creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

동형암호 컴파일러를 위한 다항식 최적화 알고리즘

Optimizing Algorithm of Polynomial Evaluation for CKKS Homomorphic Compiler

2023 년 2 월

서울대학교 대학원

컴퓨터 공학부

박 규 연

공학석사학위논문

동형암호 컴파일러를 위한 다항식 최적화 알고리즘

Optimizing Algorithm of Polynomial Evaluation for CKKS Homomorphic Compiler

2022 년 12 월

서울대학교 대학원

컴퓨터 공학부

박 규 연

동형암호 컴파일러를 위한 다항식 최적화 알고리즘

Optimizing Algorithm of Polynomial Evaluation for CKKS Homomorphic Compiler

지도교수 이 광 근

이 논문을 공학석사학위논문으로 제출함

2022 년 12 월

서울대학교 대학원

컴퓨터 공학부

박 규 연

박규연의 석사학위논문을 인준함

2023 년 1 월

위 원 장	권태경	(인)
부위원장	이광근	(인)
위 원	허충길	(인)

요약

동형암호 컴파일러에 장착될 다항식 최적화 알고리즘을 개발하고 실험을 통해 최적화 성능을 측정하였다. 딥뉴럴 네트워크 모델의 작성에 자주 사용되는 수학 함수들의 근사다항식에서 비용이 큰 곱셈 연산의 횟수가 연산 순서를 다르게 함에 따라 달라지는 것에 착안하여 다항식의 곱셈 횟수를 최소화하는 최적화 알고리즘을 고안하였다. 본 알고리즘은 차수가 작은 다항식의 계산결과를 통해 큰 다항식의 결과를 계산하는 동적 프로그래밍 기반의 방법으로, 이미 계산한 곱셈의 횟수를 중복으로 계산하지 않으며 모든 경우를 다 탐색하여 저장하여 활용도가 높아진다는 장점이 있다. 본 알고리즘은 적당한 차수의 다항식에서 기존 방법에 비해 최대 33%로 곱셈 횟수를 감소시켜 성능을 개선하였다. 그러나 적당한 차수에서는 의미있는 결과를 보여주지만, 차수가 올라감에 따라서 연산횟수가 지수적으로 늘어난다.

주요어: 동형 컴파일러, 다항식 최적화, 알고리즘

학번: 2021-23580

목차

요약 **i**

목차 **ii**

그림 목차 **iv**

표 목차 **v**

제 1 장 서론 **1**

제 2 장 최적화 대상 문제 **4**

2.1 입력 언어 H 4

2.2 도착 언어 HOS 4

2.3 최적화 기준 5

제 3 장 최적화 알고리즘 **8**

3.1 알고리즘 설명 8

3.2 알고리즘 증명 10

3.2.1 알고리즘 귀납 증명 10

3.2.2 알고리즘 커버 영역 10

제 4 장 실험 결과 **14**

4.1 벤치마크 소개 14

4.2 실험 결과 14

제 5 장 논의 및 결론 **18**

5.1 결론 18

5.2 알고리즘의 한계점과 해결방안 18
5.2.1 알고리즘 부분 적용 18
5.2.2 곱셈 깊이 감소 18

참고문헌 **19**

Abstract **21**

그림 목차

그림 1.1 동일한 다항식이지만 곱셈횟수가 달라지는 코드 예시 2

그림 2.1 H언어의 문법구조 정의 6

그림 2.2 HOS언어의 문법구조 정의 7

그림 4.1 ReLU 연산의 근사다항식 16

그림 4.2 Sigmoid 연산의 근사다항식 16

그림 4.3 SigmoidWide 연산의 근사다항식 17

그림 5.1 Sign 연산의 근사다항식 19

표 목차

표 4.1 실험 결과 15

제 1 장 서론

본 논문에서는 동형암호 컴파일러에 장착될 다항식 최적화 알고리즘을 소개하고, 실험을 통해 최적화가 어떻게 이루어지는지 확인한다.

동형암호 기술은 데이터를 암호화시킨 상태에서도 임의의 연산이 가능하도록 해주는 기술로서, 보안이 중요한 데이터의 보관 또는 연산을 제3자가 해도 안전하게 보호해주는 차세대 암호 기술이다.[1] 여타 암호 기술들과 다른 점은 동형암호 틀 위에서는 평문 데이터 x와 y가 있을때 이를 동형암호화한 $\underline{x}$, $\underline{y}$끼리 동형연산인 $\underline{+}$, $\underline{\times}$가 가능하다. 그리고 이 연산 결과를 복호화하면 평문인 x, y위에서 평문 연산자인 +, ×을 해준 결과와 각각 같아진다. ($d(\underline{x} \underline{+} \underline{y}) = x + y$, $d(\underline{x} \underline{\times} \underline{y}) = x \times y$, d는 복호화). 여기에서 연산자 +와 ×은 논리회로의 AND, XOR 연산자와 각각 대응되고 이 두 가지 연산자만으로 컴퓨터는 수행가능한 모든 연산을 구현할 수 있게 된다.

기존에는 비트나 정수 형태의 데이터에서만 동형암호 연산이 가능했지만 최근에 4세대 CKKS 동형암호 틀이 개발되어 실수 데이터의 연산도 가능해지며 이를 통해 딥뉴럴 네트워크 모델도 동형암호 연산이 가능해지게 되었다. [2]

동형암호 컴파일러는 동형암호에 대한 전문지식이 없는 사용자도 쉽게 작성할 수 있는 코드를 동형연산이 수행하는 동형암호 코드로 변환해준다.[3] 본 연구실에서 개발한 동형암호 컴파일러는 이전 동형 컴파일러에 비해 4세대 동형암호 틀을 지원하기 때문에 실수 연산을 사용하는 통계 연산이나 딥뉴럴 네트워크 연산을 수행하는 코드를 동형암호 코드로 변환할 수 있다. 이 동형 컴파일러는 딥뉴럴 네트워크 모델에서 사용하는 핵심 텐서 연산자를 포함하도록 정의된 상위 언어인 H를 입력언어로, 4세대 CKKS 동형암호틀인 HEaaN을 기반으로한 동형연산용 하위 언어인 HOS를 도착언어로 사용한다.

딥뉴럴 네트워크 모델을 구현한 동형암호 코드에서 자주 등장하는 math oper-

ation들은 곱셈과 덧셈 연산만을 사용한 근사다항식의 형태로 풀어쓰게 된다. 곱셈 연산을 할 때 드는 비용은 덧셈 연산 비용의 약 13배이므로 곱셈 횟수를 줄이는 것이 코드 최적화에 주요한 영향을 미치게 된다. 이러한 필요성에 따라 다항식의 곱셈 연산 순서를 바꾸어서 곱셈 횟수를 최소화하는 최적화 알고리즘을 고안하게 되었다.

$$
\begin{aligned}
&\text{let } x^2 = x * x \text{ in} \\
&\text{let } x^4 = x^2 * x^2 \text{ in} \\
&\text{let } x^8 = x^4 * x^4 \text{ in} \\
&a_8x^8 + x^4 * (x^2 * (a_7x + a_6) + a_5x + a_4) + x^2 * (a_3x + a_2) + a_1x + a_0
\end{aligned}
$$

$$
\begin{aligned}
&\text{let } x^2 = x * x \text{ in} \\
&\text{let } x^3 = x^2 * x \text{ in} \\
&\text{let } x^4 = x^2 * x^2 \text{ in} \\
&x^4 * (a_8x^4 + a_7x^3 + a_6x^2 + a_4) + a_3x^3 + a_2x^2 + a_1x + a_0
\end{aligned}
$$

그림 1.1 동일한 다항식이지만 곱셈횟수가 달라지는 코드 예시

그림 1.1의 두 코드는 동일한 의미를 가지는 다항식이지만 곱셈 횟수가 서로 다르다. 위쪽 코드는 곱셈을 총 5번 수행하지만 아래쪽 코드는 곱셈을 총 4번 수행한다. 곱셈 횟수는 예시와 같이 다항식의 연산 순서에 따라 달라지게 된다. 곱셈 횟수가 서로 다른 것에는 먼저 x의 n제곱 형태인 변수를 어떻게 선언하느냐와, 선언한 변수를 어떻게 묶어서 표현하느냐 이 두 가지가 영향을 미치게 된다. 따라서 알고리즘에서는 이 두 가지의 각각의 경우에 대해 탐색하여 어떤 경우가 최소인지 저장하는 방식으로 최적화가 가능하도록 구현한다.

본 연구에서는 다항식의 곱셈 횟수를 최소화하는 알고리즘을 구현하고 비용을 측정하는 실험을 통하여 이것이 얼마나 효과가 있는지 확인해본다. 이 알고리즘은 다항식 최적화를 위해 논의되었던 기존의 다른 방식에 비해 다항식의 모든 경우를 탐색하며 이미 선언한 곱셈의 횟수를 추가로 더 계산하지 않는다.

본 논문은 서론을 포함하여 총 5장으로 구성된다. 먼저 2장에서는 최적화 대상이 되는 문제 상황에 대해 자세히 정의하고 최적화 기준에 대해 설명한다. 3장에

서는 다항식 최적화 알고리즘의 구현과 알고리즘의 적용이 어떻게 이루어지는지 자세히 소개한다. 4장에서는 알고리즘 적용전후의 코드 성능을 비교하여 알고리즘의 효용성에 대해 살펴본다. 5장에서는 본 연구의 한계와 추후 연구방향과 결론에 대해 논의한다.

제 2 장 최적화 대상 문제

2.1 입력 언어 H

동형 컴파일러의 입력언어 H는 딥뉴럴 네트워크 모델에서 자주 쓰이는 convolution, maxpool, sigmoid 등의 텐서 연산을 표현할 수 있는 상위 언어이다. 동형암호에 대한 전문지식이 없이도 쉽게 코드를 작성할 수 있게 정의되었다. 그림 2.1의 H의 문법구조에 대해 보면, 외부 파일의 텐서 입력 및 let을 이용한 변수선언문과 딥뉴럴 네트워크에 자주 쓰이는 텐서 연산자 구문인 tensor operation(top)로 이루어져있다. 또한 해당 텐서가 입력될 때 암호문 상태로 연산이 이루어지는지 여부를 annot을 통해 나타낼 수 있게 하였다.

2.2 도착 언어 HOS

동형 컴파일러의 도착언어 HOS는 4세대 CKKS 동형암호 틀인 HEaaN을 기반으로 하여 정의된 하위 언어이다. 그림 2.2의 HOS의 문법구조를 보면, 외부에서 불러온 텐서 데이터를 암호로 인코딩하여 입력으로 받는 구조 및 let을 이용한 변수선언문과 다양한 동형암호 연산으로 이루어진다. arith operation(arithop)와 aux operation(auxop)은 덧셈, 뺄셈, 곱셈, 로테이션 등 기본연산 구문이고 crypt operation(cryptop)은 암호화, 복호화 및 안전성을 위한 연산과 암호문 곱셈 시 수행해야하는 linearize, rescale 등의 연산과 각각의 암호문의 레벨을 관리하는 bootstrap, relevel 등의 연산으로 이루어진다. math operation (mathop)은 딥뉴럴 네트워크에서 주로 쓰이는 수학 함수 연산자들로 이루어져있다. HOS에서는 동형암호 틀을 구성하는 핵심 연산자인 덧셈, 곱셈, 로테이션 연산을 기반으로 하며 딥뉴럴 네트워크에서 자주 쓰이는 math operation들이 근사다항식 형태로 제공되고 있다. HEaaN 틀에서 지원하는 곱셈, 덧셈, 로테이션 만으로는 math operation들의 정확한 값을 계산하지 못하므로 이를 근사하는 다항식의 형태로

연산을 수행한다.

2.3 최적화 기준

입력언어 H를 도착언어 HOS로 번역하는 과정에서 crypt operation(cryptop)을 제외한 HOS–언어를 중간언어 단계로 고안하였다. 번역 과정은 다음과 같다. H 코드를 암호적 안전성을 고려하지 않고 의미구조에 알맞게 HOS– 코드로 번역하고, HOS– 코드 내의 math operation들을 곱셈과 덧셈 연산자만을 사용한 근사다항식의 형태로 풀어쓴 다음, cryptop 연산자들을 자동으로 삽입하는 과정을 거쳐 HOS 코드로 번역한다.

근사다항식에 있는 곱셈 연산을 사용할 때마다 사용되는 암호문 연산인 cryptop 중 linearize 연산은 동형 덧셈 연산의 약 13배로 매우 높은 비용을 요구한다. 최종 HOS 코드에는 cryptop 연산자들이 삽입되므로 linearize 횟수를 줄이는 것이 결과물 코드의 성능을 개선하는데에 기여할 수 있다. 따라서 본 최적화의 기준은 최종 HOS 코드의 성능으로 측정한다.

동형 컴파일러에 의해 번역된 HOS 코드는 동형암호 틀인 HEaaN 기반의 실행기인 pi-HEaaN을 사용하여 실행을 해볼 수 있다. 실행한 결과를 통해 코드의 성능을 평가할 수 있다.

program	::=	e	program
e	::=	n *annot*?	constant number
	|	`[`e^+`]` *annot*?	constant tensor or constant list
	|	x *annot*?	variable
	|	`let` *vardecs* `in` e	variable binding
	|	`input` *annot*	input(shape is known)
	|	`<`*filename*`>(`*tensorname*`)` *annot*	stored tensor weight(shape is known)
	|	`(`e`)`	
	|	*top* e^+	tensor operation
annot	::=	*shape-annot*	
	|	*cipher-annot*	
	|	*shape-annot* *cipher-annot*	
shape-annot	::=	`:[`$n^{+,}$`]`	tensor shape, comma separated natural numbers
cipher-annot	::=	`:cipher`	to be encrypted
vardecs	::=	x `=` e	single declaration
	|	x `=` e, *vardecs*	multiple declaration

top	::=	`+` | `-` | `*`	: *Tensor* × *Tensor* → *Tensor*
	|	`tmul`	: *Tensor* × *Tensor* → *Tensor*
	|	`reshape`	: *Tensor* × *Shape* → *Tensor*
	|	`transpose`	: *Tensor* × *Axis* × *Axis* → *Tensor*
	|	`expand`	: *Tensor* × *Shape* → *Tensor*
	|	`concat`	: *Tensor* × *Axis* × *Tensor* → *Tensor*
	|	`max`	: *Tensor* → *Num*
	|	`maxr`	: *Tensor* × *Axis* → *Tensor*
	|	`argmaxr`	: *Tensor* × *Axis* → *Tensor*
	|	`sumr`	: *Tensor* × *Axis* → *Tensor*
	|	`convol`	: *Tensor* × *Tensor* × *Nat* × *Nat* → *Tensor*
	|	`maxpool`	: *Tensor* × *Shape* → *Tensor*
	|	`avgpool`	: *Tensor* × *Shape* → *Tensor*
	|	`padding`	: *Tensor* × *Nat* × *Nat* → *Tensor*
	|	`relu`	: *Tensor* → *Tensor*
	|	`sigmoid`	: *Tensor* → *Tensor*
	|	`tanh`	: *Tensor* → *Tensor*
	|	`inverse`	: *Tensor* → *Tensor*
	|	`sqrt`	: *Tensor* → *Tensor*

그림 2.1 H언어의 문법구조 정의

program	::=	e	program
e	::=	n	constant nat
	|	r	constant real
	|	*complex* e e	constant complex
	|	*message* e+	constant message (complex vector)
	|	<*filename*> (*vectorname*)	
	|	x	
	|	`let` $x = e$ `in` e	plaintext variable binding
	|	`let` x : *cipher* $= e$ `in` e	ciphertext variable binding
	|	*hos* e+	
hos	::=	*arithop*	
	|	*cryptop*	
	|	*auxop*	
	|	*mathop*	
mathop	::=	`aSign`	: *ctxt*(`l`) → *ctxt*(`m`)
	|	`aEqZero`	: *ctxt*(`l`) → *ctxt*(`m`)
	|	`aCompare`	: *ctxt*(`l`) × *ctxt*(`l'`) → *ctxt*(`m`)
	|	`aEq`	: *ctxt*(`l`) × *ctxt*(`l'`) → *ctxt*(`m`)
	|	`aSigmoid`	: *ctxt*(`l`) × *nat* → *ctxt*(`m`)
	|	`aReLU`	: *ctxt*(`l`) → *ctxt*(`m`)
	|	`aSqrt`	: *ctxt*(`l`) × *nat* → *ctxt*(`m`)
	|	`aInverse`	: *ctxt*(`l`) × *real* × *nat* → *ctxt*(`m`)
arithop	::=	`negate`	: *ctxt*(`l`) → *ctxt*(`l`)
	|	`add` | `sub`	: *ctxt*(`l`) × *ctxt*(`l`) → *ctxt*(`l`)
	|	`addr` | `subr`	: *ctxt*(`l`) × *real* → *ctxt*(`l`)
	|	`addc` | `subc`	: *ctxt*(`l`) × *complex* → *ctxt*(`l`)
	|	`addm` | `subm`	: *ctxt*(`l`) × *message* → *ctxt*(`l`)
	|	`addp` | `subp`	: *ctxt*(`l`) × *plain*(`l`) → *ctxt*(`l`)
	|	`addi` | `subi`	: *ctxt*(`l`) × *ipher*(`l`) → *ipher*(`l`)
	|	`addj` | `subj`	: *ctxt*(`l`) × *jpher*(`l`) → *jpher*(`l`)
	|	`prod`	: *ctxt*(`l`) × *ctxt*(`l`) → *ipher*(`l`)
	|	`prodr`	: *ctxt*(`l`) × *real* → *jpher*(`l`)
	|	`prodc`	: *ctxt*(`l`) × *complex* → *jpher*(`l`)
	|	`prodm`	: *ctxt*(`l`) × *message* → *jpher*(`l`)
	|	`prodp`	: *ctxt*(`l`) × *plain*(`l`) → *jpher*(`l`)
cryptop	::=	`encrypt`	: *message* → *ctxt*(`L`)
	|	`decrypt`	: *ctxt*(`l`) → *message*
	|	`encode`	: *message* × *nat*(`l`) → *plain*(`l`)
	|	`decode`	: *plain*(`l`) → *message*
	|	`linearize`	: *ipher*(`l`) → *jpher*(`l`)
	|	`rescale`	: *jpher*(`l`) → *ctxt*(`l-1`)
	|	`relevel`	: *ctxt*(`l`) × *nat*(`c`) → *ctxt*(`l-c`)
	|	`bootstrap`	: *ctxt*(`4`) × *nat*(`c`) → *ctxt*(`L-k`)
auxop	::=	`rotater`	: *ctxt*(`l`) × *nat* → *ctxt*(`l`)
	|	`rotatel`	: *ctxt*(`l`) × *nat* → *ctxt*(`l`)
	|	`conjugate`	: *ctxt*(`l`) → *ctxt*(`l`)

그림 2.2 HOS언어의 문법구조 정의

제 3 장 최적화 알고리즘

3.1 알고리즘 설명

본 다항식 최적화 알고리즘의 목표는 2장에서 설명했던 HOS– 코드의 math operation을 근사다항식으로 나타내었을때 해당 다항식의 곱셈 횟수를 최소화하여 비용이 큰 linearize 연산 수를 최소화하는 것이다. 곱셈 횟수는 곱셈 연산 순서를 어떻게 하느냐에 따라 달라지게 된다.

본 알고리즘에서는 작은 차수의 다항식에서 필요한 곱셈 횟수와 곱셈 연산 순서를 기록하여 큰 차수의 다항식에서 필요한 최소 곱셈 횟수와 곱셈 연산 순서를 계산한다. 즉 큰 문제를 작은 문제 여러 개로 나누어 해결하는 동적 프로그래밍 기반의 알고리즘이다. [4] math operation의 근사다항식들은 모두 정수가 아닌 실수 계수를 가지고 있기 때문에 계수를 이용하여 곱셈 연산 수를 줄일 수 있는 것은 특수한 경우에만 해당한다고 판단하였다. 따라서 모든 항의 계수를 1로 두고 해당 차수가 있는지 없는지만 저장하는 방식으로 다항식을 표현하였다.

해당 다항식은 크게 두 가지 단계로 만들어진다. 아래의 Algorithm의 의사 코드를 보면 알고리즘의 function이 correct_base와 reduce_order 이렇게 두 개이다. 먼저 x의 n제곱 형태인 재료를 여러 개 만든 다음, 만든 재료만을 이용하여 덧셈을 하여 묶음을 만들거나 묶음끼리 곱셈을 하는 방식으로 다항식을 완성한다.

첫 번째 단계에서 x의 n제곱 형태인 재료를 여러 개 만들 때는 우선 다항식의 차수 N이하의 재료의 집합의 부분집합 형태로 재료의 집합을 2^{N+1}개 만든 다음 유효한 재료집합만을 남긴다. 유효한 재료 집합이란 각 모든 재료가 집합의 다른 재료들을 이용하여 만들 수 있는 재료인 집합을 말한다. 예를 들어 $\{x, x^2, x^3, x^5, x^7\}$는 유효한 재료 집합이지만 $\{x, x^2, x^5, x^9\}$는 x^9가 나머지 재료들을 이용하여 곱셈 한 번만으로 만들어지지 않으므로 유효한 재료 집합이 아니다. Algorithm 1에 correct_base함수에서 유효한 재료 집합인지 아닌지 확인한다.

두 번째 단계인 reduce_order함수에서는 재료 파트와 재료들의 덧셈으로 이루어진 묶음의 곱셈 파트로 다항식을 나타낸다. 재료만으로 다항식이 만들어지는 경우 혹은 재료에 상수항 밖에 없는 경우에는 바로 리턴해준다. 재료 중 가장 큰 차수의 재료부터 시작하여, 그 재료로 묶은 다항식과 묶어지지 않은 남은 다항식 두 개를 재료 집합에 있는 재료만으로 알고리즘을 돌렸을 때 곱셈횟수와 다항식형태가 어떤지 리턴하는 방식으로 재귀적으로 채워넣는다. 묶은 다항식과 묶어지지 않은 다항식의 집합은 모든 항의 수가 i개일 때 가장 큰 차수는 포함하기 때문에 2^{i-1} 가지 경우로 나눌 수 있다. 그것을 다 계산하여 곱셈 횟수가 가장 적을 때의 횟수를 저장한다. 각각의 차수의 재료별로 이렇게 하여 가장 작은 차수의 재료까지 모두 연산하여 그 중에서 곱셈 횟수가 가장 작을 때의 곱셈 횟수와 다항식 형태를 데이터베이스에 저장한다. 저장된 데이터는 향후 이보다 큰 차수의 다항식에 대해 계산할 때 다시 사용된다.

알고리즘의 전체 과정에 대하여 자세히 설명하면 다음과 같다. 먼저 찾고자하는 다항식의 차수 N을 넣어 2^N 가지의 다항식을 생성해놓는다. 그 다음 생성된 목표 다항식을 하나씩 받아 가능한 모든 재료의 집합과 모든 묶음의 경우의 수를 탐색한다. 앞서 말한 첫 번째 단계에서 유효한 재료 집합을 계산한다. 여기 재료 집합에는 재료의 조합이 여러 개 있다. 두 번째 단계에서 생성된 목표 다항식과 재료 집합의 재료의 조합을 하나씩 reduce_order함수에 넣어서 그 재료의 조합으로 목표 다항식을 만들 수 있는 최소의 곱셈 횟수와 그 때의 다항식 형태를 리턴받는다. 리턴 받은 곱셈 횟수를 데이터베이스에 저장하고 그 곱셈 횟수에 재료를 만들 때 사용한 곱셈 횟수를 더한 것도 저장한다. 이렇게 하면 목표 다항식 하나에 가능한 모든 재료 조합일 때 각각의 최소 곱셈 횟수와 다항식 형태가 저장된다. 그 다음, 그 중에서도 가장 최소 곱셈 횟수일 때의 재료 조합 하나와 다항식 형태와 그 곱셈 횟수를 따로 출력하여 답을 얻는다.

이것을 앞서 말했던 n차의 모든 다항식에 대해 반복하는 이유는 reduce_order 함수에서 동적 프로그래밍 형태로 더 작은 차수의 다항식에 대한 결과가 큰 차수의 다항식에 대한 결과를 계산할 때 쓰이기 때문에 하나하나 계산하여 저장해두어야

하는 것이다. 각각의 최소 곱셈횟수와 그 때의 재료 조합과 다항식 형태가 저장되어 있기 때문에 N차 미만의 다항식에 대해 계산이 되어있다면 그 결과를 이용하여 N차 다항식의 결과를 계산할 수 있다.

3.2 알고리즘 증명

3.2.1 알고리즘 귀납 증명

우선 해당 알고리즘은 x의 n제곱 형태를 저장하여 다항식을 묶는 방식을 전부 커버함을 보장한다. 1차 다항식은 0차 다항식의 데이터로, N+1차 다항식은 N차 다항식의 데이터로 계산하는 것이 알고리즘의 기본 작동 원리이다. 따라서 귀납법을 사용하여 알고리즘을 증명할 수 있다. 우선 N차 다항식의 종류는 N이 1 이상인 경우에 N차항은 무조건 존재하고 0차항부터 N-1차항이 있거나 없거나에 따라 결정되기 때문에 2^N가지 이다. 0차 다항식과 1차 다항식의 경우, 모든 묶음 방식을 커버하는 것이 자명하다. 모든 N차 다항식의 x의 n제곱 형태를 저장하는 모든 묶음 방식을 알고리즘이 커버한다고 가정한다. 이렇다면 모든 경우에서 N차 다항식은 N차 이하의 다항식으로 쪼개지므로 N차 이하의 모든 묶음 방식에 대해 알고리즘이 커버한다고 볼 수 있다. 모든 N+1차 다항식은 알고리즘에 의해 x의 n제곱 형태인 재료와 재료로 묶어지는 다항식의 곱, 그리고 남은 더 작은 다항식의 덧셈으로 나타내어진다. 재료로 묶어지는 다항식은 N차 이하의 다항식이므로 묶음 방식이 가정에 의해 저장되어있고, 남은 더 작은 다항식 또한 N차 이하의 다항식이므로 묶음 방식이 가정에 의해 저장되어있다. 따라서 N차 다항식의 묶음 형태를 알고리즘이 커버할 때, N+1차 다항식의 묶음 형태도 알고리즘이 커버한다. 귀납법에 의해 x의 n제곱 형태를 저장하여 다항식을 묶어가는 형태는 이 알고리즘에 의해 전부 계산됨을 알 수 있다.

3.2.2 알고리즘 커버 영역

위 증명에 의해 해당 알고리즘은 x의 n제곱 형태를 저장하여 다항식을 묶는 방식을 전부 커버한다. 그러나 그 외의 형태를 저장하는 방식은 커버하지 못하는데,

이를테면 $(x+1)^2$ 등의 다항식을 저장할 수 없다. 현재 프로젝트의 목표는 실수 도메인의 동형암호 컴파일러의 수학함수의 근사다항식을 최적화 하는것이다. 실제 사용되는 근사다항식들은 계수가 모두 정수가 아닌 실수이므로 인수분해가 되기 위해서는 계수 사이에 특정 관계가 성립하여야 하는데 이런 경우가 거의 없다. 따라서 x의 n제곱 형태 외의 다항식 덩어리를 저장하여 곱셈횟수를 감소시키는 것은 거의 일어나지 않는 일이다. 이것이 가능하려면 계수의 특정 성질이 성립해야 한다. 이 부분은 우리 알고리즘이 커버하지 못하는데, 8차 이상의 다항식은 이런 경우가 거의 발생하지 않는다.

Algorithm 1 CORRECT_BASE 의사 코드

```
goal = input()                                  ▷ 목표 다항식을 받는다
bases = select_base(n)                          ▷ 2^{n+1}가지 재료 집합 리턴
while i < n do                                  ▷ 유효한 재료 집합만을 남긴다
    if not CORRECT_BASE(bases[i], n) then
        bases.pop(i)
        i = i − 1
    end if
    i = i + 1
end while

function CORRECT_BASE(base, n)                  ▷ 유효한 재료인지 판별하는 함수
    correct = 0
    base_cnt = 0
    for i in range(n, 3, -1) do
        if base[i] then
            base_cnt = base_cnt + 1
            for j in range(1, i//2+1) do
                if base[j] and base[i-j] then
                    correct = correct + 1
                    break
                end if
            end for
        end if
    end for
    if correct = base_cnt then
        return True
    else
        return False
    end if
end function
```

Algorithm 2 REDUCE_ORDER 함수 코드

```
function REDUCE_ORDER(goal, base)                ▷ 최소 곱셈 횟수 리턴하는 함수
    for i in range(n - 2, 0, -1) do
        if base[i] then
            part1, part2 = calcul(goal, i)  ▷ 묶음의 모든 경우의 수를 계산한다
            for j in range(n) do
                x = (database[GOAL] == part1[j]
                and database[BASE] == base[i])
                y = (database[GOAL] == part2[j]
                and database[BASE] == base[i])
                mul_cnt = x[MUL_CNT]+y[MUL_CNT]
            end for
            break
        end if
    end for
    new_cnt = REDUCE_ORDER(goal, new_base)                          ▷ 재귀
    return min(mul_cnt_list)
end function
```

제 4 장 실험 결과

동형암호 코드에 쓰이는 math operation의 근사다항식 4개와 11차 이하의 다항식 코드를 대상으로 실험을 진행하였다. 다항식의 재료를 0과 1의 배열로 받아 let을 이용하여 변수 선언하도록 바꾸고, 다항식의 형태를 받아서 H 코드를 생성하였다. 최적화 기준인 H 코드를 번역하여 최종 HOS 코드일 때를 기준으로 비용을 계산하였다. 비용은 pi-HEaaN 실행기를 이용하여 계산하였다. 전체 비용은 HEaaN 틀을 기준으로 한 상대 비용을 기준으로 하였다. 여기에서 곱셈 횟수와 정확히 비례하는 횟수를 가지는 linearize 연산은 상대 비용이 430이다. linearize 횟수가 얼마나 줄어드는지의 비율을 성능 개선의 지표로 측정하였다.

4.1 벤치마크 소개

딥뉴럴 네트워크 모델에서 실제로 자주 쓰이는 수학함수인 ReLU, Sigmoid, SigmoidWide, SquareRoot(mini)함수를 근사다항식의 형태일 때를 기준으로 벤치마크를 구성하고 이보다 조금 더 높은 차수의 다항식으로도 실험하였다.

ReLU는 그림 4.1과 같이 8차 다항식으로 표현되고, Sigmoid는 그림 4.2와 같이 7차 다항식으로 표현되고, SigmoidWide는 그림 4.3과 같이 9차 다항식으로 표현된다. SquareRoot는 실제로는 차수가 아주 크지만 loop를 적게 돌린 mini 버젼으로 실험하였다.

4.2 실험 결과

표 4.1에 기존 방식대로 재료를 그냥 2의 제곱 차수 먼저 계산하는 baseline의 결과와 알고리즘을 돌렸을 때의 결과를 비교하였다. 전체 8개의 프로그램 중 7개의 프로그램에서 곱셈 횟수가 감소하였다. 특히 9차 다항식인 SigmoidWide와 11차 다항식 코드에서 직관적인 기존의 방법인 baseline과 비교했을 때 곱셈

표 4.1 실험 결과

	baseline		with algorithm		비용 감소율
	linearize횟수	전체 비용	linearize횟수	전체 비용	(%)
ReLU	5	3011	4	2526	16.1%
Sigmoid	4	2421	4	2421	0%
SigmoidWide	6	3403	4	2437	28.4%
Sqrt(mini)	4	2410	3	2069	14.1%
poly10.1	6	3391	5	2938	13.3%
poly10.2	6	3405	5	2983	12.4%
poly11.1	7	3754	5	3022	19.5%
poly11.2	7	3816	5	3041	20.3%

횟수가 두 번씩 감소하여 30% 이상의 성능 개선을 보여주었다. 추후 더 큰 차수의 다항식에서도 이 결과를 활용할 수 있을 것으로 보인다.

ReLU

- Input
  - x: ciphertext
- Constraint
  - 남은 곱셈 횟수: 4
  - $-1 \le x \le 1$
- Pseudo code

$$
\begin{aligned}
res \leftarrow\ & 1.7345 * 10^{-2} + 5.0000 * 10^{-1} * x + 1.9049 * x^2 \\
& + 1.0784 * 10^{-14} * x^3 - 5.1817 * x^4 \\
& - 2.6412 * 10^{-14} * x^5 + 6.8598 * x^6 \\
& + 1.6165 * 10^{-14} * x^7 - 3.1178 * x^8 \\
\text{return } res
\end{aligned}
$$

그림 4.1 ReLU 연산의 근사다항식

Sigmoid

- Input
  - x: ciphertext
- Constraint
  - 남은 곱셈 횟수: 3
  - $-8 \le x \le 8$
- Pseudo code

$$
\begin{aligned}
res \leftarrow\ & 5.0000 * 10^{-1} + 2.2055 * 10^{-1} * x \\
& + 7.5968 * 10^{-17} * x^2 - 8.5555 * 10^{-3} * x^3 \\
& - 4.0454 * 10^{-18} * x^4 + 1.7437 * 10^{-4} * x^5 \\
& + 4.7743 * 10^{-20} * x^6 - 1.2479 * 10^{-6} * x^7 \\
\text{return } res
\end{aligned}
$$

그림 4.2 Sigmoid 연산의 근사다항식

SigmoidWide

- Input
  - x: ciphertext
  - d: domain extension 계수
- Constraint
  - Bootstrappable
  - Bootstrap 하기전 남은 곱셈 횟수: 2
  - $-14.5 * 2.45^{d} \leq x \leq 14.5 * 2.45^{d}$
- Pseudo code

$y \leftarrow \frac{x}{14.5}$
$do\ i = 1, d$
$\quad y \leftarrow y - \frac{4}{27}\frac{1}{2.25^{(d-i)}}$
$end\ do$
$y \leftarrow 14.5 * y$
$res \leftarrow 5.0000 * 10^{-1} + 1.9390 * 10^{-1} * y$
$\quad - 4.8130 * 10^{-3} * y^{3} + 5.9920 * 10^{-5} * y^{5}$
$\quad - 3.2320 * 10^{-7} * y^{7} + 6.1950 * 10^{-10} * y^{9}$
return res

그림 4.3 SigmoidWide 연산의 근사다항식

제 5 장 논의 및 결론

5.1 결론

다항식 최적화 알고리즘은 다항식의 곱셈 횟수와 연산 순서를 어떤 식으로 저장하느냐에 따라 많이 달라진다. 기존에 논의되었던 동일식 다모으기 방식에 비해 한 번 계산한 재료 즉 이미 선언된 곱셈의 횟수를 추가로 계산하지 않으며 모든 경우를 다 탐색하여 저장하여 활용도가 높아진다는 장점이 있었다. 또한 ReLU, Sigmoid, SigmoidWide 등 실제 딥뉴럴 네트워크 모델에서 자주 사용되는 수학 함수들에서 기존 곱셈 방법에 비해 의미있는 성능 개선을 보여주었다. 동형암호 컴파일러 시스템에 장착되어 코드 성능 최적화에 도움을 줄 수 있을 것으로 보인다.

5.2 알고리즘의 한계점과 해결방안

해당 방식이 적당한 차수에서는 의미있는 결과를 보여주지만, 차수가 올라감에 따라서 연산횟수가 지수적으로 늘어난다.

5.2.1 알고리즘 부분 적용

차수가 클 경우에는 우선 반으로 나누어 각각의 다항식에 알고리즘을 적용하는 식으로 변형하여 활용하면 현실적인 비용으로 최적화를 할 수 있을 것으로 기대된다.

5.2.2 곱셈 깊이 감소

그림 5.1과 같이 Sign 함수의 근사다항식 등 매우 큰 차수의 다항식의 경우, 곱셈 횟수를 감소시켜 linearize 횟수를 줄이는 것보다 bootstrap을 줄이기 위해 곱셈 깊이를 감소시키는 것이 전체 비용이 감소되는데에 더 크게 기여할 수 있을 것으로 보인다. 따라서 고차다항식의 경우에는 프로그램 합성을 이용하여 합성기

Sign

- Input
  - x: ciphertext
  - n_f : f 함수 반복 횟수 (Default 3)
  - n_g : g 함수 반복 횟수 (Default 8)
- Return
  - 1 if $x > 0$, -1 otherwise
- Constraint
  - Bootstrappable
  - $2^{-18} \leq |x| \leq 1$
- Pseudo code

```
res ← x
do i = 1, n_g
   res ← g(res)
end do
do i = 1, n_f
   res ← f(res)
end do
return res
```

where

$$f(x) = \frac{35}{16}x - \frac{35}{16}x^3 + \frac{21}{16}x^5 - \frac{5}{16}x^7,$$
$$g(x) = \frac{4589}{1024}x - \frac{16577}{1024}x^3 + \frac{25614}{1024}x^5 - \frac{12860}{1024}x^7$$

- Error

반복 횟수 (n_f, n_g)에 따른 Sign의 error

n_f / n_g	6	7	8
1	9.3247e-01	7.0385e-01	8.0706e-02
2	8.5297e-01	4.0607e-01	1.6823e-04
3	6.8524e-01	7.0226e-02	3.5527e-15

그림 5.1 Sign 연산의 근사다항식

에 곱셈 깊이 제한을 주어 곱셈 깊이가 적은 다항식을 찾아내는 최적화 방식을 적용할 계획이다. [5]

참고문헌

[1] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. Cryptology ePrint Archive, Paper 2015/1192, 2015. https://eprint.iacr.org/2015/1192.

[2] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.

[3] David W Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–68, 2019.

[4] Sean R Eddy. What is dynamic programming? *Nature biotechnology*, 22(7):909–910, 2004.

[5] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. Optimizing homomorphic evaluation circuits by program synthesis and term rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 503–518, 2020.

Abstract

Optimizing Algorithm of Polynomial Evaluation for CKKS Homomorphic Compiler

Kyuyeon Park
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

This paper introduces a polynomial optimization algorithm which would be on a homomorphic compiler and checks the optimization performance through experiments. We devise an optimization algorithm that minimizes the number of multiplications of polynomials, focusing on the number of multiplication operations which is very expensive in the approximate polynomial of mathematical functions frequently used in the creation of deep neural network models. This algorithm is a dynamic programming-based method that calculates the result of a large polynomial through the calculation result of a polynomial with smaller order, and has the advantage of increasing utilization by searching and storing all cases without duplicating the number of multiplications already calculated. This algorithm improves performance by reducing the number of multiplications up to 33% in a suitable order polynomial compared to the existing method. However, the number of operations increases exponentially as

the order increases.

Keywords: Homomorphic Compiler, Polynomial Optimization, Algorithm

Student Number: 2021-23580