병렬 연산을 이용한 최적 확장체의 효율적 구현

이문규° 박근수
서울대학교 컴퓨터공학부
{mklee°, kpark}@theory.snu.ac.kr

Efficient Implementation of Optimal Extension Fields Using Parallel Computation

Mun-Kyu Lee° Kunsoo Park
School of Computer Science and Engineering, Seoul National University

요 약

본 논문에서는 타원 곡선 암호의 성능을 향상시키기 위한 효율적인 최적 확장체 연산 알고리즘을 제안한다. 제안하는 알고리즘은 CPU에서 제공되는 정수 곱셈 명령 1회 실행에 두 개의 하위체 연산을 병렬적으로 수행하도록 함으로써 최적 확장체에서의 곱셈, 제곱, 역원 연산의 속도를 향상시킨다.

1. 서론

타원 곡선 암호의 효율적 구현을 위해서는 기반이 되는 유한체(finite field) 연산의 구현이 매우 중요하다. 유한체는 소수 p와 양의 정수 m에 대해 $GF(p^m)$의 형태로 표현되며, 특히 p를 CPU의 워드 크기에 맞추어 선택한 특수한 경우를 최적 확장체(optimal extension field: OEF)[1,2]라 부른다. OEF에서 구현된 타원 곡선 암호는 이진체($p=2$인 경우)나 소수체($m=1$인 경우)의 경우보다 더 효율적인 것으로 알려져 있다[3].

그러나 단일 프로토콜의 수행을 위해 서로 다른 부류의 CPU가 혼용되는 경우 OEF의 장점이 반감되는 문제점이 있다. 예를 들어 8비트 스마트카드가 ECDSA 서명을 생성하고 32비트 CPU가 서명을 확인하는 경우(또는 반대의 경우)를 고려해 보자. 프로토콜의 모든 참가자는 같은 유한체를 사용하여야 하므로, OEF의 결정에 있어 8비트 스마트카드에 맞추는 경우($p \approx 2^8$로 선택)와 32비트 CPU에 맞추는 경우($p \approx 2^{32}$로 선택) 중 하나를 선택해야 한다. 이 때 상대적으로 약한 계산 능력을 지닌 스마트카드 쪽에 유리하도록 전자를 선택하는 것이 합리적인데, 이것은 후자를 택할 경우 프로토콜의 수행에 있어 스마트카드 쪽이 심각한 병목이 될 수 있기 때문이다. 따라서 우리는 $p \approx 2^8$인 p를 선택해야 하며, 결국 32비트 CPU는 하위체(subfield)인 $GF(p)$의 연산에서 32비트 레지스터 중 하위 8비트만을 사용하게 되어 연산 능력상의 낭비가 발생하게 된다.

이에, 본 연구에서는 32비트 CPU에서 제공되는 정수 연산 명령 1회 수행에 8비트 하위체 곱셈 두 개를 동시에 수행하도록 함으로써 CPU 사용을 최적화하는 OEF 상의 곱셈 알고리즘을 제안한다. 제안된 알고리즘을 이용하면 OEF 상의 역원 계산(inversion) 역시 향상시킬 수 있는데, 이것은 역원 계산이 실질적으로 OEF 곱셈의 반복으로 구성되기 때문이다. 본 연구에서 수행한 실험 결과에 따르면, 제안된 알고리즘은 OEF 상의 곱셈, 제곱, 역원 계산의 성능을 각각 40~52%, 31~37%, 28~40% 향상시킨다. 또한, 제안된 방법은 16비트 CPU와 64비트 CPU가 혼용되는 상황에도 동일하게 적용될 수 있다.

2. 최적 확장체와 기본 연산

2.1 최적 확장체(optimal extension field: OEF)

OEF[1,2]는 다음 조건을 만족하는 유한체 $GF(p^m)$이다.

- p는 CPU의 워드 크기에 맞추어 선택된다. (본 연구의 설정처럼 여러 종류의 CPU가 혼용되는 경우, 워드 크기가 작은 쪽에 맞추기로 한다.)
- $\log_2 c \le \lfloor n/2 \rfloor$을 만족하는 c에 대해 $p = 2^n \pm c$이다.
- 기약 이항 다항식(irreducible binomial) $f(x) = x^m - w$가 존재한다.

OEF의 원소 $A(x) \in GF(p^m)$는 다항식 기저(polynomial basis)를 이용하여 표현하기로 한다. 즉, $a_i \in GF(p)$에 대해

$$A(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$$

이다. 또한, p가 CPU의 워드 하나에 들어가므로 $A(x)$는 각각 a_i를 포함하는 m개의 레지스터를 이용하여 표현되며, 모든 산술 연산은 다항식 $f(x)$에 대해 mod 연산으로 수행된다.

2.2 OEF 상에서의 곱셈 연산

기본적인 OEF 곱셈은 두 단계로 구성된다. 단계 1에서는 두 OEF 원소 $A(x)$와 $B(x)$에 대해 다항식 곱셈을 수행하여 차수 $\le 2m-2$인 중간 결과 $C'(x)$를 얻는다. $C'(x)$의 계수 c_i ($i=0, 1, \ldots, 2m-2$)를 schoolbook 방법으로 계산하면 $GF(p)$ 상에서 m^2 개의 곱셈과 $(m-1)^2$ 개의 덧셈이 필요하다.[1] 단계 2에서는 $C'(x) \bmod f(x)$를 계산해 $C(x) = A(x) \cdot B(x) \bmod f(x)$를 얻는다.

2.3 OEF 상에서의 역원 연산 (inversion)

Algorithm 1. OEF inversion

Input: $A(x) \in GF(p^m)$.
Output: $A(x) \cdot B(x) \equiv 1 \bmod f(x)$인 $B(x) \in GF(p^m)$.
Step 1: $B(x) \leftarrow A^{r-1}(x)$.
Step 2: $c_0 \leftarrow B(x) \cdot A(x)$. ▷ $c_0 = A^r(x) \in GF(p)$.
Step 3: $c \leftarrow c_0^{-1} \bmod p$. ▷ $c = A^{-r}(x) \in GF(p)$.
Step 4: $B(x) \leftarrow B(x) \cdot c$. ▷ $B(x) = A^{-1}(x)$.

본 논문에서는 OEF 상의 기본 역원 연산으로 Bailey-Paar 알고리즘[2](위의 Algorithm 1)을 이용하기로 한다.[2] 이 알고리즘은 임의의 $A(x) \in GF(p^m)$와 $r = (p^m - 1)/(p-1)$에 대해 $A^r(x) \in GF(p)$이라는 사실을 이용하고 있다.

---

1) Karatsuba 곱셈[4]은 복잡도 상으로는 schoolbook 방법보다 유리하지만, 본 논문의 설정처럼 상대적으로 큰 m을 사용하는 OEF에 적용될 경우 실제로는 효율적이지 않다[5].

2) 비교적 작은 m에 대해서는 확장 유클리드 알고리즘 및 그 응용(예를 들면 [5]의 Algorithm IM)이 Algorithm 1보다 더 효율적이다. 그러나 본 연구에서 다루는 경우(비교적 큰 m과 작은 p)에는 Algorithm 1이 더 유리하다.

이 알고리즘의 핵심 부분은 Step 1이다. 그런데 주어진 체에 대해 $r-1=p^{m-1}+p^{m-2}+\cdots+p$가 미리 고정되므로 우리는 $r-1$의 p진수 표현인 $(11\ldots10)_p$를 미리 알 수 있고, 여기에 OEF 곱셈과 p승 연산을 이용하여 A^{r-1}을 계산할 수 있다. 예를 들어 $m=6$의 경우, A^{r-1}은 다음과 같이 계산된다.

$B \leftarrow A^p = A^{(10)}$; $C \leftarrow BA = A^{(11)}$; $B \leftarrow (C^p)^p = A^{(1100)}$;
$B \leftarrow BC = A^{(1111)}$; $B \leftarrow B^p = A^{(11110)}$; $B \leftarrow BA = A^{(11111)}$;
$B \leftarrow B^p = A^{(111110)}$.

여기서 p승 연산, 즉 Frobenius map은 $GF(p)$ 상의 곱셈 $m-1$개 만으로 계산 가능하므로[6], Step 1의 주요 연산은 OEF 상의 곱셈이다. 나머지 세 Step들의 수행 시간은 Step 1에 비해 매우 작으므로,[3] 결국 Algorithm 1의 핵심 연산은 OEF 상의 곱셈 연산이 된다.

2.4 타원 곡선 연산

타원 곡선 점은 아핀(affine) 좌표 또는 사영(projective) 좌표계로 표현할 수 있는데, OEF에서 정의된 곡선의 경우 역원 연산 대 곱셈의 비율이 다른 체에 비해 작으므로 아핀 좌표계를 이용하는 것이 유리하다. 아핀 좌표계에서는 점의 덧셈을 위해 OEF 상에서의 역원 연산 1회, 곱셈 2회, 제곱 1회가 필요하며, 점의 2배를 위해서는 제곱이 1회 더 필요하다. (OEF 덧셈이나 뺄셈 등 다른 연산 시간은 무시할 수 있다.)

타원 곡선 암호에서 가장 중요한 연산은 곡선상의 점 P와 정수(스칼라) k에 대해 $Q=kP$를 계산하는 연산, 즉 스칼라 곱셈(scalar multiplication)이다. 스칼라 곱셈을 위해 가장 널리 쓰이는 알고리즘은 비인접 형태(nonadjacent form: NAF)[7]를 이용한 부호화 이진(signed binary) 스칼라 곱셈인데, 여기서 NAF란 스칼라의 표현(전개)에서 각 자리 수가 0, 1, -1 중 하나가 되도록 하고, 0이 아닌 수끼리, 즉 1과 1, -1과 -1, 또는 1과 -1이 인접하는 경우가 없도록 전개한 부호화 이진 전개(signed binary expansion)이다.

3. OEF상의 효율적인 연산 구현

이 절에서는, OEF 상의 새로운 곱셈 알고리즘을 제안하며, 이것을 어떻게 역원 연산에 적용할 수 있는지 설명한다.

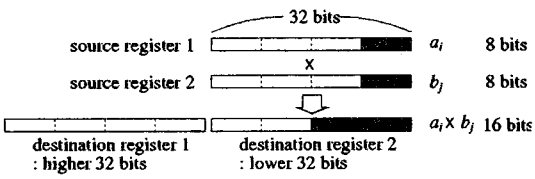

그림 1 32비트 명령어를 이용한 8×8비트 정수 곱셈

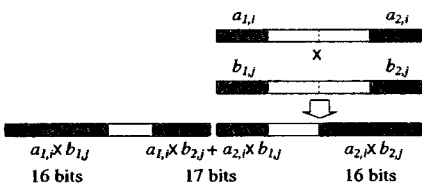

그림 2 $a_{1,i} \times b_{1,j}$, $a_{2,i} \times b_{2,j}$의 동시 계산

일반적인 32비트 CPU에서의 정수 곱셈은 먼저 두 개의 32비트 소스(source) 레지스터를 피연산자로 채운 후 두 개의 32비트 목적지(destination) 레지스터에 64비트 결과를 얻는 식으로 수행된다. 그러나 그림 1에서처럼 32비트 CPU가 8비트 CPU용 OEF의 두 원소 $A(x)$, $B(x)$의 곱셈을 할 때에는 소스 레지스터의 아래쪽 부분만을 사용하게 된다. 이 사실을 이용하면 그림 2에서처럼 한 번의 명령어 수행에서 두 개의 $GF(p)$ 곱셈 $a_{1,i} \times b_{1,j}$ 및 $a_{2,i} \times b_{2,j}$를 동시에 수행하는 새로운 곱셈 알고리즘(Algorithm 2)을 구성할 수 있다. 단, 소스 레지스터의 적당한 위치에 8비트 값들을 넣어주고 곱셈 명령 수행 후 다시 16비트 결과 값들을 추출해 내기 위해 Shift, And, Or 등의 연산들이 추가되어야 한다. 결국 원래 알고리즘에서 m^2 개의 Mult 명령을 m^2 개의 And, $2m$ 개의 Or, m^2+2m 개의 Shift 명령으로 대체하게 되는데, Shift 및 논리 연산들은 Mult 연산보다 훨씬 빠르므로, Algorithm 2가 원래의 곱셈 알고리즘을 2회 따로 수행하는 것보다 효율적일 것이라 기대할 수 있다.

Algorithm 2. parallel OEF multiplication
Input: $A_1(x), B_1(x), A_2(x), B_2(x) \in GF(p^m)$.
Output: $C_1(x) = A_1(x) \cdot B_1(x) \bmod f(x) \in GF(p^m)$,
$C_2(x) = A_2(x) \cdot B_2(x) \bmod f(x) \in GF(p^m)$
▷ $r_1, r_2, u, v, a_k, b_k, d_{1,k}, d_{2,k}$: 32비트 워드들
Step 0: For k from 0 to $m-1$ do ▷ Shift, Or 각 $2m$회
$r_1 \leftarrow \text{ShiftLeft}(a_{1,k}, 24)$; $a_k \leftarrow \text{Or}(a_{2,k}, r_1)$.
$r_2 \leftarrow \text{ShiftLeft}(b_{1,k}, 24)$; $b_k \leftarrow \text{Or}(b_{2,k}, r_2)$.
Step 1: For k from 0 to $2(m-1)$ do ▷ And, Shift 각 m^2회
$r_1 \leftarrow 0$; $r_2 \leftarrow 0$.
For each element of $\{(i,j) \mid i+j=k, 0 \le i,j < m\}$ do
$uv \leftarrow \text{Mult}(a_u, b_j)$.
$v \leftarrow \text{And}(v, 0000ffff_{16})$.
$u \leftarrow \text{ShiftRight}(u, 16)$.
$r_1 \leftarrow \text{Add}(r_1, u)$; $r_2 \leftarrow \text{Add}(r_2, v)$.
$d_{1,k} \leftarrow r_1$; $d_{2,k} \leftarrow r_2$.
Step 2: For k from 0 to $m-2$ do ▷ $x^m \equiv w \bmod f(x)$ 이용
$v \leftarrow \text{Mult}(d_{1,k+m}, w)$; $d_{1,k} \leftarrow \text{Add}(d_{1,k}, v)$.
$v \leftarrow \text{Mult}(d_{2,k+m}, w)$; $d_{2,k} \leftarrow \text{Add}(d_{2,k}, v)$.
Step 3: For k from 0 to $m-1$ do
$c_{1,k} \leftarrow d_{1,k} \bmod p$; $c_{2,k} \leftarrow d_{2,k} \bmod p$.

다음에는 OEF 상의 역원 연산을 고려해 보자. 2.3절에서 언급한 바와 같이, Algorithm 1의 핵심 연산은 A^{r-1}의 계산(Step 1)에서 필요한 OEF 곱셈들이며, 따라서 Step 1을 또 다른 OEF 원소 $A'(x)$에 대한 Step 1 계산 $(A')^{r-1}$과 병치하고 이들 두 계산을 병렬적으로 수행하는 데 Algorithm 2를 이용하면 A^{-1}과 $(A')^{-1}$, 두 개의 역원을 빠르게 구할 수 있다.

4. 효율적인 타원 곡선 연산

3절의 병렬 연산 알고리즘들을 이용하면, 타원 곡선 점의 덧셈 연산(ADD)과 2배 연산(DBL)을 병렬적으로 수행할 수 있다. 즉, 'ADD-and-DBL' 알고리즘(Algorithm 3)을 구성할 수 있다. (알고리즘 기술을 간단히 하기 위해 $P_1 \neq -P_2$ 및 $P_1, P_2, P_3 \neq O$를 가정하자. 점의 덧셈 및 2배 연산 공식은 [8]을 참조하라.)

Algorithm 3. ADD-and-DBL (parallel ADD and DBL)
Input: 곡선 $y^2 = x^3 + ax + b$ $(a, b \in GF(p^m))$ 상의 세 점
$P_1 = (x_1, y_1), P_2 = (x_2, y_2), P_3 = (x_3, y_3)$.
Output: $P_A = (x_A, y_A) = P_1 + P_2$ 및 $P_D = (x_D, y_D) = 2P_3$.
Step 1: $t \leftarrow 3x_3^2$.
Step 2: $t_A \leftarrow (x_2 - x_1)^{-1}$; $t_D \leftarrow (2y_3)^{-1}$.
Step 3: $\lambda_A \leftarrow (y_2 - y_1)t_A$; $\lambda_D \leftarrow (t + a)t_D$.
Step 4: $x_A \leftarrow \lambda_A^2 - x_1 - x_2$; $x_D \leftarrow \lambda_D^2 - 2x_3$.
Step 5: $y_A \leftarrow \lambda_A(x_1 - x_A) - y_1$; $y_D \leftarrow \lambda_D(x_3 - x_D) - y_3$.

Algorithm 3은 Step 1에서 OEF 제곱 연산을 1개 필요로 하며, 다른 Step들은 모두 병렬 OEF 연산을 이용할 수 있음을

3) $c_k, c \in GF(p)$이므로 Step 2는 하위체 곱셈 m개와 상수 w에 대한 곱셈 하나만으로, Step 4는 하위체 곱셈 m개만으로 계산 가능하다. 또한, Step 3은 $GF(p)$ 상의 확장 유클리드 알고리즘으로 간단히 계산된다.

알 수 있다. 즉, Step 2에서는 1회의 병렬 역원 계산, Step 4에서는 1회의 병렬 제곱, Step 3 및 5에서는 각각 1회씩의 병렬 곱셈이 필요하다.

ADD-and-DBL 알고리즘을 이용하면 다음과 같은 효율적인 NAF 스칼라 곱셈 알고리즘을 설계할 수 있다.

Algorithm 4. new NAF scalar multiplication
Input: 점 P 및 정수(스칼라) k.
Output: 점 $Q = kP$.
Step 1: k에 대한 NAF를 구성한다.
(즉, $k_i \in \{-1, 0, 1\}$이 되도록 $k = \sum_{i=0}^{l-1} k_i 2^i$를 구한다.)
Step 2: $S \leftarrow P$; $Q \leftarrow O$.
Step 3: For i from 0 to $l-1$ do
if ($k_i = 1$) then $(Q, S) \leftarrow$ ADD-and-DBL(Q, S, S).[4]
else if ($k_i = -1$) then $(Q, S) \leftarrow$ ADD-and-DBL($Q, -S, S$).
else $S \leftarrow$ DBL(S).

5. 성능 분석

이 절에서는 세 가지의 OEF에 대한 연산 속도 측정 결과를 제시하고, 이들을 이용하여 스칼라 곱셈을 구현했을 때의 성능 향상 정도를 예측한다. 본 논문에서 고려한 OEF들은 다음과 같다.

- OEF 1: $GF((2^7-1)^{21})$, $(f(x) = x^{21} - 3)$ [2]
- OEF 2: $GF((2^8-17)^{17})$, $(f(x) = x^{17} - 2)$ [8]
- OEF 3: $GF((2^8-5)^{25})$, $(f(x) = x^{25} - 3)$

또한, 위의 각 OEF들에 대해 다음 여섯 가지의 알고리즘을 구현하여 시간을 측정하였다.

- 곱셈: 기본 OEF 곱셈, 병렬 OEF 곱셈(Algorithm 2)
- 제곱: 기본 OEF 곱셈과 Algorithm 2의 변형
- 역원: Algorithm 1과 그 병렬 버전

표 1은 Pentium III 750MHz에서 Microsoft Visual C++ 6.0 (in-line assembler 포함)을 이용하여 실험한 결과를 보이고 있다. 이 CPU에서는 MMX 명령과 같은 특수한 형태의 64비트 명령어가 지원되지만, 32비트 아키텍처에서의 일반적인 비교 결과를 도출하기 위해 32비트 명령어만을 이용하였다.[5] 표 1에서 우리는 병렬 곱셈, 제곱, 역원 계산 알고리즘들이 성능[6]을 각각 40~52%, 31~37%, 28~40% 향상시키는 것을 확인할 수 있다.

마지막으로, 스칼라 곱셈의 소요 시간을 추정한 결과를 제시한다. 이것은 표 1에서 제시된 개별 OEF 연산들의 시간 측정값을 더하여 계산한 것이다. 본 논문에서는 지면 관계상 하나의 NAF 스칼라 곱셈이 수행되는 상황만을 고려하기로 한다.[7] 일단, l 자리 NAF의 경우 평균적으로 0이 아닌 숫자, 즉 ± 1은 총 $l/3$개가 존재하며, 따라서 원래의 NAF 스칼라 곱셈 알고리즘을 이용할 경우 l 개의 'DBL' 연산과 $l/3$ 개의 'ADD' 연산이 필요하다. 반면에 Algorithm 4의 경우 $2l/3$ 개의 'DBL' 연산과 $l/3$ 개의 'ADD-and-DBL' 연산이 필요하다. 표 2에 의하면 본 논문에서 제시한 병렬 연산에 의해 NAF 스칼라 곱셈의 성능이 12~16% 향상될 것으로 추정된다.

4) ADD-and-DBL을 사용하지 않을 경우 $Q \leftarrow$ ADD(Q, S) 및 $S \leftarrow$ DBL(S)로 따로 계산해야 한다.

5) Pentium MMX 명령어와 같은 SIMD 아키텍처를 이용하는 방법에 대해서는 [9]를 참고하라.

6) 여기서 '성능'은 '단위 시간당 연산 량'을 의미한다.

7) 많은 스칼라 곱셈이 동시에 수행되는 특수한 환경, 예를 들어 다자간 프로토콜(multi-party protocol)과 같은 경우에는 'ADD-and-DBL' 이외에 'ADD-and-ADD' 및 'DBL-and-DBL'과 같은 또 다른 병렬 연산들을 정의하여 응용할 수 있다. 결과적으로, 병렬 연산의 장점을 더 효율적으로 이용하게 됨으로써 스칼라 곱셈에서 34~45%의 성능 향상을 예상할 수 있다.

표 1 Pentium III에서 OEF 연산의 소요 시간 (μsec)
[a] : (병렬 연산의 소요 시간) / 2

	OEF 1	OEF 2	OEF 3
곱셈	13.02	9.45	17.76
병렬 곱셈(Alg.2)[a]	8.57	6.75	12.44
성능 향상	51.9%	40.0%	42.8%
제곱	10.13	7.81	13.80
병렬 제곱[a]	7.38	5.92	10.51
성능 향상	37.3%	31.9%	31.3%
역원(Alg.1)	78.56	49.68	108.45
병렬 역원[a]	56.15	38.83	81.64
성능 향상	39.9%	27.9%	32.8%

표 2 타원 곡선 점 연산 및 l 자리 NAF 스칼라 곱셈의 소요 시간에 대한 추정값 (μsec)

	OEF 1	OEF 2	OEF 3
독립적인 ADD	114.73	76.39	157.77
독립적인 DBL	124.86	84.20	171.57
ADD-and-DBL (Alg.3)	171.47	124.31	247.86
기본 NAF 스칼라 곱셈	163.10l	109.66l	224.16l
변형된 NAF 스칼라 곱셈(Alg.4)	140.39l	97.57l	197.00l
성능 향상	16.2%	12.4%	13.8%

6. 결론

본 논문에서는 하나의 정수 곱셈 명령어 내에서 두 개의 하위체 곱셈을 동시에 수행하는 새로운 OEF 곱셈 알고리즘을 제안하였다. 또한, OEF 상의 역원 계산이 실질적으로 OEF 곱셈의 반복으로 구성되므로, 제안된 알고리즘을 역원 계산에도 응용할 수 있었다. 본 논문의 실험 결과에 따르면 Pentium III에서 새로운 알고리즘을 이용할 경우 OEF 연산에서 30~50% 정도의 성능 향상을 얻을 수 있었으며, 이를 타원 곡선의 스칼라 곱셈에 적용할 경우 12~16% 정도의 성능 향상을 추정할 수 있었다.

참고문헌

[1] D.V.Bailey and C.Paar, Optimal extension fields for fast arithmetic in public key algorithms, Crypto '98, LNCS 1462, pp.472~485, Springer-Verlag, 1998.
[2] D.V.Bailey and C.Paar, Efficient arithmetic in finite field extensions with application in elliptic curve cryptography, Journal of Cryptology 14(3), pp.153~176, 2001.
[3] N.P.Smart, A comparison of different finite fields for elliptic curve cryptosystems, Computers and Mathematics with Applications 42, pp.91~100, 2001.
[4] D.Knuth, The art of computer programming. volume 2: seminumerical algorithms, Addison-Wesley, Reading, Massachusetts, 3rd edition, 1998.
[5] C.H.Lim and H.S.Hwang, Fast implementation of elliptic curve arithmetic in $GF(p^n)$, PKC 2000, LNCS 1751, pp.405~421, Springer-Verlag, 2000.
[6] T.Kobayashi, Base-ϕ method for elliptic curves over OEF, IEICE Trans. Fundamentals E83-A(4), pp.679~686, 2000.
[7] J.A.Solinas, An improved algorithm for arithmetic on a family of elliptic curves, Crypto '97, LNCS 1294, pp.357~371, Springer-Verlag, 1997.
[8] A.D.Woodbury, D.V.Bailey, and C.Paar, Elliptic curve cryptography on smart cards without coprocessors, CARDIS 2000, pp.71~92, 2000.
[9] K.Aoki, F.Hoshino, T.Kobayashi, and H.Oguro, Elliptic curve arithmetic using SIMD, ISC 2001, LNCS 2200, pp.235~247, Springer-Verlag, 2001.