot S$
end for

return $f^{(q^3-1)(q+1)(q+\sqrt{3q}+1)}$, where $q = 3^{509}$.

distortion map is applied to Q. Algorithm 2 does not show this map explicitly. The order of P is a prime r, and μ_r is the order-r subgroup of $\mathbb{F}^*_{(3^{509})^6}$.

The first **for** loop of Algorithm 2 is a precomputation loop. The second **for** loop implements the Miller iterations. The final exponentiation in the last line uses Frobenius endomorphism [19, 35]. The most time-consuming operations involved in Algorithm 2 are 508 cubes, 508 cube roots and 3556 multiplications in the field $\mathbb{F}_{3^{509}}$ (given that one multiplication of $\mathbb{F}_{(3^{509})^6}$ is implemented by 18 multiplications in $\mathbb{F}_{3^{509}}$). The final exponentiation again does not incur a major computation overhead in Algorithm 2.

3 Horizontal and Vertical Vectorization

Many modern CPUs, even in desktop machines, support a set of data-parallel instructions operating on SIMD registers. For example, Intel has been releasing SIMD-enabled CPUs since 1999 [21, 30]. As of now, most vendors provide support for 128-bit SIMD registers and parallel operations on 8-, 16-, 32- and 64-bit data. Recently, CPUs with 256-bit SIMD registers are also available. We work with Intel's SSE2 (128-bit) and AVX2 (256-bit) registers. Since we use 64-bit words for packing of data, using these SIMD intrinsics can lead to speedup of nearly

two or four. In practice, we expect less speedup for various reasons. First, all steps in a computation do not possess inherent data parallelism. Second, the input and output values are usually available in chunks of machine words which are 32 or 64 bits in size. Before the use of an SIMD instruction, one needs to pack data stored in normal registers or memory locations to SIMD registers. Likewise, after using an SIMD instruction, one needs to unpack the content of an SIMD register back to normal registers or memory locations. Frequent conversion of data between scalar and vector forms may be costly. Finally, if the algorithm is memory-intensive, SIMD features do not help much.

We use SIMD-based vectorization techniques for the computation of eta pairing. These vectorization techniques provide speedup by reducing the overheads due to packing and unpacking. We study two common SIMD-based vectorization techniques called horizontal and vertical vectorization. Though vertical vectorization is capable of reducing data-conversion overheads substantially, it encounters an increased memory overhead in terms of cache misses. Experimental results of eta pairing computation over fields of characteristics two and three validate the claim that vertical vectorization achieves better performance gains compared to horizontal vectorization.

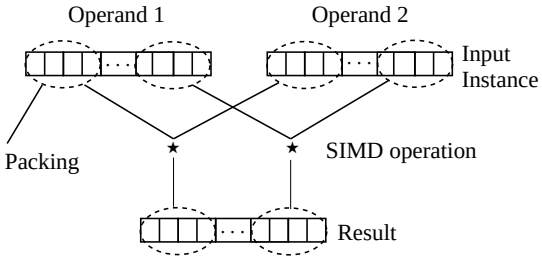
3.1 Horizontal Vectorization

Figure 1 explains the working of horizontal vectorization using AVX2 (256-bit) registers. One single operation $\star$ between two multi-word operands is to be performed. Four 64-bit machine words of individual operands are first packed into SIMD registers, and one SIMD instruction for $\star$ is used to compute the output in a single SIMD register. The result stored in the output SIMD registers can further be used in remaining computations.

As an example, consider operands a and b each stored in an array of twenty 64-bit words. Suppose that we need to compute the bit-wise XOR of a and b, and store the result in c. A usual 64-bit implementation calls for twenty invocations of the CPU instruction for XOR. AVX2-based XOR handles 256 bits of the operands in one CPU instruction, and finishes after only five invocations of this instruction. The output array c of SIMD registers is available in the packed format required in future data-parallel operations in which c is an input.

There are, however, situations where horizontal vectorization requires unpacking of data after a CPU instruction. Consider the unary left-shift operation on an array a of twenty 64-bit words. Let us index the words of a as $a_1, a_2, \ldots, a_{20}$. The words $a_{4i-3}, a_{4i-2}, a_{4i-1}, a_{4i}$ are packed into an SIMD register R_i. Currently, SIMD intrinsics do not provide facilities for shifting R_i as a 256-bit value by any amount (except in multiples of eight). What we instead obtain in the output SIMD register is a 256-bit value in which all the 64-bit components are individually left-shifted. The void created in the shifted version of a_{4i-k} needs to be filled by the most significant bits of the pre-shift value of a_{4i-k+1}. More frustratingly, the void created in a_{4i} by the shift needs to be filled by the most significant bits of the pre-shift value of a_{4i+1} which is a 64-bit member of a separate SIMD register R_{i+1}. The other 64-bit members in R_{i+1} must

Fig. 1. Horizontal Vectorization



not interfere with the shifted value of R_i. Masking out these members from R_{i+1} eats up a clock cycle. To sum up, horizontal vectorization may result in frequent scalar-to-vector and vector-to-scalar conversions, and suffer from packing and unpacking overheads.

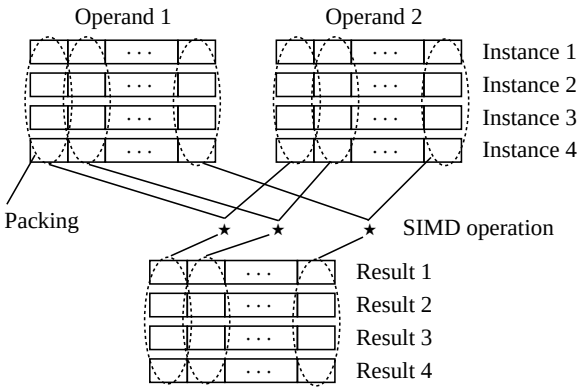
3.2 Vertical Vectorization

Vertical vectorization using AVX2 (256-bit) registers and 64-bit words works as shown in Figure 2. Four instances of the same operation are carried out on two different sets of data. Data of matching operands from the four instances are packed into SIMD registers, and the same sequence of operations is performed on these registers using SIMD intrinsics. Each one-fourth of an SIMD register pertains to one of the instances. After an SIMD instruction, each one-fourth of the output SIMD register contains the result for one of the four instances. Thus, data from four separate instances are maintained in 64-bit formats in these SIMD registers throughout a sequence of operations. When the sequence is completed, data from the final output SIMD registers are unpacked into the respective 64-bit storage outputs for the four instances.

The advantage of this vectorization technique is that it adapts naturally to any situation where two identical sequences of operations are performed on four separate sets of data. The algorithm does not need to possess inherent data parallelism. However, the sequence of operations must be identical (or nearly identical) on four different sets of data. Finally, a computation using vertical vectorization does not require data conversion after every SIMD operation in the CPU, that is, potentially excessive packing and unpacking overheads associated with horizontal vectorization are significantly eliminated.

Let us now explain how vertical vectorization gets rid of the unpacking requirement after a left-shift operation. Suppose that four operands a, b, c, d need to be left-shifted individually by the *same* number of bits. The i-th words a_i, b_i, c_i, d_i are packed in an SIMD register R_i. First, a suitably right-shifted version of R_{i+1} is stored in another SIMD register S_{i+1}. After that, R_i is left-shifted by a single SIMD instruction causing all of a_i, b_i, c_i, d_i to be left-shifted individually. This

Fig. 2. Vertical Vectorization



shifted SIMD register is then XOR-ed with the SIMD register S_{i+1}. The individual 64-bit words $a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1}$ are not needed in the unpacked form.

3.3 Vectorization of Eta Pairing

Eta pairing on supersingular curves defined over fields of characteristics two and three can be computed using bit-wise operations only (that is, no arithmetic operations are needed). More precisely, only the XOR, OR, AND, and the left- and right-shift operations on 64-bit words are required. As explained earlier, both horizontal and vertical vectorizations behave gracefully for the XOR, OR and AND operations. On the contrary, shift operations are efficient with vertical vectorization only. Therefore the presence and importance of shift operations largely determine the relative performance of the two vectorization methods. We now study each individual field operation (in $\mathbb{F}_{2^{1223}}$ or $\mathbb{F}_{3^{509}}$) in this respect.

- Addition/Subtraction: Only XOR, OR and AND operations are needed to carry out addition and subtraction of two elements in both types of fields. So both the vectorization models are suitable for these operations.
- Multiplication (without reduction): We use comb-based multiplication algorithms in which both left- and right-shift operations play a crucial role. Consequently, multiplication should be faster for vertical vectorization than horizontal vectorization.
- Square/Cube (without reduction): Since we have used precomputations in eight-bit chunks, byte-level shifts suffice, that is, both models of vectorization are efficient for these operations.
- Modular reduction: Reduction using the chosen irreducible polynomials call for bit-level shift operations, so vertical vectorization is favored.
- Square-root/Cube-root with modular reduction: Extraction of the polynomials a, b (and c for characteristic three), and multiplication by $x^{1/2}$ (or $x^{1/3}$

and $x^{2/3}$) involve several shift operations. So vertical vectorization seems to be the better choice.
- Inverse: The extended Euclidean algorithm is problematic for both horizontal and vertical vectorization models. On the one hand, bit-level shifts impair the performance of horizontal vectorization. On the other hand, the sequence for a gcd calculation depends heavily on the operands, rendering vertical vectorization infeasible to implement. We therefore use only non-SIMD implementations for the inverse operation.

Multiplication (with modular reduction) happens to be the most frequent operation in Algorithms 1 and 2. Vertical vectorization is therefore expected to outperform horizontal vectorization for these algorithms. We present our horizontal and vertical multiplication algorithms as Algorithms 3 and 4. These pertain to the curve over $\mathbb{F}_{2^{1223}}$. Multiplication in $\mathbb{F}_{3^{509}}$ can be similarly handled.

Algorithms 3 and 4 implement comb-based multiplication [29] with four-bit windows. We take 64-bit words and 256-bit SIMD registers (AVX2). We pack four words in an SIMD register. In the algorithms, word variables are denoted by lower-case letters, and SIMD registers by upper-case letters. We use the subscripts 64 and 256 to differentiate between word-level and SIMD-level operations. For example, $a[i][j] \ll_{64} 41$ indicates 41-bit left-shift of the word $a[i][j]$. Likewise, $T_1 \oplus_{256} T_2$ stands for an AVX2 XOR operation. Switching between word-based and SIMD representations is denoted by pack and unpack. Comb-based multiplication has three stages: precomputation, intermediate product computation, and reduction. The reduction stage is shown separately as Algorithm 5.

An element of $\mathbb{F}_{2^{1223}}$ is stored in twenty 64-bit words. The inputs of horizontal vectorization are two arrays A, B of five 256-bit values. The intermediate product is output as an array c of forty 64-bit words (actually, 39 words suffice). Precomputations are done on the second input B. A 16×20 table t is prepared in this stage. During intermediate product computation, we use 256-bit XOR, but both the operands of this operation are packed and the result is unpacked, inside the loop. Finally, the shift-intensive reduction of c (see Algorithm 5) would proceed at the 64-bit word level. As a consequence, horizontal vectorization is expected to show poor performance.

In vertical vectorization, four pairs of inputs are packed in two arrays, each consisting of 20 SIMD registers. The intermediate products are output as an array of 40 SIMD registers. Since four pairing computations now proceed in parallel, the precomputed table t is now a 64×20 array of 64-bit words. During intermediate product computation, entries from t are packed in an SIMD register P and XOR-ed with an appropriate register in the output array C. This entry in C does not need to be packed inside the loop. Likewise, after the XOR operation, there is no need to unpack the entry in C inside the loop. We can pass the packed intermediate product C straightaway to the reduction function which can now operate on SIMD registers. This is how vertical vectorization shows the promise of improved performance. It would be nice if we could additionally avoid the packing of entries of t in P. But since the indices in t of the words to be packed depend very much on the four inputs in A, this overhead seems unavoidable.

Algorithm 3 Horizontal Vectorization of Multiplication in $\mathbb{F}_{2^{1223}}$

Input: $A[0,\ldots,4], B[0,\ldots,4]$.
Output: $c[0,\ldots,39]$.

Initialize all the words of c to 0, $U \leftarrow \mathrm{pack}(0\mathrm{x}F, 0\mathrm{x}F, 0\mathrm{x}F, 0\mathrm{x}F)$, and $v[0] \leftarrow 0$.
for $i = 0$ **to** 4 /* Precomputation loop */ **do**
  $j \leftarrow i \ll_{64} 2$, $k \leftarrow j + 1$, $T \leftarrow B[i]$, $u \leftarrow \mathrm{unpack}(T)$.
  $v[1] \leftarrow u[1] \gg_{64} 61, v[2] \leftarrow u[2] \gg_{64} 61, v[3] \leftarrow u[3] \gg_{64} 61$.
  $V \leftarrow \mathrm{pack}(v[0,1,2,3])$, $t[0][j, j+1, j+2, j+3] \leftarrow (0,0,0,0)$
  $T_1 \leftarrow T$, $t[1][j, j+1, j+2, j+3] \leftarrow \mathrm{unpack}(T_1)$.
  $T_2 \leftarrow (T \ll_{256} 1) \oplus_{256} (V \gg_{256} 2)$, $t[2][j, j+1, j+2, j+3] \leftarrow \mathrm{unpack}(T_2)$.
  $T_3 \leftarrow T \oplus_{256} T_2$, $t[3][j, j+1, j+2, j+3] \leftarrow \mathrm{unpack}(T_3)$.
  $T_5 \leftarrow (T \ll_{256} 2) \oplus_{256} (V \gg_{256} 1)$.
  for $k = 1$ **to** 3 **do**
    if $(k = 1)$ **then** $T_4 \leftarrow T_5$,
    else if $(k = 2)$ **then** $T_4 \leftarrow (T \ll_{256} 3) \oplus_{256} V$, **else** $T_4 \leftarrow T_4 \oplus_{256} T_5$.
    $t[4k][j, j+1, j+2, j+3] \leftarrow \mathrm{unpack}(T_4)$.
    $t[4k+1][j, j+1, j+2, j+3] \leftarrow \mathrm{unpack}(T_4 \oplus_{256} T_1)$.
    $t[4k+2][j, j+1, j+2, j+3] \leftarrow \mathrm{unpack}(T_4 \oplus_{256} T_2)$.
    $t[4k+3][j, j+1, j+2, j+3] \leftarrow \mathrm{unpack}(T_4 \oplus_{256} T_3)$.
  end for
  $v[0] \leftarrow t[0][0] \gg_{64} 61$.
end for
$j \leftarrow 15$.
while $(j \geq 0)$ /* Intermediate product computation loop */ **do**
  $l \leftarrow j \ll_{64} 2$.
  for $i = 0$ **to** 4 **do**
    $T \leftarrow (A[i] \gg_{256} l)\ \mathrm{AND}_{256}\ U$.
    for $k = 0, 4, 8, 12, 16$ **do**
      $T_1 \leftarrow \mathrm{pack}(c[4i+k, 4i+k+1, 4i+k+2, 4i+k+3])$.
      $T_2 \leftarrow \mathrm{pack}(t[u[0]][k], t[u[0]][k+1], t[u[0]][k+2], t[u[0]][k+3])$.
      $T_3 \leftarrow T_1 \oplus_{256} T_2$.
      $c[4i+k, 4i+k+1, 4i+k+2, 4i+k+3]) \leftarrow \mathrm{unpack}(T_3)$.
    end for
    for $l = 1$ **to** 3 **do**
      for $k = 0$ **to** 19 **do**
        $c[4i+k+l] \leftarrow c[4i+k+l] \oplus_{64} t[u[l]][k]$.
      end for
    end for
  end for
  if $(j = 0)$ **then** break.
  $v_1 \leftarrow 0$.
  for $i = 0$ **to** 39 **do**
    $t_1 \leftarrow c[i]$, $v_0 \leftarrow t_1 \gg_{64} 60$, $c[i] \leftarrow (t_1 \ll_{64} 4) \oplus_{64} v_1$, $v_1 \leftarrow v_0$.
  end for
  $j \leftarrow j - 1$.
end while

Algorithm 4 Vertical Vectorization of Multiplication in $\mathbb{F}_{2^{1223}}$

Input: $A[0,\ldots,19], B[0,\ldots,19]$
Output: $C[0,\ldots,39]$

$L_1 \leftarrow \mathrm{pack}(0\mathrm{x}1, 0\mathrm{x}1, 0\mathrm{x}1, 0\mathrm{x}1)$, $L_2 \leftarrow \mathrm{pack}(0\mathrm{x}3, 0\mathrm{x}3, 0\mathrm{x}3, 0\mathrm{x}3)$,
$L_3 \leftarrow \mathrm{pack}(0\mathrm{x}7, 0\mathrm{x}7, 0\mathrm{x}7, 0\mathrm{x}7)$. $V_0 \leftarrow \mathrm{pack}(0, 0, 0, 0)$.
$T_{63} \leftarrow \mathrm{pack}(0\mathrm{x}F^{15}E, 0\mathrm{x}F^{15}E, 0\mathrm{x}F^{15}E, 0\mathrm{x}F^{15}E)$,
$T_{62} \leftarrow \mathrm{pack}(0\mathrm{x}F^{15}C, 0\mathrm{x}F^{15}C, 0\mathrm{x}F^{15}C, 0\mathrm{x}F^{15}C)$,
$T_{61} \leftarrow \mathrm{pack}(0\mathrm{x}F^{15}8, 0\mathrm{x}F^{15}8, 0\mathrm{x}F^{15}8, 0\mathrm{x}F^{15}8)$.
for $i = 0$ **to** 19 /* Precomputation loop */ **do**
  $T_1 \leftarrow ((V_0 \gg_{256} 2)\ \mathrm{AND}_{256}\ L_1)\ \mathrm{OR}_{256}\ ((B[i] \ll_{256} 1)\ \mathrm{AND}_{256}\ T_{63})$,
  $T_2 \leftarrow ((V_0 \gg_{256} 1)\ \mathrm{AND}_{256}\ L_2)\ \mathrm{OR}_{256}\ ((B[i] \ll_{256} 2)\ \mathrm{AND}_{256}\ T_{62})$,
  $T_3 \leftarrow (V_0)\ \mathrm{OR}_{256}\ ((B[i] \ll_{256} 3)\ \mathrm{AND}_{256}\ T_{61})$.
  $S_1 \leftarrow B[i] \oplus_{256} T_1$, $S_2 \leftarrow T_2 \oplus_{256} T_3$.
  $t_0[0,1,2,3] \leftarrow \mathrm{unpack}(B[i])$, $t_1[0,1,2,3] \leftarrow \mathrm{unpack}(T_1)$,
  $t_2[0,1,2,3] \leftarrow \mathrm{unpack}(T_2)$, $t_3[0,1,2,3] \leftarrow \mathrm{unpack}(T_3)$.
  $s_1[0,1,2,3] \leftarrow \mathrm{unpack}(S_1)$, $s_2[0,1,2,3] \leftarrow \mathrm{unpack}(S_2)$.
  for $j = 0$ **to** 3 **do**
    $t[16j][i] \leftarrow 0$, $t[16j+1][i] \leftarrow t_0[j]$, $t[16j+2][i] \leftarrow t_1[j]$, $t[16j+3][i] \leftarrow s_1[j]$,
    $t[16j+4][i] \leftarrow t_2[j]$, $t[16j+5][i] \leftarrow t_2[j] \oplus_{64} t_0[j]$, $t[16j+6][i] \leftarrow t_2[j] \oplus_{64} t_1[j]$,
    $t[16j+7][i] \leftarrow t_2[j] \oplus_{64} s_1[j]$, $t[16j+8][i] \leftarrow t_3[j]$, $t[16j+9][i] \leftarrow t_3[j] \oplus_{64} t_0[j]$,
    $t[16j+10][i] \leftarrow t_3[j] \oplus_{64} t_1[j]$, $t[16j+11][i] \leftarrow t_3[j] \oplus_{64} s_1[j]$,
    $t[16j+12][i] \leftarrow s_2[j]$, $t[16j+13][i] \leftarrow s_2[j] \oplus_{64} t_0[j]$,
    $t[16j+14][i] \leftarrow s_2[j] \oplus_{64} t_1[j]$, $t[16j+15][i] \leftarrow s_2[j] \oplus_{64} s_1[j]$.
  end for
end for
Initialize C to zero, $U \leftarrow \mathrm{pack}(0\mathrm{x}F, 0\mathrm{x}F, 0\mathrm{x}F, 0\mathrm{x}F)$, and $j \leftarrow 15$.
while $(j \geq 0)$ /* Intermediate product computation loop */ **do**
  $l \leftarrow j \ll_{64} 2$
  for $i = 0$ **to** 19 **do**
    $U \leftarrow (A[i] \gg_{256} l)\ \mathrm{AND}_{256}\ U$, $(u_1, u_2, u_3, u_4) \leftarrow \mathrm{unpack}(U)$, $ival \leftarrow i$, $k \leftarrow 0$.
    while $k < 20$ **do**
      $P \leftarrow \mathrm{pack}(t[u_4][k], t[u_3][k], t[u_2][k], t[u_1][k])$,
      $C[ival] \leftarrow C[ival] \oplus_{256} P$.
      $P \leftarrow \mathrm{pack}(t[u_4][k+1], t[u_3][k+1], t[u_2][k+1], t[u_1][k+1])$,
      $C[ival+1] \leftarrow C[ival+1] \oplus_{256} P$.
      $P \leftarrow \mathrm{pack}(t[u_4][k+2], t[u_3][k+2], t[u_2][k+2], t[u_1][k+2])$,
      $C[ival+2] \leftarrow C[ival+2] \oplus_{256} P$.
      $P \leftarrow \mathrm{pack}(t[u_4][k+3], t[u_3][k+3], t[u_2][k+3], t[u_1][k+3])$,
      $C[ival+3] \leftarrow C[ival+3] \oplus_{256} P$.
      $k \leftarrow k+4$, $ival \leftarrow ival+4$.
    end while
  end for
  if $(j = 0)$ break.
  Initialize V_0 to zero.
  for $i = 0$ **to** 39 **do**
    $V_1 \leftarrow C[i] \gg_{256} 60$, $C[i] \leftarrow V_0 \oplus_{256} (C[i] \ll_{256} 4)$, $V_0 \leftarrow V_1$.
  end for
  $j \leftarrow j - 1$.
end while

Algorithm 5 Reduction of the Intermediate Product

Input: The intermediate product $\gamma[0 \dots 38]$
Output: The reduced product stored in $\gamma[0 \dots 19]$

for $i = 38$ **down to** 20 **do**
$\alpha \leftarrow \gamma[i]$, $\gamma[i-20] \leftarrow \gamma[i-20] \oplus (\alpha \ll 57)$, $\gamma[i-19] \leftarrow \gamma[i-19] \oplus (\alpha \gg 7)$,
$\gamma[i-16] \leftarrow \gamma[i-16] \oplus (\alpha \ll 56)$, $\gamma[i-15] \leftarrow \gamma[i-15] \oplus (\alpha \gg 8)$.
end for
$\alpha \leftarrow \gamma[19] \gg 7$, $\gamma[0] \leftarrow \gamma[0] \oplus \alpha$, $\gamma[3] \leftarrow \gamma[3] \oplus (\alpha \ll 63)$,
$\gamma[4] \leftarrow \gamma[4] \oplus (\alpha \gg 1)$, $\gamma[19] \leftarrow \gamma[19]$ AND δ.

Algorithm 5 shows the reduction of the intermediate product for both models of vectorization. In horizontal vectorization, γ is the array c of 64-bit words, α a 64-bit variable, $\delta = 0\text{x}7F$, and all operations are on 64-bit words. In vertical vectorization, γ is the array C of 256-bit SIMD registers, α a 256-bit variable, $\delta = \text{pack}(0\text{x}7F, 0\text{x}7F, 0\text{x}7F, 0\text{x}7F)$, and all operations are on SIMD registers.

4 Experimental Results

We have carried out our experiments on an Intel Corei7-4770S platform (CPU clock 3.10 GHz) running the 64-bit Ubuntu operating system version 13.10. The programs are compiled by version 4.8.1 of the gcc compiler with the -O3 optimization flag. In some of our experiments, the widely available SSE2 (Streaming SIMD Extension) intrinsics are used [21, 30]. SSE2 uses 128-bit SIMD registers, so we can pack two 64-bit words in a single SIMD register. The Sandy Bridge architecture released by Intel in 2011 introduces 256-bit SIMD registers AVX (Advanced Vector Extension). AVX supports 256-bit floating-point vector operations only. The Haswell architecture released in 2013 introduces another extension AVX2 which supports 256-bit integer vector operations. Our machine supports all these SIMD features. In addition to SSE2, we have also worked with AVX2 intrinsics [21]. With AVX2, we can pack four 64-bit words in a register and hope to exploit data parallelism more than what can be achieved with SSE2.

The timing results are reported in clock cycles. For non-SIMD and horizontal-SIMD implementations, the timings correspond to the execution of one field operation or one eta-pairing computation. For the vertical-SIMD implementation, two (for SSE2) or four (for AVX2) operations are performed in parallel. The times obtained by our implementation are divided by two or four in the tables below in order to indicate the average time per operation. This is done to make the results directly comparable with the results from the non-SIMD and horizontal-SIMD implementations. We use gprof and valgrind to profile our program. Special cares are adopted to minimize cache misses [14].

Tables 1 and 2 summarize the average computation times of basic field operations in $\mathbb{F}_{2^{1223}}$ and $\mathbb{F}_{3^{509}}$. For the addition and multiplication operations, SIMD-based implementations usually perform better than the non-SIMD implementation. For the square, square-root, cube and cube-root operations, the

Table 1. Timing for field operations in $\mathbb{F}_{2^{1223}}$ (clock cycles)

Mode	Addition	Multiplication*	Square*	Square root*
Non-SIMD	52.5	10546.2	279.7	2806.9
SSE2 (H)	27.2	11780.3	415.0	2565.1
SSE2 (V)	27.5	8192.6	285.5	1663.7
AVX2 (H)	10.3	5525.1	276.9	3140.5
AVX2 (V)	6.6	5478.9	244.4	853.7
Hankerson et al. [19]		8200	600	500
Beuchat et al. [10]		5438.4	480	748.8
Aranha et al. [5]		4030	160	166

*Including modular reduction

Table 2. Timing for field operations in $\mathbb{F}_{3^{509}}$ (clock cycles)

Mode	Addition	Multiplication*	Cube*	Cube root*
Non-SIMD	132.3	14928.4	773.9	4311.9
SSE2 (H)	64.1	11025.2	835.4	5710.6
SSE2 (V)	54.4	6494.6	442.8	1953.3
Hankerson et al. [19]		7700	900	1200
Beuchat et al. [10]		4128	900	974.4

*Including modular reduction

Table 3. Times for computing one eta pairing (in millions of clock cycles)

Implementation	Characteristic	Time	Speedup
Non-SIMD	2	40.6	
	3	53.7	
SSE2 (H)	2	41.5	–2.2%
	3	37.7	29.8%
SSE2 (V)	2	29.5	27.3%
	3	24.9	53.6%
AVX2 (H)	2	29.1	28.3%
AVX2 (V)	2	27.2	33.0%
Hankerson et al. [19]	2	39	
	3	33	
Beuchat et al. [10]	2	26.86	
	3	22.01	
Aranha et al. [5]	2	18.76	

performance of the horizontal implementation is often poorer than that of the non-SIMD implementation, whereas the performance of the vertical implementation is noticeably better than that of the non-SIMD implementation. The experimental results tally with our theoretical observations discussed in Section 3.3. That is, field operations involving bit-level shifts significantly benefit from the vertical model of vectorization. In particular, with SSE2, the time of each multiplication operation can be reduced by up to 25% using horizontal vectorization. For vertical vectorization, this reduction is in the range 25–50%. AVX2 can produce an additional speedup of 30–50% over SSE2.

In Table 3, we mention the average times for computing one eta pairing for non-SIMD, horizontal-SIMD and vertical-SIMD implementations. The speedup figures tabulated are with respect to the non-SIMD implementation. Vertical vectorization is seen to significantly outperform both non-SIMD and horizontal-SIMD implementations. Once again, we obtain noticeably higher benefits if we use AVX2 in place of SSE2.

In Tables 1–3, we also mention other reported implementation results on finite-field arithmetic and eta-pairing computation. Hankerson et al. [19] use only SIMD features, and our implementations are already faster than their implementations in both characteristics two and three. Our implementations are, however, slower than the implementations reported in the other two papers [5, 10]. In fact, these two papers employ other parallelization techniques (multi-threading in multi-core machines). SIMD-based parallelization is not incompatible with multi-core implementations. Indeed, these two parallelization techniques can go hand in hand, that is, SIMD techniques may provide additional speedup in the computation of every individual core. The scope of our work is to compare the performances of horizontal and vertical vectorization techniques in the context of eta pairing over finite fields of small characteristics. To this end, our experimental results, although slower than the best reported implementations, appear to have served our objectives.

5 Conclusion

In this paper, we establish the superiority of the vertical model of SIMD vectorization over the horizontal model for eta-pairing computations over finite fields of small characteristics. Some possible extensions of our work are stated now.

- We have studied vectorization for bit-wise operations only. It is unclear how the two models compare when arithmetic operations are involved. Eta pairing on elliptic curves defined over prime fields heavily use multiple-precision integer arithmetic. These form a class of curves still immune to the recent attacks [1, 2, 6, 24]. Other types of pairing and other cryptographic primitives also require integer arithmetic. Managing carries and borrows during addition and subtraction stands in the way of effective vectorization. Multiplication poses a more potent threat to data-parallelism ideas.
- Porting our implementations to curves over larger fields of small characteristics (in order to achieve 128-bit security) is important.

– In near future, Intel is going to release the Broadwell architecture which is promised to feature 512-bit SIMD registers (AVX-512) [22]. These registers should produce some additional speedup in eta-paring computations.
– It needs experimentation to understand the extent to which multi-threaded implementations of [5, 10] additionally benefit from the application of SIMD-based vectorization techniques.

References

1. Adj, G., Menezes, A., Oliveira, T., Rodríguez-Henríquez, F.: Weakness of $\mathbb{F}_{3^{6\cdot 1429}}$ and $\mathbb{F}_{2^{4\cdot 3041}}$ for discrete logarithm cryptography. In: IACR Eprint Archive (2013), http://eprint.iacr.org/2013/737
2. Adj, G., Menezes, A., Oliveira, T., Rodríguez-Henríquez, F.: Weakness of $\mathbb{F}_{3^{6\cdot 509}}$ for discrete logarithm cryptography. In: IACR Eprint Archive (2013), http://eprint.iacr.org/2013/446
3. Ahmadi, O., Hankerson, D., Menezes, A.: Software implementation of arithmetic in $\mathbb{F}_{3^m}$. In: International Workshop on the Arithmetic of Finite Fields (WAIFI 2007). pp. 85–102 (2007)
4. Ahmadi, O., Rodriguez-Henriquez, F.: Low complexity cubing and cube root computation over $\mathbb{F}_{3^m}$ in polynomial basis. IEEE Transactions on Computers 59, 1297–1308 (2010)
5. Aranha, D.F., López, J., Hankerson, D.: High-speed parallel software implementation of the η_T pairing. In: CT-RSA 2010. pp. 89–105 (2010)
6. Barbulescu, R., Gaudry, P., Joux, A., Thomé, E.: A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In: IACR Eprint Archive (2013), http://eprint.iacr.org/2013/400
7. Barreto, P.S.L.M.: A note on efficient computation of cube roots in characteristic 3. In: IACR Eprint Archive (2004), http://eprint.iacr.org/2004/305
8. Barreto, P.S.L.M., Galbraith, S.D., OÉigeartaigh, C., Scott, M.: Efficient pairing computation on supersingular abelian varieties. Designs, Codes and Cryptography 42(3), 239–271 (2007)
9. Barreto, P.S.L.M., Kim, H.Y., Lynn, B., Scott, M.: Efficient algorithms for pairing-based cryptosystems. In: CRYPTO 2002. pp. 354–368 (2002)
10. Beuchat, J.L., López-Trejo, E., Martínez-Ramos, L., Mitsunari, S., Rodrguez-Henríquez, F.: Multi-core implementation of the Tate pairing over supersingular elliptic curves. In: Cryptology and Network Security. pp. 413–432 (2009)
11. Boneh, D., Franklin, M.K.: Identity-based encryption from the Weil pairing. In: CRYPTO 2001. pp. 213–229 (2001)
12. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. Journal of Cryptology 17, 297–319 (2004)
13. Bose, U., Bhattacharya, A.K., Das, A.: GPU-based implementation of 128-bit secure eta pairing over a binary field. In: Africacrypt 2013. pp. 26–42. LNCS (2013)
14. Drepper, U.: What every programmer should know about memory (2007), http://lwn.net/Articles/250967/
15. Freeman, D., Scott, M., Teske, E.: A taxonomy of pairing-friendly elliptic curves. Journal of Cryptology 23, 224–280 (2010)
16. Gorla, E., Puttmann, C., Shokrollahi, J.: Explicit formulas for efficient multiplication in $\mathbb{F}_{3^{6m}}$. In: SAC. pp. 173–183 (2007), http://portal.acm.org/citation.cfm?id=1784881.1784893

17. Grabher, P., Großschädl, J., Page, D.: On software parallel implementation of cryptographic pairings. In: SAC. pp. 35–50 (2008)
18. Granger, R., Page, D., Stam, M.: Hardware and software normal basis arithmetic for pairing-based cryptography in characteristic three. IEEE Trans. Computers 54(7), 852–860 (2005)
19. Hankerson, D., Menezes, A., Scott, M.: Software Implementation of Pairings. In: Identity Based Cryptography, pp. 188–206. IOS Press (2008)
20. Hess, F., Smart, N.P., Vercauteren, F.: The eta pairing revisited. IEEE Transactions on Information Theory 52(10), 4595–4602 (2006)
21. Intel: Intel® C++ compiler XE 13.1 user and reference guide: Compiler reference: Intrinsics (2013), http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-712779D8-D085-4464-9662-B630681F16F1.htm
22. Intel: Intel instruction set architecture extensions (2014), http://software.intel.com/en-us/intel-isa-extensions
23. Joux, A.: A one round protocol for tripartite Diffie-Hellman. Journal of Cryptology 17, 263–276 (2004)
24. Joux, A.: Faster index calculus for the medium prime case application to 1175-bit and 1425-bit finite fields. In: Eurocrypt 2013. pp. 177–193. LNCS (2013)
25. Katoh, Y., Huang, Y.J., Cheng, C.M., Takagi, T.: Efficient implementation of the η_T pairing on GPU. In: IACR Eprint Archive (2011), http://eprint.iacr.org/2011/540
26. Kawahara, Y., Aoki, K., Takagi, T.: Faster implementation of η_T pairing over $GF(3^m)$ using minimum number of logical instructions for GF(3)-addition. In: Pairing. pp. 282–296 (2008)
27. Kerins, T., Marnane, W.P., Popovici, E.M., Barreto, P.S.L.M., Brazil, S.P.: Efficient hardware for the Tate pairing calculation in characteristic three. In: CHES. pp. 412–426 (2005)
28. Lee, E., Lee, H.S., Park, C.M.: Efficient and generalized pairing computation on abelian varieties. IEEE Transactions on Information Theory 55, 1793–1803 (2009)
29. López, J., Dahab, R.: High speed software implementation in $\mathbb{F}_{2^m}$. In: Indocrypt 2000. pp. 93–102. LNCS (2000)
30. Microsoft: MMX, SSE, and SSE2 Intrinsics (2010), http://msdn.microsoft.com/en-us/library/y0dh78ez(v=vs.90).aspx
31. Miller, V.: The Weil pairing and its efficient calculation. Journal of Cryptology 17, 235–261 (2004)
32. Montgomery, P.L.: Vectorization of the elliptic curve method. ACM (1991)
33. Page, D., Smart, N.P.: Parallel cryptographic arithmetic using a redundant Montgomery representation. IEEE Transactions on Computers 53, 1474–1482 (2004)
34. Scott, M.: Optimal irreducible polynomials for GF(2^m) arithmetic. In: IACR Eprint Archive (2007), http://eprint.iacr.org/2007/192
35. Scott, M., Benger, N., Charlemagne, M., Perez, L.J.D., Kachisa, E.J.: On the final exponentiation for calculating pairings on ordinary elliptic curves. In: Pairing-Based Cryptography – Pairing 2009. pp. 78–88. LNCS (2009)
36. Smart, N.P., Harrison, K., Page, D.: Software implementation of finite fields of characteristic three. LMS Journal Computation and Mathematics 5, 181–193 (2002)
37. Takahashi, G., Hoshino, F., Kobayashi, T.: Efficient GF(3^m) multiplication algorithm for η_T pairing. In: IACR Eprint Archive (2007), http://eprint.iacr.org/2007/463
38. Vercauteren, F.: Optimal pairings. IEEE Transactions on Information Theory 56, 455–461 (2010)