확장 이진 유한체 제곱근 연산의 최적 구현 기법에 관한 연구

박민진 오진석 인재휘 전창열 김동찬

국민대학교

A Study on Optimal Implementation of Square Root in Extended Binary Field

MinJin Park, JinSeok Oh, JaeHui In, ChangYeol Jeon, Dong-Chan Kim

Kookmin University

요 약

확장 이진 유한체의 제곱근 연산은 사전계산 테이블 참조 방식의 지수승 연산으로 계산할 수 있다. Takuya Sumi 등은 이진 유한체의 성질을 이용하여 보다 효율적으로 제곱근을 계산하는 방식을 사용하였다. 본 논문에서는 두 제곱근 연산 방식인 지수승 기반 방식과 Takuya Sumi 등의 방식을 소개하고, 두 연산을 C언어로 구현하여 시간과 메모리 관점에서 성능을 비교 분석한다.

I. 서론

확장 이진 유한체의 원소의 제곱근은 항상 존재하고, 사전계산 테이블 참조 방식의 지수승 연산으로 계산할 수 있다. Takuya Sumi 등은 이진 유한체의 성질을 이용하여 보다 효율적으로 제곱근을 계산하였다[2]. 해당하는 두 가지 제곱근 연산에 대해 [1]에서는 FLINT로 구현한 후 연산 시간과 메모리를 비교하였다. 구현 시 FLINT를 이용했기 때문에 최적화 과정에서 한계를 보였다.

본 논문에서는 [1]에서 사용한 두 제곱근 연산인 지수승 기반 제곱근 연산과 Takuya Sumi 등의 제곱근 연산을 소개한다. 그리고 두 연산을 C언어로 구현 시 진행한 최적 구현 기법에 대해 설명한다. 이후 구현한 연산을 시간과 메모리 관점에서 [1]과 비교한다. 비교는 Classic McEliece에서 제안한 4개의 파라미터에 대해 이루어진다. C언어로 구현 시 FLINT를 이용한 [1]에 비해 지수승 기반 제곱근 연산 기준 최대 122배, Takuya Sumi 등의 제곱근 연산 기준 최대 33배 빠르게 동작하였다.

본 논문의 구성은 다음과 같다. Ⅱ절에서는 기호를 정의한다. Ⅲ절에서는 [1]에서 사용한 두 알고리듬에 대해 소개한다. Ⅳ절에서는 두 알고리듬에 적용한 최적 구현 기법에 대해 설명한다. Ⅴ절에서는 C언어로 구현한 두 연산의 시간과 메모리를 측정하고 [1]의 결과와 비교한다.

II. 기호

본 논문에서 사용하는 기호는 다음과 같다.

$-F_{2^m}$	2^m개의 원소를 가진 유한체
$-f(s)(=\sum_{i=0}^{m} z_i s^i)$	유한체 F_{2^m}의 생성 기약 다항식 ($z_i \in F_2$)
$-F_{2^m}[X]$ $-g(X) \in F_{2^m}[X]$	F_{2^m}으로 정의한 다항식 환 t차 기약다항식
$-F_{2^m}[X]/(g(X))$	$g(X)$로 정의한 2^{mt}개의 원소를 가진 유한체
$-\sqrt{\alpha}$	$\alpha(\in F_{2^m})$의 제곱근
$-\sqrt{A(X)}$	$A(X)(\in F_{2^m}[X]/(g(X)))$의 제곱근

III. 제곱근 연산

$F_{2^m}[X]/(g(X))$의 원소 $A(X)=\sum_{i=0}^{t-1} a_i X^i$는 다음 두 연산으로 제곱근을 계산할 수 있다.

$$\sqrt{A(X)} = A(X)^{2^{mt-1}} \quad \text{(식 1)}$$

$$= \sum_{i=0}^{t-1} \sqrt{a_i X^i}\ . \quad \text{(식 2)}$$

(식 1) 이용 시 다항식 $A(X)$에 대한 제곱 모듈러 연산을 $mt-1$회 하는 것으로 제곱근을 구할 수 있다. 이때 사전계산 테이블 T, S를 이용한다. T에는 F_{2^m}의 모든 원소를 제곱한 결과를 저장하고, S에는 $\lfloor (t+1)/2 \rfloor \le i \le t-1$에 대해 $X^{2i} \bmod g(X)$를 저장한다. 이 방식의 수행과정은 알고리듬 1과 같다.

알고리듬 1: (식 1)을 이용한 제곱근 연산
입력: $A(X) = \sum_{i=0}^{t-1} a_i X^i \in F_{2^m}[X]/(g(X))$, 사전계산 테이블 T, S
출력: $\sqrt{A(X)}$
1. $k \leftarrow \lfloor (t-1)/2 \rfloor$
2. for $j = mt-2$ downto 0 do
3. $\sum_{i=0}^{t-1} a_i X^i \leftarrow \sum_{i=0}^{k} T[a_i] X^{2i} + \sum_{i=k+1}^{t-1} T[a_i] S[i]$
4. return $\sum_{i=0}^{t-1} a_i X^i$

(식 2)는 Takuya Sumi 등이 이용한 제곱근 연산 방법에서 이용한다. 이때 사전계산 테이블 $\widetilde{T}, \widetilde{S}$를 사용한다. $\widetilde{T}$에는 F_{2^m}의 모든 원소의 제곱근을 저장하고, $\widetilde{S}$에는 $0 \le i \le \lfloor t/2 \rfloor - 1$에 대해 $X^i \sqrt{X} \bmod g(X)$를 저장한다. 이 방식의 수행과정은 알고리듬 2와 같다.

알고리듬 2: (식 2)를 이용한 제곱근 연산
입력: $A(X) = \sum_{i=0}^{t-1} a_i X^i \in F_{2^m}[X]/(g(X))$, 사전계산 테이블 $\widetilde{T}, \widetilde{S}$
출력: $\sqrt{A(X)}$
1. $A'(X) \leftarrow \sum_{i=0}^{\lfloor (t-1)/2 \rfloor} \widetilde{T}[a_{2i}] X^i$
2. $A''(X) \leftarrow \sum_{i=0}^{\lfloor t/2 \rfloor - 1} \widetilde{T}[a_{2i+1}] \widetilde{S}[i]$
3. return $A'(X) + A''(X)$

IV. 구현

알고리듬 구현을 위해 다항식을 구조체로 저장한다. 구조체는 다항식의 계수와 최고차수로 구성된다. 이때 다항식의 최고차수는 int형 변수로 설정하고, 다항식의 계수는 int형 배열을 이용하여 F_{2^m}의 원소 하나를 4바이트에 저장한다.

두 알고리듬은 사전계산을 통한 테이블 참조 이외에 유한체 곱셈, 다항식 덧셈으로 구성된다.

유한체 곱셈에서는 사전계산 테이블 R을 사용한다. $m \le i \le 2m-1$에 대해 $s^i \bmod f(s)$를 계산하여 R에 저장한다. 이후 과정에서 유한체 곱셈 시 발생하는 모듈러 연산은 테이블 참조로 처리하였다.

다항식 덧셈은 동일한 차수에 대해 F_{2^m}상에서의 덧셈이다. 해당 연산은 입력받은 다항식의 계수에 대해 XOR 연산을 하는 것으로 처리하였다.

V. 측정 결과

실험 환경은 다음과 같다.

- 하드웨어 환경: 1.4GHz 쿼드 코어 Intel Core i5, 8GB RAM, macOS Big Sur 버전 11.0.1
- 컴파일러: Apple clang version 12.0.0 (clang-1200.0.32.27) 컴파일 옵션 -O2

Classic McEliece에서 제안한 파라미터에 대해 사용한 테이블의 메모리, 연산 수행 횟수(테이블 참조 횟수, XOR 연산 횟수), 연산 시간을 측정하였다.

사용한 사전계산 테이블은 $T, \widetilde{T}, S, \widetilde{S}, R$이며 구현에서 사용한 총 메모리는 다음과 같다. $T, \widetilde{T}$테이블은 각각 F_{2^m}의 원소를 2^m개 저장한다. F_{2^m}의 각 원소는 4바이트의 공간을 차지하므로 각 테이블은 2^{m+2}바이트의 메모리를 사용한다. $S, \widetilde{S}$테이블은 각각 $F_{2^m}[X]/(g(X))$의 원소를 $\lfloor t/2 \rfloor$개 저장한다. 각 원소는 곱셈을 고려하여 계수를 최대 $2t$개로 설정한다. 따라서 테이블 당 $2t \cdot \lfloor t/2 \rfloor \cdot 2^2$바이트의 메모리를

사용한다. R테이블은 F_{2^m}의 원소를 m개 저장한다. 그러므로 2^2m 바이트의 메모리를 사용한다. 최종적으로 각 알고리듬에서 사용하는 메모리는 다음과 같다.

$$(2^m + 2t\lfloor t/2 \rfloor + m) \cdot 2^2$$

연산 수행 횟수는 $A(X)$에 따라 달라진다. 그러므로 임의의 $A(X)$를 1000회 생성하여 측정한 후 평균값으로 나타내었다. 메모리와 연산 수행 횟수는 [표 1]과 같다. 사용한 메모리의 단위는 바이트(byte) 이다. 또한 XOR 횟수는 4바이트에 대한 연산 횟수이다.

[표 1] 알고리듬 연산 수행 횟수 및 메모리

파라미터 (m, t)	Mceliece348864 (12, 64)		Mceliece460896 (13, 96)	
알고리듬	1	2	1	2
참조횟수	1,330,541	12,376	4,716,419	30,051
XOR횟수	25,422,382	88,217	98,334,238	214,094
메모리	32,816		69,684	
파라미터 (m, t)	Mceliece6960119 (13, 119)		Mceliece6688128 (13, 128)	
알고리듬	1	2	1	2
참조횟수	9,260,570	45,817	17,623,528	53,504
XOR횟수	186,319,780	329,093	270,897,271	380,841
메모리	88,988		98,356	

FLINT로 구현한 [1]과 C언어로 구현한 알고리듬의 연산 시간은 [표 2]와 같다. 1000회 동작 시 걸린 시간의 평균으로 나타내었고, 사용한 시간 단위는 밀리초(ms)이다.

[표 2] 알고리듬 연산 시간 측정결과 (단위: ms)

파라미터 (m, t)	알고리듬	FLINT [1]	C(본 논문)
Mceliece348864 (12, 64)	1	2,984.971	24.294
	2	2.878	0.086
Mceliece460896 (13, 96)	1	6,320.881	100.715
	2	4.182	0.218
Mceliece6960119 (13, 119)	1	11,400.386	173.928
	2	6.472	0.284
Mceliece6688128 (13, 128)	1	14,497.782	292.432
	2	7.904	0.393

연산 시간에 대한 신뢰성을 확인하기 위해 C언어로 구현한 두 알고리듬의 수행 횟수와 연산 시간에 대한 비율을 확인하였다. 이는 [표 3]과 같다. 비는 수행 횟수 비율을 연산 시간 비율로 나눈 값이고, 각각의 비율은 알고리듬 1에 대한 측정값을 알고리듬 2에 대한 측정값으로 나눈 값이다.

[표 3] 파라미터별 수행 횟수 및 연산 시간 비율

파라미터 (m,t)	수행 횟수	연산 시간	비
Mceliece348864 (12, 64)	265	282	0.93
Mceliece460896 (13, 96)	422	461	0.91
Mceliece6960119 (13, 119)	521	612	0.85
Mceliece6688128 (13, 128)	664	744	0.89

VI. 결론

본 논문에서는 두 알고리듬을 C언어로 구현한 결과와 FLINT로 구현한 [1]의 결과를 비교하였다. 결과적으로 연산 시간이 알고리듬 1 기준 최대 122배, 알고리듬 2 기준 최대 33배 줄었다. 또한 파라미터 별로 연산 수행 횟수 비율과 연산 시간 비율을 확인한 결과 두 비율이 유사함을 확인하였다.

추후에는 SIMD를 이용한 병렬연산으로 제곱근 연산의 고속 구현 기법에 대해 연구할 예정이다.

[참고문헌]

[1] 전창열, 박민진, 오진석, 인재휘 and 김동찬. "확장 이진 유한체 제곱근 연산의 효율적 구현에 관한 연구." 한국통신학회 동계종합학술발표회. 2021.

[2] Sumi Takuya, Morozov Kirill, and Takagi Tsuyoshi. Efficient implementation of the McEliece cryptosystem. In computer security symposium 2011, volume 2011, pages 582-586, oct 2011.