High-speed parallel software implementation of the η_T pairing

Diego F. Aranha
Institute of Computing – UNICAMP

Joint work with

Julio López and Darrel Hankerson

Introduction

Pairing computation is the most expensive operation in Pairing-Based Cryptography.

Parallelism is being increasingly introduced in modern architectures.

Objective

Explore two types of parallelism in software to reduce pairing computation latency:

- Vector instructions;
- Multiprocessing.

Applications: real-time services (DNS?), embedded devices.

Contributions

- Novel ways for implementing binary field arithmetic;
- Parallelization of Miller's Algorithm;
- Static load balancing technique;
- Experimental results.

Arsenal

Intel Core architecture:

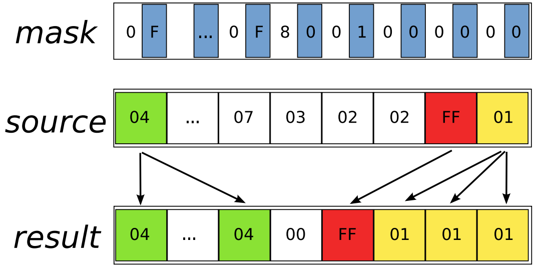
- 128-bit *Streaming SIMD Extensions* instruction set;
- Multiprocessing with overheads of around 10 microsec;
- *Super shuffle engine* introduced in 45 nm series.

Relevant vector instructions:

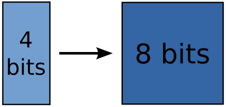
Instruction	**Description**	**Cost**	**Mnemonic**
`MOVDQA`	Memory load/store	2.5	$\leftarrow$
`PSLLQ`, `PSRLQ`	64-bit bitwise shifts	1	$\ll_{\nmid 8}, \gg_{\nmid 8}$
`PXOR,PAND,POR`	Bitwise `XOR,AND,OR`	1	$\oplus, \wedge, \vee$
`PUNPCKLBW/HBW`	Byte interleaving	3	*interlo/hi*
`PSLLDQ,PSRLDQ`	128-bit bytewise shift	2 (1)	$\ll_8, \gg_8$
`PSHUFB`	Byte shuffling	3 (1)	*shuffle,lookup*
`PALIGNR`	Memory alignment	2 (1)	$\lhd$

New SSSE3 instructions

PSHUFB instruction (`_mm_shuffle_epi8`):

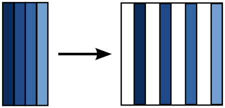


Real power: We can implement **in parallel** any function:



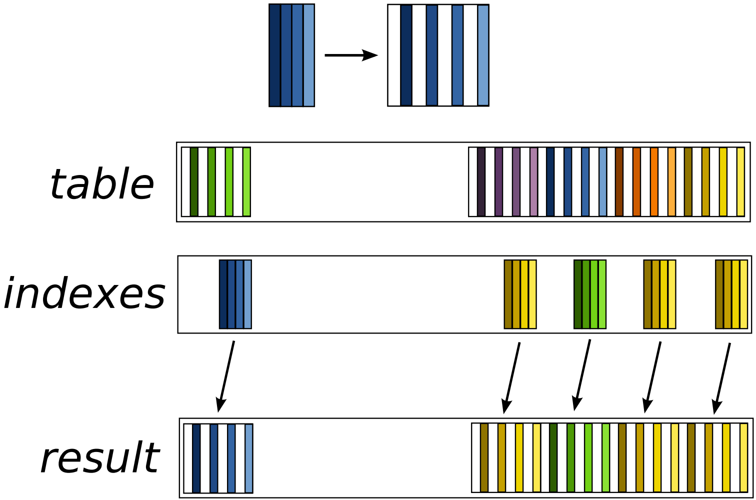
New SSSE3 instructions

Example: Bit manipulation



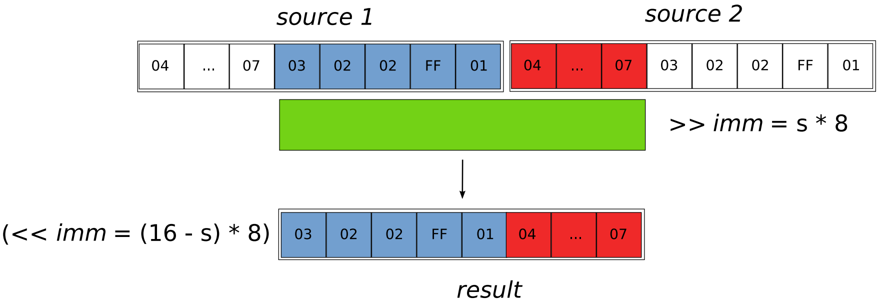
New SSSE3 instructions

Example: Bit manipulation



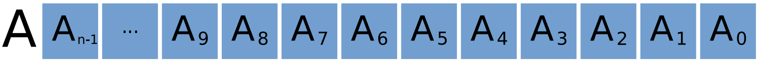
New SSSE3 instructions

PALIGNR instruction (_mm_alignr_epi8):



Binary field $\mathbb{F}_{2^m}$

- Irreducible polynomial: $f(z)$ (trinomial or pentanomial)
- Polynomial basis: $a(z) \in \mathbb{F}_{2^m} = \sum_{i=0}^{m-1} a_i z^i$.
- Software representation: vector of $n = \lceil m/64 \rceil$ words.
- Notation: A is a 64-bit variable, $\overline{A}$ is a 128-bit variable.
- Graphical representation:



Squaring in $\mathbb{F}_{2^m}$

$$a(z) = \sum_{i=0}^{m} a_i z^i = a_{m-1} + \cdots + a_2 z^2 + a_1 z + a_0$$

$$a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i} = a_{m-1} z^{2m-2} + \cdots + a_2 z^4 + a_1 z^2 + a_0$$

Example:

$$a(z) = (a_{m-1}, a_{m-2}, \ldots, a_2, a_1, a_0)$$

$$a(z)^2 = (a_{m-1}, 0, a_{m-2}, 0, \ldots, 0, a_2, 0, a_1, 0, a_0)$$

Squaring in $\mathbb{F}_{2^m}$

We can write:

$$a(z) = a_L(z) + a_H(z) \cdot z^4.$$

Since squaring is a linear operation:

$$a(z)^2 = a_L(z)^2 + a_H(z)^2 \cdot z^8.$$

Polynomials $a_L(z)$ and $a_H(z)$ are easy to compute:

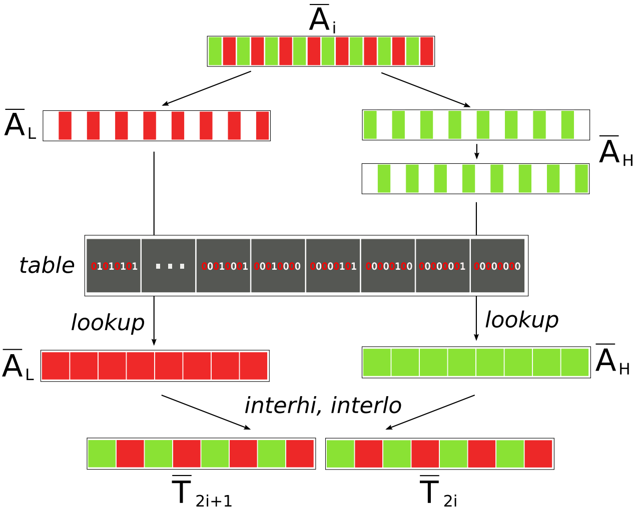
$$\overline{A_L} = \overline{A_i} \wedge \texttt{0x0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F}$$
$$\overline{A_H} = \overline{A_i} \wedge \texttt{0xF0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0}$$

We can compute $a_L(z)^2$ and $a_H(z)^2$ with a lookup table.

For $u = (u_3, u_2, u_1, u_0)$ we use $table(u) = (0, u_3, 0, u_2, 0, u_1, 0, u_0)$:

table								
	0	1	2	3	4	5	6	7
	00000000	00000001	00000100	00000101	00010000	00010001	00010100	00010101
	8	9	10	11	12	13	14	15
	01000000	01000001	01000100	01000101	01010000	01010001	01010100	01010101

Proposed squaring in $\mathbb{F}_{2^m}$



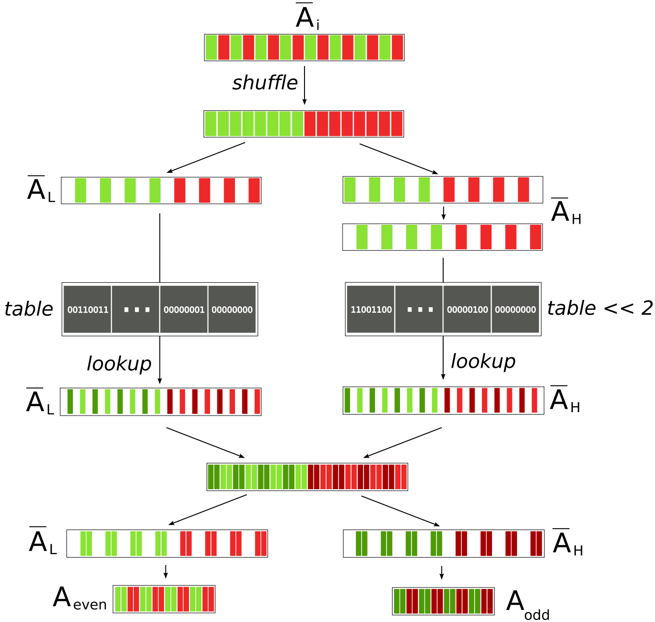
$$t(z) = a_L(z)^2 + a_H(z)^2 \cdot z^8$$

Square root extraction in $\mathbb{F}_{2^m}$

$$
\begin{aligned}
\sqrt{a} = a^{2^m-1} &= \sum_{i=0}^{m-1} \left(a_i z^i\right)^{2^m-1} = \sum_{i=0}^{m-1} a_i \left(z^{2^m-1}\right)^i \\
&= \sum_{i \text{ even}} a_i z^{\frac{i}{2}} + \sqrt{z} \sum_{i \text{ odd}} a_i z^{\frac{i-1}{2}} \\
&= a_{\text{even}} + \sqrt{z} \cdot a_{\text{odd}}
\end{aligned}
$$

For $f(z) = z^{1223} + z^{255} + 1$ in $\mathbb{F}_{2^{1223}}$, we have $\sqrt{z} = z^{612} + z^{128}$.

Important: Multiplication by $\sqrt{z}$ requires shifts and additions only.


$\overline{A}_i$
shuffle
$\overline{A}_L$
$\overline{A}_H$
table
00110011
00000001
00000000
11001100
00000100
00000000
table << 2
lookup
lookup
$\overline{A}_L$
$\overline{A}_H$
$\overline{A}_L$
$\overline{A}_H$
A_{even}
A_{odd}

Multiplication in $\mathbb{F}_{2^m}$

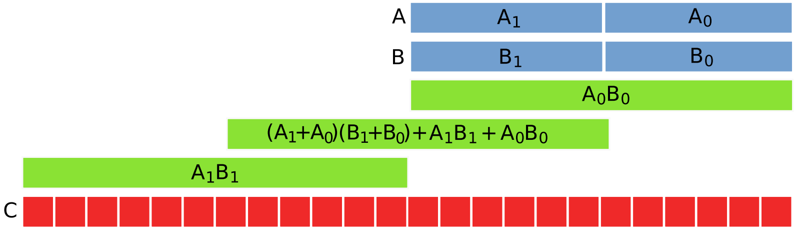
1. Multi-precision multiplication:
   - An instance of Karatsuba;
   - López-Dahab *comb* method;
2. Modular reduction.

Karatsuba multiplication in $\mathbb{F}_{2^m}$

$$
\begin{aligned}
c(z) &= a(z) \cdot b(z) \\
&= A_1 B_1 z^m + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] z^{\lceil m/2 \rceil} + A_0 B_0.
\end{aligned}
$$

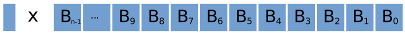
Karatsuba multiplication in $\mathbb{F}_{2^m}$

$$
\begin{aligned}
c(z) &= a(z) \cdot b(z) \\
&= A_1 B_1 z^m + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] z^{\lceil m/2 \rceil} + A_0 B_0.
\end{aligned}
$$

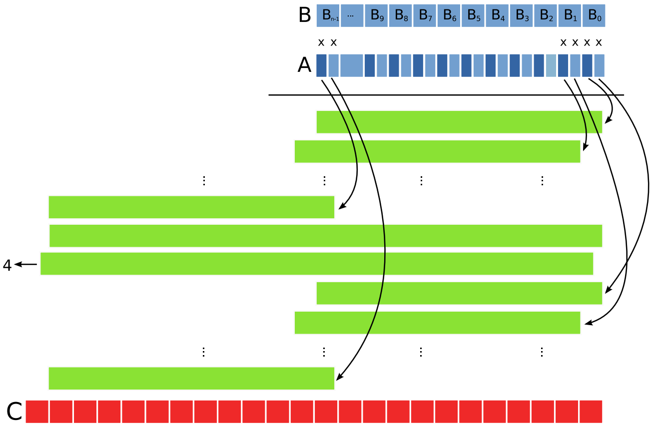


López-Dahab multiplication in $\mathbb{F}_{2^m}$

We can compute $u \cdot b(z)$ using shifts and additions.



If $a(z)$ is divided into 4-bit polynomials, compute $a(z) \cdot b(z)$ by:



Proposed multiplication in $\mathbb{F}_{2^m}$

Algorithm 1 LD multiplication implemented with n 128-bit registers.

Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..n-1]$.
Note: m_i denotes the vector of $\frac{n}{2}$ 128-bit registers $(r_{(i-1+n/2)}, \ldots, r_i)$.

Compute $T_0(u) = u(z) \cdot b(z), T_1(u) = u(z) \cdot (b(z) \ll 4)$ for all $u(z)$ of degree lower than 4.
$(r_{n-1} \ldots, r_0) \leftarrow 0$
for $k \leftarrow 56$ **downto** 0 **by** 8 **do**
    for $j \leftarrow 1$ **to** $n-1$ **by** 2 **do**
        Let $u = (u_3, u_2, u_1, u_0)$, where u_t is bit $(k+t)$ of $a[j]$.
        Let $v = (v_3, v_2, v_1, v_0)$, where v_t is bit $(k+t+4)$ of $a[j]$.
        $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_0(u)$, $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_1(v)$
    end for
    $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
end for
for $k \leftarrow 56$ **downto** 0 **by** 8 **do**
    for $j \leftarrow 0$ **to** $n-2$ **by** 2 **do**
        Let $u = (u_3, u_2, u_1, u_0)$, where u_t is bit $(k+t)$ of $a[j]$.
        Let $v = (v_3, v_2, v_1, v_0)$, where v_t is bit $(k+t+4)$ of $a[j]$.
        $m_{j/2} \leftarrow m_{j/2} \oplus T_0(u)$, $m_{j/2} \leftarrow m_{j/2} \oplus T_1(v)$
    end for
    if $k > 0$ **then** $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
end for
return $c = (r_{n-1} \ldots, r_0) \bmod f(z)$

Modular reduction (64-bit mode)

Algorithm 2 Fast modular reduction by $f(z) = z^{1223} + z^{255} + 1$.

Input: $c(z) = c[0..2n-1]$.
Output: $c(z) \bmod f(z) = c[0..n-1]$.

for $i \leftarrow 2n-1$ **downto** n **do**
  $t \leftarrow c[i]$
  $c[i-15] \leftarrow c[i-15] \oplus (t \gg 8)$
  $c[i-16] \leftarrow c[i-16] \oplus (t \ll 56)$
  $c[i-19] \leftarrow c[i-19] \oplus (t \gg 7)$
  $c[i-20] \leftarrow c[i-20] \oplus (t \ll 57)$
end for
$t \leftarrow c[19] \gg 7, c[0] \leftarrow c[0] \oplus t, t \leftarrow t \ll 7$
$c[3] \leftarrow c[3] \oplus (t \ll 56)$
$c[4] \leftarrow c[4] \oplus (t \gg 8)$
$c[19] \leftarrow (c[19] \oplus t) \wedge \texttt{0x7F}$
return c

Algorithm 3 Proposed fast modular reduction.

Input: $t(z) = t[0..n-1]$ (vector of 128-bit elements).
Output: $c(z) \bmod f(z) = c[0..n-1]$.
Note: The accumulate function $R(r_3, r_2, r_1, r_0, t)$ executes:

$s \leftarrow t \gg_{\nmid 8} 7, r_3 \leftarrow t \ll_{\nmid 8} 57$
$r_3 \leftarrow r_3 \oplus (s \ll_8 64)$
$r_2 \leftarrow r_2 \oplus (s \gg_8 64)$
$r_1 \leftarrow r_1 \oplus (t \ll_8 56)$
$r_0 \leftarrow r_0 \oplus (t \gg_8 72)$

$r_0, r_1, r_2, r_3 \leftarrow 0$
for $i \leftarrow 19$ **downto** 15 **by** 4 **do**
  $R(r_3, r_2, r_1, r_0, t[i])$, $t[i-7] \leftarrow t[i-7] \oplus r_0$
  $R(r_0, r_3, r_2, r_1, t[i-1])$, $t[i-8] \leftarrow t[i-8] \oplus r_1$
  $R(r_1, r_0, r_3, r_2, t[i-2])$, $t[i-9] \leftarrow t[i-9] \oplus r_2$
  $R(r_2, r_1, r_0, r_3, t[i-3])$, $t[i-10] \leftarrow t[i-10] \oplus r_3$
end for
$R(r_3, r_2, r_1, r_0, t[11])$, $t[4] \leftarrow t[4] \oplus r_0$
$R(r_0, r_3, r_2, r_1, t[10])$, $t[3] \leftarrow t[3] \oplus r_1$
$t[2] \leftarrow t[2] \oplus r_2$, $t[1] \leftarrow t[1] \oplus r_3$, $t[0] \leftarrow t[0] \oplus r_0$
$r_0 \leftarrow m[9] \gg_8 64$, $r_0 \leftarrow r_0 \gg_{\nmid 8} 7$, $t[0] \leftarrow t[0] \oplus r_0$
$r_1 \leftarrow r_0 \ll_8 64$, $r_1 \leftarrow r_1 \ll_{\nmid 8} 63$, $t[1] \leftarrow t[1] \oplus r_1$
$r_1 \leftarrow r_0 \gg_{\nmid 8} 1$, $t[2] \leftarrow t[2] \oplus r_1$
for $i \leftarrow 0$ **to** 9 **do** $c[2i] \leftarrow store(t[i])$, $c[19] \leftarrow c[19] \wedge$ `0x7F`
return c

Implementation timings

	Operation		
Implementation	$a^2 \bmod f$	$a^{\frac{1}{2}} \bmod f$	$a \cdot b \bmod f$
Hankerson *et al.*	600	500	8200
Beuchat *et al.*	480	749	5438
This work (65nm)	160	166	4030
Improvement	66.7%	66.8%	25.9%
This work (45nm)	108	140	3785

Table: Timings are reported in cycles.

Bilinear Pairings

Let $\mathbb{G}_1 = \langle P \rangle$ and $\mathbb{G}_2 = \langle Q \rangle$ be additive groups and $\mathbb{G}_T$ be a multiplicative group such that $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| =$ prime q.

An efficiently-computable map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an **admissible bilinear map** if the following properties are satisfied:

1. *Bilinearity:* given $(V,\ W) \in \mathbb{G}_1 \times \mathbb{G}_2$ and $(a,\ b) \in \mathbb{Z}_q^*$: $e(aV, bW) = e(V, W)^{ab} = e(abV, W) = e(V, abW)$.
2. *Non-degeneracy:* $e(P, Q) \neq 1_{\mathbb{G}_T}$, where $1_{\mathbb{G}_T}$ is the identity of the group $\mathbb{G}_T$.

Pairing computation

Let P, Q be r-torsion points. The pairing $e(P, Q)$ is defined by the evaluation of $f_{r,P}$ at a divisor related to Q.

[Miller 1986] constructed $f_{r,P}$ in stages combining **Miller functions** evaluated at divisors.

[Barreto et al. 2002] showed how to evaluate $f_{r,P}$ at Q using the final exponentiation employed by the Tate pairing.

Pairing computation

Let $g_{U,V}$ be the line equation through points $U, V \in E(\mathbb{F}_{q^k})$ and g_U the shorthand for $g_{U,-U}$.

For any integers a and b, we have:

1. $f_{a+b,P}(\mathcal{D}) = f_{a,P}(\mathcal{D}) \cdot f_{b,P}(\mathcal{D}) \cdot \dfrac{g_{aP,bP}(\mathcal{D})}{g_{(a+b)P}(\mathcal{D})}$;
2. $f_{2a,P}(\mathcal{D}) = f_{a,P}(\mathcal{D})^2 \cdot \frac{g_{aP,aP}(\mathcal{D})}{g_{2aP}(\mathcal{D})}$;
3. $f_{a+1,P}(\mathcal{D}) = f_{a,P}(\mathcal{D}) \cdot \frac{g_{(a)P,P}(\mathcal{D})}{g_{(a+1)P}(\mathcal{D})}$.

Pairing computation

Algorithm 4 Miller's Algorithm [Miller 1986, Barreto et al. 2002].

Entrada: $r = \sum_{i=0}^{\log_2 r} r_i 2^i, P, Q.$

Saída: $e_r(P, Q).$

$T \leftarrow P$

$f \leftarrow 1$

$r \leftarrow r - 1$

for $i = \lfloor \log_2(r) \rfloor - 1$ **downto** 0 **do**

    $f \leftarrow f^2 \cdot l_{T,T}(Q)$

    $T \leftarrow 2T$

    if $r_i = 1$ **then**

        $f \leftarrow f \cdot l_{T,P}(Q)$

        $T \leftarrow T + P$

    end if

end for

return $f^{(q^k - 1/r)}$

Related work

Scalable approaches:

- [Mitsunari 2009] and [Beuchat et al. 2009] precompute pairs (T_i, part of $l_{T_i,T_i}(Q)$) in the symmetric case and divide loop iterations among processors.

Problem: High storage costs (large precomputation).

New approach

Property of Miller functions

$$f_{a \cdot b,P}(\mathcal{D}) = f_{b,P}(\mathcal{D})^{a} \cdot f_{a,bP}(\mathcal{D})$$

New approach

Property of Miller functions

$$f_{a\cdot b,P}(\mathcal{D}) = f_{b,P}(\mathcal{D})^{a} \cdot f_{a,bP}(\mathcal{D})$$

We can write $r = 2^w r_1 + r_0$ and compute $f_{r,P}(\mathcal{D})$:

$$\begin{aligned} f_{r,P}(\mathcal{D}) &= f_{2^w r_1 + r_0,P}(\mathcal{D}) \\ &= f_{r_1,P}(\mathcal{D})^{2^w} \cdot f_{2^w, r_1 P}(\mathcal{D}) \cdot f_{r_0,P}(\mathcal{D}) \cdot \frac{g_{(2^w r_1)P, r_0 P}(\mathcal{D})}{g_{rP}(\mathcal{D})}. \end{aligned}$$

New approach

Property of Miller functions

$$f_{a\cdot b,P}(\mathcal{D}) = f_{b,P}(\mathcal{D})^{a} \cdot f_{a,bP}(\mathcal{D})$$

We can write $r = 2^{w} r_1 + r_0$ and compute $f_{r,P}(\mathcal{D})$:

$$\begin{aligned} f_{r,P}(\mathcal{D}) &= f_{2^{w} r_1 + r_0, P}(\mathcal{D}) \\ &= f_{r_1,P}(\mathcal{D})^{2^{w}} \cdot f_{2^{w}, r_1 P}(\mathcal{D}) \cdot f_{r_0,P}(\mathcal{D}) \cdot \frac{g_{(2^{w} r_1)P, r_0 P}(\mathcal{D})}{g_{rP}(\mathcal{D})}. \end{aligned}$$

If r has low Hamming weight, w can be chosen so that r_0 is **small**.

For many processors, we can:

- Apply the formula recursively:
- Write r as $r = 2^{w_i} r_i + \cdots + 2^{w_2} r_2 + 2^{w_1} r_1 + r_0$.

If P is fixed (private key), $r_i P$ can also be precomputed.

Load balancing

Problem: We must determine an optimal partition w_i.

Let $c_1(1)$ the cost of a serial loop and $c_\pi(i)$ the cost of a parallel loop for processor $1 \leq i \leq \pi$.

Load balancing

Problem: We must determine an optimal partition w_i.

Let $c_1(1)$ the cost of a serial loop and $c_\pi(i)$ the cost of a parallel loop for processor $1 \leq i \leq \pi$.

We can count the operations executed by each processor and solve the system $c_\pi(1) = c_\pi(i)$ to obtain w_i. The **speedup** is:

$$s(\pi) = \frac{c_1(1) + exp}{c_\pi(1) + par + exp},$$

where *par* is the cost of parallelization and *exp* is the cost of the final exponentiation.

Symmetric case – Elliptic curves

A **pairing-friendly supersingular binary elliptic curve** is the set of solutions $(x, y) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$ satisfying the equation

$$y^2 + y = x^3 + x + \mathbf{b},$$

where $\mathbf{b} \in \{0, 1\}$, and a **point at infinity** ∞.

The **order** of this curve is $N = 2^m + 1 \pm 2^{\frac{m+1}{2}}$ and the **embedding degree** is $k = 4$ (the least integer such that N divides $2^{km} - 1$).

Symmetric case – Pairing definition

Choosing $T = 2^m - N$ and a prime r dividing N, [Barreto et al. 2004] defined the reduced η_T pairing:

$$\begin{aligned} \eta_T &: E(\mathbb{F}_{2^m})[r] \times E(\mathbb{F}_{2^m})[r] \rightarrow \mathbb{F}^*_{2^{4m}} \\ \eta_T(P, Q) &= f_{T',P'}(\psi(Q))^{\frac{2^{4m}-1}{N}}, \end{aligned}$$

where $T' = \pm T$ and $P' = \pm P$.

The function f is a **Miller function** and ψ is the **distortion map** $\psi(x, y) = (x^2 + s, y + sx + t)$.

Symmetric case – Pairing algorithm

Algorithm 5 η_T pairing [Barreto et al. 2004], [Beuchat et al. 2008].

Input: $P = (x_P, y_P), Q = (x_Q, y_Q) \in E(\mathbb{F}_{2^m})[r]$.
Output: $\eta_T(P, Q) \in \mathbb{F}^*_{2^{4m}}$.

$y_P \leftarrow y_P + 1 - \delta$
$u \leftarrow x_P + \alpha, v \leftarrow x_Q + \alpha$
$g_0 \leftarrow u \cdot v + y_P + y_Q + \beta$
$g_1 \leftarrow u + x_Q, g_2 \leftarrow v + x_P^2$
$G \leftarrow g_0 + g_1 s + t$
$L \leftarrow (g_0 + g_2) + (g_1 + 1)s + t$
$F \leftarrow L \cdot G$
for $i \leftarrow 1$ **to** $\frac{m-1}{2}$ **do**
    $x_P \leftarrow \sqrt{x_P}, y_P \leftarrow \sqrt{y_P}, x_Q \leftarrow x_Q^2, y_Q \leftarrow y_Q^2$
    $u \leftarrow x_P + \alpha, v \leftarrow x_Q + \alpha$
    $g_0 \leftarrow u \cdot v + y_P + y_Q + \beta$
    $g_1 \leftarrow u + x_Q$
    $G \leftarrow g_0 + g_1 s + t$
    $F \leftarrow F \cdot G$
end for
return $F^{(2^{2m}-1)(2^m+1\pm 2^{\frac{m+1}{2}})}$

Algorithm 6 Proposed parallel η_T pairing.

Input: $P = (x_P, y_P), Q = (x_Q, y_Q) \in E(\mathbb{F}_{2^m})[r]$.
Output: $\eta_T(P, Q) \in \mathbb{F}^*_{2^{4m}}$.

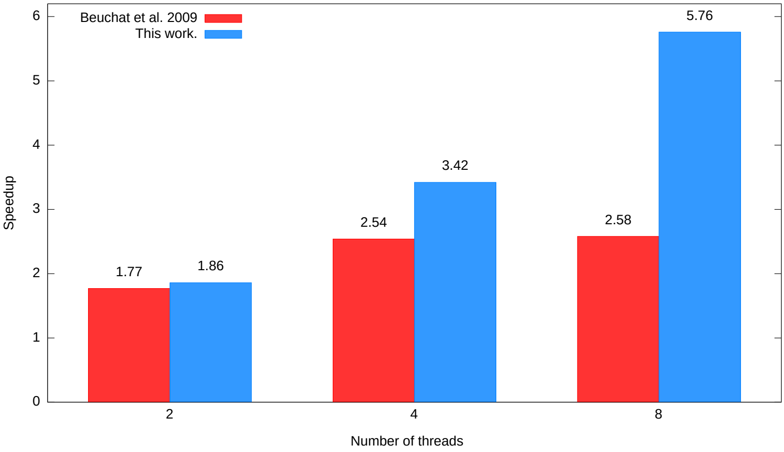
parallel section(processor i)
if $i = 1$ **then**
  Initialize F_1 as in lines 1-7 of the previous algorithm;
else
  $F_i \leftarrow 1$
end if
$x_{P_i} \leftarrow (x_P)^{\frac{1}{2^{w_i}}}, y_{P_i} \leftarrow (y_P)^{\frac{1}{2^{w_i}}}, x_{Q_i} \leftarrow (x_Q)^{2^{w_i}}, y_{Q_i} \leftarrow (y_Q)^{2^{w_i}}$
for $j \leftarrow w_i$ **to** $w_{i+1} - 1$ **do**
  $x_{P_i} \leftarrow \sqrt{x_{P_i}}, y_{P_i} \leftarrow \sqrt{y_{P_i}}, x_{Q_i} \leftarrow x_{Q_i}^2, y_{Q_i} \leftarrow y_{Q_i}^2$
  $u_i \leftarrow x_{P_i} + \alpha, v_i \leftarrow x_{Q_i} + \alpha$
  $g_{0_i} \leftarrow u_i \cdot v_i + y_{P_i} + y_{Q_i} + \beta$
  $g_{1_i} \leftarrow u_i + x_{Q_i}$
  $G_i \leftarrow g_{0_i} + g_{1_i}s + t$
  $F_i \leftarrow F_i \cdot G_i$
end for
$F \leftarrow \prod_{i=1}^{\pi} F_i$
end parallel
return F^M

Material:

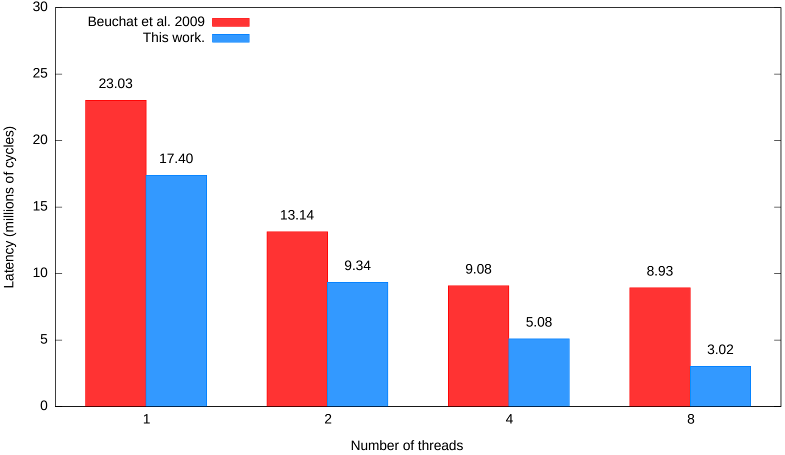
- GCC 4.1.2 (fastest SSE intrinsics);
- RELIC cryptographic library[1];
- OpenMP constructs;
- Intel 4-core 65nm and 8-core 45nm processors.

[1] http://code.google.com/p/relic-toolkit/

Experimental results – Speedup (45nm)



Experimental results – Latency (45nm)



Conclusions

New state-of-the-art for parallel implementation of pairings:

- No significant storage costs, smaller precomputation;
- Improvements in field arithmetic from 25% to 67%;
- In comparison with our serial implementation, speedups of 46%, 70% and 83% with 2, 4 and 8 cores;
- In comparison with previous state-of-the-art, improvements in *latency* of 24%, 29%, 44% and 66% with 1, 2, 4 and 8 cores.

Parallelization scales:

- In the covered case, point doublings and extension field squarings are efficient;
- Our finite field implementation make these exceptionally fast.

Future work

Extend techniques to other cases:

- Ternary case should be simple;
- Asymmetric case is harder (point doublings are expensive).

For the R-ate pairing over Barreto-Naehrig curves:

- Preliminary data points to a small 10% speedup with 2 processors.

Thank you for your attention!
Any questions?

Detailed results

	Number of threads					
Platform 1 – 65nm	**1**	**2**	**4**	**8***	**16***	**32***
Hankerson *et al.* – latency	39	–	–	–	–	–
Beuchat *et al.* – latency	26.86	16.13	10.13	–	–	–
Beuchat *et al.* – speedup	1	1.67	2.65	–	–	–
This work – latency	18.76	10.08	5.72	3.55	2.51	2.14
This work – speedup	1	1.86	3.28	5.28	7.47	8.76
Improvement	30.2%	32.9%	39.9%	–	–	–
Platform 2 – 45nm	**1**	**2**	**4**	**8**	**16***	**32***
Beuchat *et al.* – latency	23.03	13.14	9.08	8.93	–	–
Beuchat *et al.* – speedup	1	1.77	2.54	2.58	–	–
This work – latency	17.40	9.34	5.08	3.02	2.03	1.62
This work – speedup	1	1.86	3.42	5.76	8.57	10.74
Improvement	24.4%	28.9%	44.0%	66.2%	–	–

Table: Timings are reported in millions of cycles.