On Software Parallel Implementation of Cryptographic Pairings ⋆ ⋆⋆

P. Grabher, J. Großschädl and D. Page

Abstract. A significant amount of research has focused on methods to improve the efficiency of cryptographic pairings; in part this work is motivated by the wide range of applications for such primitives. Although numerous hardware accelerators for pairing evaluation have used parallelism within extension field arithmetic to improve efficiency, similar techniques have not been examined in software thus far. In this paper we focus on parallelism within one pairing evaluation (intra-pairing), and parallelism between different pairing evaluations (inter-pairing). We identify several methods for exploiting such parallelism (extending previous results in the context of ECC) and show that it is possible to accelerate pairing evaluation by a significant factor in comparison to a naive approach.

1 Introduction

Generally speaking, one uses the term cryptographic pairing to describe a non-degenerate bilinear map of the form

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T.$$

In this paper we focus on the Ate pairing which takes the concrete form

$$e : E(\mathbb{F}_p) \times \overline{E}(\mathbb{F}_{p^{k/2}}) \longrightarrow \mathbb{F}^{\times}_{p^k}$$

where $\overline{E}$ is the quadratic twist of an elliptic curve E defined over $\mathbb{F}_{p^{k/2}}$. The type and volume of applications enabled by pairings of this form has dictated that methods for their evaluation remain an ongoing research challenge. This is magnified by the fact that said applications have permeated both high-performance and embedded contexts: computational efficiency and storage footprint are both important. Improvements to high-level algorithms that relate to the pairing itself are clearly the most significant in terms of efficiency; for an overview of the evolution of this topic, see the excellent description by Scott [38]. In short, improvement of seminal but unpublished work by Miller [33] resulted in the first practical algorithms for evaluation of the Tate pairing [5, 19]. These results were further optimised by Duursma and Lee [15] who developed an inexpensive, closed form for specific parameterisations later improved by Kwon [29]. Their techniques were generalised and extended to produce the Eta [4] and Ate [23] pairings, currently considered the fastest means of evaluation.

However, as well as the pairing itself, one depends on lower-level algorithms for arithmetic in the fields $\mathbb{F}_p$, $\mathbb{F}_{p^{k/2}}$ and $\mathbb{F}_{p^k}$. Previous results have reported on analysis and efficient

⋆ The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT. The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

⋆⋆ The work described in this paper has been supported in part by EPSRC grant EP/E001556/1.

realisation of said algorithms; see for example [27, 20, 13]. One can readily identify two types of parallelism within these algorithms and within pairing based cryptosystems more generally: that within a single pairing evaluation (intra-pairing) or between several pairing evaluations (inter-pairing). Put more simply, in the first case the aim is to compute $R = e(P, Q)$ for some P and Q from the appropriate groups; our focus is on parallelism within algorithms for the pairing and constituent arithmetic. Efficient implementation of pairings in hardware have used this feature to great effect; see [26] for an example design where extension field arithmetic is realised using several parallel computational units to reduce latency. In the second case, the aim is to compute all n pairings $R_i = e(P_i, Q_i)$ for $0 \leq i < n$; our focus in on the fact that each R_i can be computed independently. Although Granger and Smart [21] describe a method to improve performance where the pairings form terms in a larger product, i.e. $R = \prod_{i=0}^{n-1} e(P_i, Q_i)$, actually capitalising on the parallelism between disjoint pairings is less well examined. This is despite the fact that numerous instances exist, verification of BLS signatures [9] to name one, where this could be useful.

Identifying parallelism in algorithms for the pairing and constituent arithmetic is only the first step: in order to exploit said parallelism, one must have effective methods to map an algorithm onto the capabilities of a given host platform. Often this mapping is difficult enough that any perceived advantage offered by parallelism is eliminated by implementation overhead, in other cases the correct choice of technique is limited by issues such as parameterisation and use of the pairing in real applications. Our goal in this paper is to focus on parallelism, realised using software techniques, as a means of optimising concrete implementations of the Ate pairing. We organise the paper as follows. In Section 2 we recap on the Ate pairing and standard methods for parameterisation and evaluation. Then, in Section 3, we make a detailed study of parallelism within algorithms for the pairing and constituent arithmetic. Section 4 describes details of our implementation including an efficient algorithm for parallel multiplication in $\mathbb{F}_p$. Using the identified techniques we present and analyse experimental results derived from their implementation on Intel Core2 and Pentium 4 processors; this is captured in Section 5. Finally, we summarise our findings and conclude in Section 6.

2 The Ate Pairing

To recap, the Ate pairing is a bilinear map of the form

$$e : E(\mathbb{F}_p) \times \overline{E}(\mathbb{F}_{p^{k/2}}) \longrightarrow \mathbb{F}_{p^k}^{\times}.$$

Successful parameterisation requires an elliptic curve $E(\mathbb{F}_p)$ whose order n is divisible by some large prime r. Let k, the embedding degree of the curve, be the smallest positive integer such that $r \mid p^k - 1$. A Barreto-Naehrig curve or BN-curve [6] of the form

$$E(\mathbb{F}_p) : y^2 = x^3 + b$$

where $b \neq 0$, satisfies these requirements. In particular, such a curve has prime order, i.e. $r = n$, and embedding degree $k = 12$. Additionally, the trace, curve order and characteristic

Algorithm 1: An algorithm to compute the Ate pairing.

Input : $Q \in E(\mathbb{F}_p)$, $P \in E(\mathbb{F}_{p^2})$, $s = t - 1 \in \mathbb{Z}$.
Output: $e(Q, P)$.

$T \leftarrow P$
$f \leftarrow 1$
for $i = |s| - 2$ **downto** 0 **do**
    $f \leftarrow f^2 \cdot l_{T,T}(Q)$
    $T \leftarrow 2 \cdot T$
    if $s_i = 1$ **then**
        $f \leftarrow f \cdot l_{T,P}(Q)$
        $T \leftarrow T + P$
return $f^{(p^k-1)/n}$

of $\mathbb{F}_p$ can be parameterised by x as follows

$$\begin{aligned} t(x) &= 6x^2 + 1 \\ n(x) &= 36x^4 - 36x^3 + 18x^2 - 6x + 1 \\ p(x) &= 36x^4 - 36x^3 + 24x^2 - 6x + 1. \end{aligned}$$

We closely follow the excellent description of Devegili et al. [14] who show that by selecting $x = -6917529027641089837$ for example, one specifies a 256-bit value p and associated curve where n is of low Hamming weight. Selecting such an x makes the notation $t(x)$, for example, extraneous; using this specific value of x we simply write t instead. Since the associated p satisfies various congruences, it enables an efficient construction of extension field arithmetic using the tower

$$\begin{aligned} \mathbb{F}_{p^2} &= \mathbb{F}_p[X]/(X^2 - \beta) \\ \mathbb{F}_{p^6} &= \mathbb{F}_{p^2}[Y]/(Y^3 - \xi) \\ \mathbb{F}_{p^{12}} &= \mathbb{F}_{p^6}[Z]/(Z^2 - \xi\prime) \end{aligned}$$

where $\beta = -2 \in F_p$, $\xi = -1 - \sqrt{\beta} \in F_{p^2}$ and $\xi\prime = \sqrt[3]{\xi} \in \mathbb{F}_{p^6}$.

Evaluation of the pairing is achieved using Algorithm 1 where $l_{P,Q}(R)$ denotes the line function between points P and Q evaluated at R. The selection of a sparse x allows for efficient realisation of the final exponentiation by $(p^k - 1)/n$ as described fully by Devegili et al. [14].

3 Exploitation of Parallelism

SIMD and SWAR. Many commodity processors now support SWAR (SIMD Within a Register), a form of vector processing; exemplar designs include several generations of SSE by Intel, VIS by Sun, 3DNow! by AMD, and AltiVec by Apple, IBM and Motorola. To utilise this feature, one packs say p sub-words, each q bits in size, into a large SWAR vector. Using such vectors one can permit SIMD style vector operations. Let $\overline{x}_i$ denote the i-th sub-word packed into vector $\overline{x} = (\overline{x}_0, \overline{x}_1, \ldots, \overline{x}_{p-1})_q$. Using such a representation, one can compute all p component-wise additions $\overline{r}_i = \overline{x}_i + \overline{y}_i$ with one operation. The choice of p and q which dictate the number and size of sub-words that can be packed into

a fixed vector length depends on the application. Often an instruction set will support a (somewhat) orthogonal set of operations and choices of p and q. This approach has brought significant performance improvements in easily vectorised kernels such as those found in media processing; by making parallelism explicit the processor can maintain a high issue rate and ensure a good trade-off between provision of computational resources and their utilisation.

Use of SWAR style instruction sets have been successful used to accelerate kernels in symmetric cryptography; for example [11, 10, 31, 36, 32]. Although exploiting parallelism within point multiplication in vanilla Elliptic Curve Cryptography (ECC) is possible [2, 25], vectorisation of the public-key cryptography is often more problematic. Consider two n-bit multi-precision integers x and y represented by $l = \lceil n/w \rceil$ machine words where x_i denotes the i-th such w-bit word. Values represented as such are commonly manipulated within cryptosystems such as RSA and ECC. For the sake of clarity, imagine we set $n = 128$ and $q = w = 32$ such that $p = l = 4$. This implies that we can store x and y in one SWAR register each, i.e.

$$(x_0, x_1, x_2, x_3)_{32} \qquad \text{and} \qquad (y_0, y_1, y_2, y_3)_{32}$$

The problem is that to perform multi-precision addition, for example, one must deal with carry from one sub-word into another sub-word within the same vector. That is, say we want to compute $r = x + y$. The addition of x and y is not component-wise: for example, we need to take the carry produced by the primitive addition $x_0 + y_0$ and factor it into $x_1 + y_1$ thereby destroying the component-wise nature of computation and hence the SIMD style parallelism.

The flexibility of ECC parameterisations helps somewhat in resolving this problem. One might view specific field representations such as Residue Number Systems (RNS) and Optimal Extension Fields (OEF) [3] as more suitable for vectorisation; parameterisation and parallel implementation over $\mathbb{F}_{2^n}$ has also been effective [7] since carries are essentially eliminated by the nature of arithmetic. Motivated by application in RSA as well as ECC, there is a similar effort to accelerate arithmetic in $\mathbb{F}_p$ (or more exactly modulo some integer p). Work by Acar [1] and reports by Intel [24] and Apple [12] all investigate the use of SIMD parallelism for implementing multi-precision integer arithmetic. Acar states that his implementation of RSA on a processor with an MMX instruction set runs significantly slower due to a lack of unsigned 16-bit and 32-bit multiplication. Intel are more positive in their results that focus on the SSE2 instruction set. Their method applies a form of recoding into a representation with a smaller digit size; this allows fast combination of partial products without requiring carries at all. Hankerson et al. [22, Chapter 5.1.3] also discuss the same technique within the context of ECC.

This previous work offers a natural way to exploit intra-pairing parallelism: one simply accelerates arithmetic in $\mathbb{F}_p$ which, in turn, accelerates all higher layers of arithmetic and therefore the pairing evaluation itself.

Bit-slicing and Digit-slicing. Considering a scalar processor with a w-bit word size, let x_i denote the i-th bit of a machine word x where i is termed the index of the bit. Such a processor operates natively on word sized operands. For example, with a single operation

one might perform addition of w-bit operands x and y to produce $r = x + y$, or component-wise XOR to produce $r_i = x_i \oplus y_i$ for all $0 \leq i < w$. This ability is restricted however when an algorithm is required to perform some operation involving different bits from the same word. For example one might be required to combine x_i and x_j, where $i \neq j$, using an XOR operation in order to compute the parity of x. In this situation one is required to shift (and potentially mask) the bits so they are aligned at the same index ready for combination via a native, component-wise XOR. The technique of bit-slicing, proposed by Biham for efficient implementation of DES [8], offers a way to reduce the associated overhead. Instead of representing the w-bit value x as one machine word, we represent x using w machine words where word i contains x_i aligned at the same fixed index j. As such, there is no need to align bits ready for use in a component-wise XOR operation. Additionally, since native word oriented logical operations in the processor operate on all w bits in parallel, one can pack w different values (say $x[k]$ for $0 \leq k < w$) into the w words and proceed using an analogy of SIMD style parallelism. Conversion to and from a bit-sliced representation can represent an overhead but this can be amortised if the cost of computation using the bit-sliced values is significant enough: Biham used this technique to extract a five-fold performance improvement from DES using a 64-bit Alpha processor.

Although it overloads the term somewhat, one might describe previous SWAR based implementations of public-key cryptography as digit-serial in the sense that they try to extract parallelism from a series of digits representing one value. An alternative approach, which one might describe as digit-sliced SWAR, represents the digit based analogy of the bit based slicing approach outlined above. This seems to have been first investigated by Montgomery in the context of ECM based factoring [35] and then rediscovered and applied in the context of RSA by Page and Smart [37]. Following the example above, the basic idea is that instead of representing an l-word multi-precision integer x by packing the digits x_i into one SWAR vector, we slice the digits into l separate SWAR vectors where vector i contains x_i aligned at the same fixed index j. For the case where $n = 128$, $q = w = 32$ and $p = l = 4$ we therefore represent x and y using four SWAR registers

$$\begin{array}{lcl} (x_0, \cdot, \cdot, \cdot)_{32} & & (y_0, \cdot, \cdot, \cdot)_{32} \\ (x_1, \cdot, \cdot, \cdot)_{32} & \text{and} & (y_1, \cdot, \cdot, \cdot)_{32} \\ (x_2, \cdot, \cdot, \cdot)_{32} & & (y_2, \cdot, \cdot, \cdot)_{32} \\ (x_3, \cdot, \cdot, \cdot)_{32} & & (y_3, \cdot, \cdot, \cdot)_{32} \end{array}$$

where $\cdot$ denotes some arbitrary padding. The premise is that this makes carry easier to deal with: we are now faced with carries between sub-words of different vectors which are aligned at the same index rather than carries between sub-words in the same vector. As such and in a naive sense, one expects the amount of sub-word reorganisation, which represents a significant computational overhead, to be lower. Again there is an overhead in conversion to and from the digit-sliced representation. However, in common with the bit-slicing approach, we can operate on p packed values at the same time by replacing the padding (i.e. $\cdot$) with useful data. This essentially allows us to compute p separate multi-precision additions (say $x[k] + y[k]$ for $0 \leq k < p$), for example, at the same time. We call each such parallel digit-sliced operation a channel and term an implementation c-way digit-sliced if there are c channels utilised.

Algorithm 2: An algorithm to compute the Ate pairing.

Input : $Q \in E(\mathbb{F}_p)$, $P \in E(\mathbb{F}_{p^2})$, $s = t - 1 \in \mathbb{Z}$.
Output: $e(Q, P)$.

for $i = 1$ **upto** $|s| - 1$ **do**
  $\tau_f[i] \leftarrow \tau_f[i-1]^2 \cdot l_{\tau_T[i-1],\tau_T[i-1]}(Q)$
  $\tau_T[i] \leftarrow 2 \cdot \tau_T[i-1]$
$f_0 \leftarrow 1$, $T_0 \leftarrow \mathcal{O}$
$f_1 \leftarrow 1$, $T_1 \leftarrow \mathcal{O}$
par
  for $i = 0$ **upto** $|s| - 1$ **do**
    if $s_i = 1$ **and** $i = 0 \pmod 2$ **then**
      $f_0 \leftarrow f_0 \cdot \tau_f[i] \cdot l_{T_0,\tau_T[i]}(Q)$
      $T_0 \leftarrow T_0 + \tau_T[i]$
  for $i = 0$ **upto** $|s| - 1$ **do**
    if $s_i = 1$ **and** $i = 1 \pmod 2$ **then**
      $f_1 \leftarrow f_1 \cdot \tau_f[i] \cdot l_{T_1,\tau_T[i]}(Q)$
      $T_1 \leftarrow T_1 + \tau_T[i]$
$f \leftarrow f_0 \cdot f_1 \cdot l_{T_0,T_1}(Q)$
return $f^{(p^k-1)/n}$

In terms of the pairing and constituent arithmetic, the technique of digit-slicing is potentially interesting. At any level, all the algorithms for arithmetic are (or are close to) control-flow invariant; for example for any given pairing evaluation using some fixed parameterisation, one performs the same operation at a give step so only the data values differ. As such, one can deploy digit-slicing to exploit intra-pairing parallelism (for example performing c multiplications in $\mathbb{F}_p$ at once to accelerate arithmetic in $\mathbb{F}_{p^2}$), or inter-pairing parallelism (for example evaluating c pairings at once).

Multi-core Processors. A modern trend in the design of microprocessors is that of multi-core, i.e. having say n physical processor cores on a single die. This philosophy is in part guided by the need to make effective use of advances in fabrication which allow dies to house huge number of transistors, and the so-called memory wall which posits that memory access dominates the performance of conventional single-core processors. In software, one can take advantage of multi-core processors using, for example, the OpenMP standard; with suitable compiler and operating system support this enables multiple code sequences to be executed in parallel, one on each core.

The use of multi-core processors is an emerging research topic in the context of cryptographic implementation, for example Fan et al. investigate modular multiplication [16] and ECC [17] on this type of platform. Intra-pairing parallelism is clearly possible at the field arithmetic level as evidenced by related hardware based approaches [26]. In software however, the overhead of thread management is a limiting factor: if the threads are too fine-grained then the cost of their management will dominate useful computation and eliminate the advantage of parallelism. An alternative, therefore, is to consider more course-grained parallelism. In this setting, inter-pairing parallelism is easy to exploit: we simply have each core compute a separate pairing. Exploiting course-grained intra-pairing parallelism requires

more thought. For example, one might redesign Algorithm 1 to allow parallelism between point arithmetic or line function evaluations.

Consider Algorithm 2 which is derived from a specialisation of so-called fixed-base windowing [22, Algorithm 3.41] for $w = 1$. Use of the **par** keyword shows that after a precomputation phase comprised of point (resp. line) doublings, two threads can compute point (resp. line) additions in parallel (one thread deals with odd-indexed bits in s, the other even-indexed bits). The clear advantage of this approach is parallelism; the clear disadvantage is the significant memory overhead for tables τ_f and τ_T, and the fact that the point (resp. line) additions are now projective rather than mixed.

4 Implementation Details

In the following we elaborate on the concrete implementation of the field arithmetic using scalar (i.e. non-SIMD) as well as SIMD (i.e. MMX, SSE) instruction sets. Both implementations have in common that the modular multiplication (resp. squaring) operation is realised via Montgomery reduction [34]. The inversion is performed using the Extended Euclidean Algorithm (EEA).

4.1 Field Arithmetic with the IA32/IA64 Instruction Set

The IA32 architecture provides an add-with-carry instruction (`adc`) and a 32-bit unsigned multiply instruction yielding a 64-bit result (`mul`). Thanks to the availability of these two instructions, the arithmetic operations in $\mathbb{F}_p$ can be implemented in a fairly straightforward way: a field element is simply represented in form of an array of single-precision (i.e. 32-bit) words and the software routines for addition and multiplication loop through these arrays and produce the result using the afore-mentioned instructions. Our implementation of the field arithmetic is written in ANSI C and contains some hand-optimised assembly language sections for the performance-critical inner-loop operations. As the size of the fields used in pairing-based cryptography is relatively small, it is possible to unroll the inner loops and gain some extra performance at the expense of a slight increase in code footprint.

Algorithm 3 shows the Coarsely Integrated Operand Scanning (CIOS) method for calculating the Montgomery product $Z = A \cdot B \cdot 2^{-n} \bmod M$ [28]. The n-bit operands A, B, M are represented by arrays of s single-precision w-bit words. The algorithm has a nested loop structure with two inner loops; the first contributes to the calculation of the product $A \cdot B$ and the second implements the modular reduction operation. Both inner loops perform the same operation: two single-precision words are multiplied together, and then two other words are added to the product. Therefore, each iteration of the inner loop executes a `mul`, two `add`, and two `adc` instructions, respectively.

4.2 Field Arithmetic with the MMX/SSE Instruction Set

In order to accelerate the execution of multimedia kernels, Intel introduced the MMX instruction set in 1997 as a SIMD extension to the IA32 architecture. MMX provides eight 64-bit registers and adds 57 new instructions. Most of these instructions operate on packed

Algorithm 3: Montgomery multiplication (CIOS method).

Input : An s-word modulus $M = (m_{s-1}, \ldots, m_1, m_0)$, two operands $A = (a_{s-1}, \ldots, a_1, a_0)$ and $B = (b_{s-1}, \ldots, b_1, b_0)$ with $A, B < M$, and the constant $m'_0 = -m_0^{-1} \bmod 2^w$.
Output: The Montgomery product $Z = A \cdot B \cdot 2^{-n} \bmod M$.

$Z \leftarrow 0$
for i *from* 0 *by* 1 *to* $s-1$ **do**
  $u \leftarrow 0$
  for j *from* 0 *by* 1 *to* $s-1$ **do**
    $(u, v) \leftarrow a_j \times b_i + z_j + u$
    $z_j \leftarrow v$
  $(u, v) \leftarrow z_s + u$
  $z_s \leftarrow v$
  $z_{s+1} \leftarrow u$
  $q \leftarrow z_0 \times m'_0 \bmod 2^w$
  $(u, v) \leftarrow z_0 + m_0 \times q$
  for j *from* 1 *by* 1 *to* $s-1$ **do**
    $(u, v) \leftarrow m_j \times q + z_j + u$
    $z_{j-1} \leftarrow v$
  $(u, v) \leftarrow z_s + u$
  $z_{s-1} \leftarrow v$
  $z_s \leftarrow z_{s+1} + u$
if $Z \geq M$ **then**
  $Z \leftarrow Z - M$
return $Z = (z_{s-1}, \ldots, z_1, z_0)$

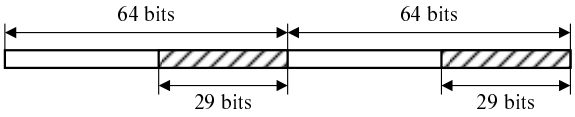

Fig. 1. The packed 29-bit digits within a single 128-bit SSE register, as detailed by [24].

data types, which means that a 64-bit MMX operand can also be treated as either two 32-bit, four 16-bit, or eight 8-bit quantities. The Streaming SIMD Extensions (SSE) further enhance the capabilities of the IA32 architecture through the integration of eight 128-bit registers and appropriate instructions. For example, the SSE2 instruction `pmuludq` allows one to execute two 32×32-bit multiplications independently and in parallel, each yielding a 64-bit result. However, the main drawback of the MMX and SSE instruction sets in the context of multi-precision integer arithmetic is the lack of an add-with-carry instruction.

The fact that neither MMX nor SSE provide an add-with-carry instruction not only makes multiple-precision addition relatively costly, but also defines how multiple-precision multiplication must be implemented in order to exploit SIMD-level parallelism. In [24], Intel recommends that multi-precision integers should be represented as arrays of 29-bit words (instead of the more intuitive representation with 32-bit words) and to pack two such 29-bit words into a 128-bit quantity which can be loaded into SSE registers using the `movdqa` instruction; this is detailed in Figure 1. Two 29×29-bit multiplications can be executed in parallel and several 58-bit products can be accumulated without overflow. More precisely, the 29-bit representation eliminates the need to propagate carry bits from less to more significant words during a multiple-precision multiplication; a single carry propagation

Table 1. Timings for Montgomery multiplication and squaring (in cycles as reported by `rdtsc`) on a Pentium 4 processor for 256-bit, 384-bit and 512-bit operands.

Implementation	256-bit	384-bit	512-bit
SIMD Montgomery Mul.	1182	2104	2978
GMP (`mpn_mul_n` + `redc`)	1171	2429	3700
SIMD Montgomery Sqr.	1063	1875	2523
GMP (`mpn_sqr_n` + `redc`)	1051	2257	3151

must be performed at the very end to obtain the correct result. We implemented the CIOS method for Montgomery multiplication following these guidelines which also allowed us to fuse the two inner loops. This loop fusion does not only reduces the loop overhead, but also eliminates a number of load/store instructions as, for example, the quantity z_j in Algorithm 3 needs to be loaded only once. However, a disadvantage of the multiplication technique described in [24] is that two arrays are necessary for storing the intermediate results during a Montgomery multiplication. This "redundant" representation makes the outer loop of Algorithm 3 relatively costly, in particular the calculation of the quotient q.

Table 1 compares the execution times (in clock cycles) of our Montgomery arithmetic implemented according to Algorithm 3 using the 29-bit representation detailed in [24], and the corresponding functions[1] from the GMP library version 4.2.2. Our implementation is slightly slower for 256-bit operands, but outperforms GMP for 384-bit and 512-bit operands. As mentioned previously, our implementation of Algorithm 3 is characterised by a relatively costly outer loop, while the inner loop is extremely efficient. However, for short operands, the operations in the outer loop dominate the execution time, which renders the 29-bit representation less attractive.

5 Implementation Results

In order to evaluate the options for exploiting parallelism introduced in previous sections, we used two experimental platforms; the rational for their selection was that they represent previous (NetBurst) and current (Core2) generation micro-architectures in commodity microprocessors:

Platform *A* housed a 2.80GHz Intel Pentium 4 processor running a 32-bit installation of Linux including a 2.6.9 series kernel and 32-bit Intel C compiler version 10.1. The SIMD instruction set on this platform was limited to SSE2 series (and earlier) instructions only.

Platform *B* housed a 2.40GHz Intel Core2 Duo processor running a 64-bit installation of Linux including a 2.6.18 series kernel and 64-bit Intel C compiler version 10.1. The SIMD instruction set on this platform was limited to SSE3 series (and earlier) instructions only.

Since our goal is to highlight issues with existing processors, we do not investigate the impact of altering the number of execution pipelines within a particular micro-architecture

[1] Note that GMP features a function for Montgomery reduction (`redc`), but not for Montgomery multiplication. Therefore, a Montgomery multiplication must be composed of `mpn_mul_n` and `redc`. We evaluated the execution times of `mpn_mul_n`, `mpn_sqr_n`, and `redc` with help of the `speed` program.

Table 2. Timings for major operations (in cycles as reported by `rdtsc`) on experimental platform ***A*** (Pentium 4). $\mathbb{F}_p$ is a 256-bit prime field.

	$\mathbb{F}_p$			$\mathbb{F}_{p^{12}}$			$e(P,Q)$
	Inv	Add	Mul	Inv	Add	Mul	
A	278754	188	5826	892508	1870	347249	177634471
B	–	–	–	–	–	–	–
C	271063	226	1182	624667	2144	174774	58266382
D	278012	186	5813	633801	1803	229323	127986142
E	–	–	–	–	–	–	–
F	–	–	–	–	–	–	–
G	299268	566	3444	818690	6134	312738	147441219
H	–	–	–	–	–	–	–

(which could be interesting). Note that the second experimental platform includes a multi-core processor: it has two processor cores. Using the platforms we constructed eight separate implementations which represent a cross-section of the presented approaches to intra-pairing and inter-pairing parallelism (recalling that we have a fixed parameterisation where p is a 256-bit prime):

Implementation *A* uses the scalar (i.e. non-SIMD) instruction set and a 32-bit digit size; evaluates one pairing at a time using Algorithm 1.

Implementation *B* uses the scalar (i.e. non-SIMD) instruction set and a 64-bit digit size; evaluates one pairing at a time using Algorithm 1.

Implementation *C* uses the SIMD (i.e. SSE) instruction set and a 29-bit digit size to perform digit-serial $\mathbb{F}_p$ arithmetic; evaluates one pairing at a time using Algorithm 1.

Implementation *D* uses the SIMD (i.e. SSE) instruction set and a 32-bit digit size to perform 2-way digit-sliced $\mathbb{F}_{p^2}$ arithmetic (i.e. two $\mathbb{F}_p$ operations in parallel); evaluates one pairing at a time using Algorithm 1.

Implementation *E* takes Implementation B as a starting point, uses OpenMP to perform parallel $\mathbb{F}_{p^6}$ arithmetic within $\mathbb{F}_{p^{12}}$ and parallel $\mathbb{F}_{p^2}$ arithmetic within Algorithm 1 in order to evaluate one pairing at a time.

Implementation *F* takes Implementation B as a starting point, but uses OpenMP to implement Algorithm 2 and thereby evaluate one pairing at a time.

Implementation *G* uses the SIMD (i.e. SSE) instruction set and a 32-bit digit size to perform 2-way digit-sliced pairing evaluation (i.e. two $e(P,Q)$ operations in parallel) and therefore evaluates two pairings at a time.

Implementation *H* takes Implementation B as a starting point, but uses OpenMP to execute two instances of Algorithm 1 in parallel and therefore evaluate two pairings at a time.

5.1 Analysis of Results

Timings obtained by executing these implementation on the two experimental platforms are detailed in Tables 2 and 3. In each case the number of cycles (as reported by `rdtsc`) required for the entire operation is quoted. That is, if an operation generates n results in parallel then the tables quote the total time: the per-result time requires division by n.

Table 3. Timings for major operations (in cycles as reported by `rdtsc`) on experimental platform ***B*** (Core2 Duo). $\mathbb{F}_p$ is a 256-bit prime field.

	$\mathbb{F}_p$			$\mathbb{F}_{p^{12}}$			$e(P,Q)$
	Inv	Add	Mul	Inv	Add	Mul	
A	156179	132	1117	287160	1061	76002	44814516
B	155567	107	395	208603	779	31484	23319673
C	155514	114	477	290536	842	64490	28452901
D	154295	132	1106	278503	1062	73336	35215963
E	154217	107	399	207236	787	24494	14429439
F	155567	108	394	208612	781	31491	25321173
G	157287	261	1444	390626	2705	137356	64879334
H	155567	108	390	208607	773	31485	25925534

Although our results are not exhaustive, for the given parameterisation they prompt some interesting conclusions. On the Pentium 4 based platform, if one is required to evaluate a single pairing then the best option is to parallelise arithmetic in $\mathbb{F}_p$ (Implementation C); if the requirement is for two pairing evaluations, the best option is actually two invocations of Implementation C. On the Core2 based platform, if one is required to evaluate a single pairing then it makes more sense to use 64-bit scalar arithmetic in $\mathbb{F}_p$ and multi-core parallel arithmetic within $\mathbb{F}_{p^{12}}$ and the pairing itself (Implementation E) than consider SIMD parallelism; if the requirement is for two pairing evaluations, the slightly moronic conclusion is that one can perform one pairing on each core (Implementation H), doubling the performance versus two sequential invocations of any other method that does not already use multi-core parallelism internally.

5.2 Analysis of Platforms

Design of SIMD Instruction Sets. Interestingly, in early 2008 Intel announced an updates to the SSE lineage of SIMD instruction sets, and a totally new instruction set specialised toward implementation of AES. Specifically, the Advanced Vector Extensions (AVX) includes the `pclmulqdq` instruction for carryless multiplication that can be used to accelerate arithmetic in binary finite fields. In addition, the Advanced Encryption Standard Instructions Set (AES-NI) includes instructions that perform whole AES rounds with the view to improving performance and eliminating cache based side-channel attack.

In contrast with this new emphasis on supporting cryptography, our results show that on a current Core2 platform, a 64-bit implementation (Implementation B) is faster than that based on SIMD parallel techniques. In the short term, microprocessors with a 64-bit datapath width seem sure to be ubiquitous before longer operands (e.g. 512-bit). One might conclude that using current technology, non-parallel implementation is best; given the specific nature of the updates described above, it seems this will remain the fact in next-generation processors. This seems an unattractive conclusion since it implies that current support for SIMD parallelism is less effective that it could be for this particular domain. We posit that this problem demands research into more public-key cryptography centric SIMD instruction sets: in the longer term, the chance of the processor datapath width doubling (e.g. from 64-bit to 128-bit) is less likely than the operand length doubling and so effective use of parallelism is crucial to scalability.

In a sense, it is not a surprise that Implementation B outperforms C on the Core2 platform. For example, the SSE3 instruction set allows 2-way parallel 32×32-bit multiplication; the cost of such multiplication plus the overhead of data reorganisation will intuitively be greater than native 64×64-bit multiplication. Furthermore, the SSE3 instruction set lacks a method for performing an add-with-carry operation that exists in the scalar instruction set. As such, enhancements over SSE3 such as the `pshufb` instruction help to reduce said overhead but the instruction set still lacks features which could improve performance of our results. For example, the PLX [30] processor eases the issue of shuffles between sub-words by including odd and even multiplication, i.e. both

$$\overline{r}_{2i+1\ldots 2i+0} = \overline{x}_{2i+0} \cdot \overline{y}_{2i+0}$$

and

$$\overline{r}_{2i+1\ldots 2i+0} = \overline{x}_{2i+1} \cdot \overline{y}_{2i+1}$$

for $i \in \{0, 1\}$. Another improvement would be provision of hardware support for add-with-carry via vector-carry registers; Fournier [18] investigates this approach within the context of a dedicated vector processor. The upcoming SSE5 instruction set offers an alternative approach by departing from purely 3-address instructions by adding support for a range of 4-address alternatives. In this context, it seems possible to extend the instruction set further and allow explicit specification of a vector-carry register rather than via an implicit, special purpose register as proposed by Fournier.

Effective Utilisation of Multi-core. Another interesting feature is that using the multi-core capabilities of the Core2 platform to evaluate one pairing, we are presented with two problems. Firstly, the overhead from use of OpenMP limits where we can exploit the inherent parallelism within field arithmetic; if the processor had a more light-weight means of managing fine-grained threads, Implementation E would potentially be even more lucrative. The results from using course-grained threads in Algorithm 2 are underwhelming. The low Hamming weight of s coupled with the significant overhead introduced by using projective rather than mixed point (resp. line) addition means it is slower than the non-parallel alternative. The first problem motivates research into fine-grained multi-core and multi-threaded processors; an exemplar design is the XCore. The second problem motivates research into forms of easily parallelised pairing algorithms.

6 Conclusions

The efficient evaluation of cryptographic pairings underpins a wide range of modern cryptographic applications. There are a wide range of parameterisation and implementation options to consider, in this paper we focused on the exploitation of parallelism in software. The capabilities of modern processors in this respect are diverse; the correct option and realisation in terms of implementation is therefore far from trivial. In particular we found that, unlike implementation in hardware, on a Pentium 4 based platform one should parallelise arithmetic in $\mathbb{F}_p$ rather than a higher level; on a Core2 based platform one should utilise native support for 64-bit arithmetic and then harness the multi-core features to parallelise arithmetic in $\mathbb{F}_{p^{12}}$ and the pairing itself. Although our results improve significantly

on a naive approach, we identified areas for further improvement through study of new algorithm types and changes to processor architecture. The results for arithmetic in $\mathbb{F}_p$ have a direct implication for vanilla ECC in which it seems a similar argument wrt. to implementation approach should apply.

Acknowledgements

The authors would like to thank Peter Schwabe for helping to correct some problems with initial performance results.

References

1. T. Acar. High-Speed Algorithms & Architectures For Number-Theoretic Cryptosystems. PhD Thesis, Oregon State University, 1997.
2. K. Aoki, F. Hoshino, T. Kobayashi and H. Oguro. Elliptic Curve Arithmetic Using SIMD. In *Information Security Conference (ISC)*, Springer-Verlag LNCS 2200, 235–247, 2001.
3. D.V. Bailey and C. Paar. Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography. In *Journal of Cryptology* **14** (3), 153–176, 2001.
4. P.S.L.M. Barreto, S. Galbraith, C. Ó hÉigeartaigh and M. Scott. Efficient Pairing Computation on Supersingular Abelian Varieties. In *Designs, Codes and Cryptography*, **42** (3), 239–271, 2007.
5. P.S.L.M. Barreto, H. Kim, B. Lynn and M. Scott. Efficient Algorithms for Pairing-Based Cryptosystems. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 2442, 354–368, 2002.
6. P.S.L.M. Barreto and M. Naehrig. Pairing-friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography (SAC)*, Springer-Verlag LNCS 3897, 319–331, 2005.
7. R. Bhaskar, P.K. Dubey, V. Kumar, A. Rudra and A. Sharma. Efficient Galois Arithmetic on SIMD Architectures. In *ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, 256–257, 2003.
8. E. Biham. A Fast New DES Implementation in Software. In *Fast Software Encryption (FSE)*, Springer-Verlag LNCS 1267, 260–272, 1997.
9. D. Boneh, B. Lynn and H. Shacham. Short signatures from the Weil pairing. In *Journal of Cryptology*, **17** (4), 297–319, 2004.
10. A. Bosselaers, R. Govaerts and J. Vandewalle. SHA: A Design for Parallel Architectures ? In *Advances in Cryptology (EUROCRYPT)*, Springer-Verlag LNCS 1233, 348–362, 1997.
11. C.S.K. Clapp. Optimizing a Fast Stream Cipher for VLIW, SIMD, and Superscalar Processors. In *Fast Software Encryption (FSE)*, Springer-Verlag LNCS 1267, 273–287, 1997.
12. R. Crandall and J. Klivington. Vector Implementation of Multiprecision Arithmetic. Technical Report, 1999.
13. A.J. Devegili, C. Ó hÉigeartaigh, M. Scott, and R. Dahab, Multiplication and Squaring on Pairing-Friendly Fields, In *Cryptology ePrint Archive, Report 2006/471*, 2006.
14. A.J. Devegili, M. Scott and R. Dahab. Implementing Cryptographic Pairings over Barreto-Naehrig Curves In *Pairing Based Cryptography (PAIRING)*, Springer-Verlag LNCS 4575, 197–207, 2007.
15. I. Duursma and H. Lee. Tate Pairing Implementation for Hyperelliptic Curves $y^2 = x^p - x + d$. In *Advances in Cryptology (ASIACRYPT)*, Springer-Verlag LNCS 2894, 111–123, 2003.
16. J. Fan, K. Sakiyama and I. Verbauwhede. Montgomery Modular Multiplication Algorithm on Multi-Core Systems. In *IEEE Workshop on Signal Processing Systems: Design and Implementation (SIPS)*, 261–266, 2007.
17. J. Fan, K. Sakiyama and I. Verbauwhede. Elliptic Curve Cryptography on Embedded Multicore Systems. In *Workshop on Embedded Systems Security (WESS)*, 17–22, 2007.
18. J.J.A. Fournier. Vector Microprocessors for Cryptography. PhD Thesis, University of Cambridge, 2007.
19. S. Galbraith, K. Harrison and D. Soldera. Implementing the Tate Pairing. In *Algorithmic Number Theory Symposium (ANTS-V)*, Springer LNCS 2369, 324–337, 2002.
20. R. Granger, D. Page and N.P. Smart. High Security Pairing-Based Cryptography Revisited. In *Algorithmic Number Theory Symposium (ANTS-VII)*, Springer-Verlag LNCS 4076, 480–494, 2006.

21. R. Granger and N.P. Smart. On Computing Products of Pairings. In *Cryptology ePrint Archive*, Report 2006/172, 2006.
22. D. Hankerson, A. Menezes and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.
23. F. Hess, N.P. Smart and F. Vercauteren. The Eta Pairing Revisited. In *Transactions on Information Theory* **52**, 4595–4602, 2006.
24. Intel Cooperation. Using Streaming SIMD Extensions (SSE2) to Perform Big Multiplications. Technical Report, 2000.
25. T. Izu and T. Takagi. Fast Elliptic Curve Multiplications with SIMD Operations. In *4th International Conference on Information and Communications Security (ICICS)*, Springer-Verlag LNCS 2513, 217–230, 2002.
26. T. Kerins, W.P. Marnane, E.M. Popovici and P.S.L.M. Barreto. Efficient Hardware for the Tate Pairing Calculation in Characteristic Three. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 3659, 412–426, 2005.
27. N. Koblitz and A. Menezes. Pairing-based Cryptography at High Security Levels. In *Cryptography and Coding*, Springer-Verlag LNCS 3796, 13–36, 2005.
28. C.K. Koc, T. Acar and B.S. Kaliski. Analyzing and Comparing Montgomery Multiplication Algorithms. In *IEEE Micro*, **16** (3), 26–33, 1996.
29. S. Kwon. Efficient Tate Pairing Computation for Elliptic Curves over Binary Fields. In *Information Security and Privacy (ISP)*, Springer-Verlag LNCS 3574, 134–145, 2005.
30. R.B. Lee and A.M. Fiskiran. PLX: A Fully Subword-Parallel Instruction Set Architecture for Fast Scalable Multimedia Processing In *IEEE International Conference on Multimedia and Expo (ICME)*, 117–120, 2002.
31. H. Lipmaa. IDEA: A Cipher for Multimedia Architectures? In *Selected Areas in Cryptography (SAC)*, Springer-Verlag LNCS 1556, 248–263, 1998.
32. M. Matsui and J. Nakajima. On the Power of Bitslice Implementation on Intel Core2 Processor. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 4727, 121–134, 2007.
33. V. Miller. Short programs for functions on curves. Available at: `http://crypto.stanford.edu/miller/miller.pdf`
34. P.L. Montgomery. Modular Multiplication without Trial Division. In *Mathematics of Computation*, **44** (170), 519–521, 1985.
35. P.L. Montgomery. Vectorization of the Elliptic Curve Method. Available at: `ftp://ftp.cwi.nl/pub/pmontgom/ecmvec.psl.gz`
36. J. Nakajima and M. Matsui. Performance Analysis and Parallel Implementation of Dedicated Hash Functions. In *Advances in Cryptology (EUROCRYPT)*, Springer-Verlag LNCS 2332, 165–180, 2002.
37. D. Page and N.P. Smart. Parallel Cryptographic Arithmetic Using a Redundant Montgomery Representation. In *IEEE Transactions on Computers*, **53** (11), 1474–1482, 2004.
38. M. Scott. Implementing Cryptographic Pairings. Available at: `ftp://ftp.computing.dcu.ie/pub/resources/crypto/pairings.pdf`