Доказательства с нулевым разглашением и пост-квантовая криптография

Анатолий М. Георгиевский

Дается обзор методов шифрования с использованием структуры нейросети для безопасного запуска моделей нейросетей. Рассматривается три направления развития. Ряд методов доказательства с нулевым разглашением (ZKP); постановка задачи шифрования данных полностью гомоморфного шифрования (FHE) с использованием принципов пост-квантовой криптографии (PQC) и методов построения доказательства с использованием цепной записи данных (blockchain). Цель - сформировать необходимый уровень знаний для разработки собственной концепции запуска нейросетей с использованием RPC протокола и доказательства использования тензоров и модели нейросети с нулевым разглашением. Рассматриваются варианты канонизации описания графов и тензоров для передачи заданий по сети в рамках RPC-протокола.

ключевые технологии: ZKP, zk-SNARK, FHE, LWE, PQC, XOF, blockchain, RPC-протоколы, protobuf, CBOR.

Введение

Современные нейросетевые модели требуют защиты данных и вычислений, особенно в децентрализованных системах. Документ рассматривает интеграцию ZKP, FHE и PQC для обеспечения конфиденциальности и целостности при запуске моделей, а также стандартизацию форматов данных и протоколов для распределенных вычислений и доказательства с использованием цепной записи данных.

Проблематика: необходимость стандартизации и канонизации для безопасного запуска нейросетей. Доказательство использования тензоров и моделей нейросетей с нулевым разглашением (ZKP), использование методов гомоморфного шифрования (FHE) для защиты данных и вычислений. Безопасный поиск.

> Fully Homomorphic Encryption (FHE) is the Holy Grail of Cryptography

Криптографические примитивы

- ZKP (zk-SNARK, zk-STARK): математические основы, свойства (полнота, корректность, нулевое разглашение).
- FHE (Ring-LWE, CKKS, TFHE): принципы работы, алгоритмы шифрования и дешифрования, пост-квантовая устойчивость.
- PQC: роль LWE/Ring-LWE в защите от квантовых угроз.
- XMSS: eXtended Merkle Signature Scheme.
- Blockchain: цепная запись данных,
- Zero-Knowledge Machine Learning (ZKML)
- Inference Privacy: атаки на inference (model inversion, membership inference), защита с использованием ZKP/FHE.

Примеры фреймворков: ZKTorch, Artemis, opp/ai, opML.
Применение в блокчейне: DeFi, идентификация, генеративный ИИ.

- Интеграция ZKP с машинным обучением и генерацией.

Стандартизация и сериализация

- Форматы данных: ONNX, GGUF, Safetensors; бинарное кодирование Protobuf, CBOR.
- Канонизация графов вычислений и тензоров: Protobuf, CBOR.
- Протоколы RPC: gRPC, CoAP, JSON-RPC.

Квантизация и оптимизация

- Форматы квантизации: MXFP8, MXINT8, MXFP4, тернарная логика.
- Влияние и согласование квантового шума (QSNR) в схемах ZKP/FHE.
- Методы компенсации ошибок: пост-квантизация (PTQ, post-training quantization), диффузия ошибки.
- Методы стохастического округления в векторных операциях с добавление квантового шума

Среди обозначенных направлений, нас интересует возможность интеграции пост-квантовой криптографии для безопасного запуска нейросетей и дообучения на приватных данных. Выделяются методы шифрования такие как (LWE, learning with errors), основанные на добавлении ошибки квантования с нормальным или гауссовым распределением. Данные в нейросетях должны сопровождаться доказательством того, что генерация получена с использованием авторизованных данных пользователя и данной нейросети. Для доказательства предложено использовать технологию цепной записи данных поверх вычислительного графа тензорных операций нейросети и протоколы ZKP, доказывающие без раскрытия весов и архитектуры сети, что для получения результата использована определенная нейросеть с входными данными клиента.

Использование методов гомоморфного шифрования требует стандартизации методов сериального представления тензоров, методов канонизации бинарного описания тензоров и графа тензорных операций нейросети. Стандартизация и канонизация рассматривается в контексте RPC протокола. Для реализации бинарного RPC протокола подходят методы кодирования Google Protobuf и CBOR. Протоколы RPC могут быть использованы не только в задачах распределения вычислений в кластере, но и Edge-computing, таких как распределенная обработка визуальных данных.

Мы делаем акцент на использовании квантизованных данных при обучении и запуске нейросетей. Все расчеты выполняются в условиях пониженной разрядности, когда квантовая ошибка с предыдущего цикла накапливается и компенсирует ошибку квантизации. Это важный

принцип работы с физическими моделями, который обеспечивает законы сохранения. Для цифровых фильтров изображений ошибка может компенсироваться за счет перераспределения остатка между соседними пикселями изображения - это принцип диффузии ошибки. Для векторов, так можно минимизировать ошибку длины вектора за счет диффузии ошибки между компонентами вектора.

Zero-Knowledge Proof

Доказательства с нулевым разглашением (Zero-Knowledge Proofs, ZKP) представляют собой криптографический метод, позволяющий одной стороне (проверяющему) убедиться в истинности утверждения, не раскрывая при этом дополнительной информации. Рассмотрим математические принципы, лежащие в основе ZKP, ключевые схемы и их применение в современных системах, таких как блокчейны и протоколы аутентификации.

ZKP позволяют доказать, что некоторое утверждение истинно, без передачи какой-либо дополнительной информации. Например, можно доказать, что вы знаете пароль, не раскрывая сам пароль. В нашем контексте необходимо доказать что к данным пользователя применена функция, без раскрытия информации о самой функции. ZKP обладают тремя основными свойствами:

- *Полнота (Perfect Completeness)*: Если утверждение истинно, честный проверяющий убедится в этом с высокой вероятностью.
- *Корректность (Computational Soundness)*: Если утверждение ложно, ни один злонамеренный доказывающий не сможет убедить проверяющего в обратном, кроме как с ничтожно малой вероятностью.
- *Нулевое разглашение (Perfect Zero-Knowledge)*: Проверяющий не узнает ничего, кроме факта истинности утверждения.

ZKP основаны на трудных вычислительных задачах, таких как дискретное логарифмирование, факторизация больших чисел или эллиптические кривые.

Существует несколько типов ZKP, включая интерактивные и неинтерактивные схемы.

Интерактивные ZKP
Интерактивные ZKP, такие как протокол Шнорра [], требуют многократного обмена сообщениями между доказывающим и проверяющим. Они эффективны, но требуют активного взаимодействия между доказывающей и проверяющей стороной.

Неинтерактивные ZKP (NIZK)
Неинтерактивные ZKP, такие как zk-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of

Knowledge), позволяют доказывающему создать доказательство, которое можно проверить без дальнейшего взаимодействия. В основе лежит принцип публикации (Commitment) некоторых данных, необходимых для выполнения проверки.

zk-SNARK используют эллиптические кривые и полиномиальные обязательства.

- [2024/2025] Mira: Efficient Folding for Pairing-based Arguments
- [2016/260] On the Size of Pairing-based Non-interactive Arguments

{Привести из статьи определения криптографических примитивов}

zk-SNARK. Пусть дана функция $f(x)$, которую нужно проверить. Процесс включает:

- Преобразование $f(x)$ в полиномиальную форму.
- Создание доказательства с использованием парного соответствия (pairing) на эллиптических кривых.
- Проверка доказательства с использованием публичного ключа.

Пример применения: блокчейн Zcash, где zk-SNARK обеспечивают анонимность транзакций.

zk-STARKs. Zero-Knowledge Scalable Transparent Arguments of Knowledge использует пост-квантовую криптографию, криптографический хеш и подпись на основе merkle-деревьев. {подробно}

- [2018/046] Scalable, transparent, and post-quantum secure computational integrity
- (https://aszepieniec.github.io/stark-anatomy)
- (https://github.com/elibensasson/libSTARK)
- [2202.06877] A Review of zk-SNARKs

Zero-Knowledge Machine Learning (ZKML, машинное обучение с нулевым разглашением) is a cryptographic technique that enables the verification of machine learning models on blockchain networks without revealing the underlying data or computations. This technology allows for secure, privacy-preserving, and transparent use of AI models in decentralized applications, ensuring the integrity and trustworthiness of the results. ZKML is particularly useful in DeFi, gaming, and identity verification, where it can enhance user experience, automate decision-making processes, and protect sensitive information.

- [2507.07031] ZKTorch: Compiling ML Inference to Zero-Knowledge Proofs via Parallel Proof Accumulation
  Authors: Bing-Jyue Chen, Lilia Tang, Daniel Kang

> ZKTorch — фреймворк для компиляции вывода (inference) машинного обучения в доказательства с нулевым разглашением (ZKP) с использованием параллельного накопления доказательств. ZKTorch оптимизирует процесс создания zk-SNARK для ML-моделей, снижая вычислительные затраты за счет параллелизации операций над тензорами. Подход протестирован на моделях, таких как ResNet, и демонстрирует улучшение производительности по сравнению с существующими библиотеками, такими как ezkl.

- [2502.18535] A Survey of Zero-Knowledge Proof Based Verifiable Machine Learning
  Authors: Zhizhi Peng, Taotao Wang, Chonghe Zhao, Guofu Liao, Zibin Lin, Yifeng Liu, Bin Cao, Long Shi, Qing Yang, Shengli Zhang
- [2410.13752] Privacy-Preserving Decentralized AI with Confidential Computing
  Authors: Dayeol Lee, Jorge António, Hisham Khan
- [2409.12055] Artemis: Efficient Commit-and-Prove SNARKs for zkML
  Authors: Hidde Lycklama, Alexander Viand, Nikolay Avramov, Nicolas Küchler, Anwar Hithnawi
- [2405.17934] Proof of Quality: A Costless Paradigm for Trustless Generative AI Model Inference on Blockchains
  Authors: Zhenjie Zhang, Yuyang Rao, Hao Xiao, Xiaokui Xiao, Yin Yang
- [2404.16109] zkLLM: Zero Knowledge Proofs for Large Language Models
- [2402.15006] opp/ai: Optimistic Privacy-Preserving AI on Blockchain
  Authors: Cathie So, KD Conway, Xiaohang Yu, Suning Yao, Kartin Wong
- [2402.06414] Trust the Process: Zero-Knowledge Machine Learning to Enhance Trust in Generative AI Interactions
  Authors: Bianca-Mihaela Ganescu, Jonathan Passerat-Palmbach
- [2401.17555] opML: Optimistic Machine Learning on Blockchain
  Authors: KD Conway, Cathie So, Xiaohang Yu, Kartin Wong

> Объединение технологий искусственного интеллекта (ИИ) и технологии блокчейн меняет цифровой мир, предлагая децентрализованные, безопасные и эффективные сервисы ИИ на блокчейн-платформах. Несмотря на обещания, высокие вычислительные требования ИИ на блокчейне вызывают серьёзные проблемы с конфиденциальностью и эффективностью. Фреймворк Optimistic Privacy-Preserving AI (opp/ai) представлен как новаторское решение этих проблем, обеспечивая баланс между защитой конфиденциальности и вычислительной эффективностью. Фреймворк объединяет машинное обучение с нулевым разглашением (zkML) для обеспечения конфиденциальности с оптимистичным машинным обучением (opML) для повышения эффективности, создавая гибридную модель, специально разработанную для сервисов ИИ на блокчейне. В данном исследовании представлен фреймворк opp/ai.

- [arXiv:2402.06414] Trust the Process: Zero-Knowledge Machine Learning to Enhance Trust in Generative AI Interactions
  Authors: Bianca-Mihaela Ganescu, Jonathan Passerat-Palmbach
- [arXiv:2401.17555] opML: Optimistic Machine Learning on Blockchain
  Authors: KD Conway, Cathie So, Xiaohang Yu, Kartin Wong

— использование машинного обучения с нулевым разглашением (zkML). zkML представляет собой новую парадигму интеграции машинного обучения и блокчейна. zkML использует *zk-SNARK* (краткие неинтерактивные аргументы знаний с нулевым разглашением) и играет ключевую роль в защите конфиденциальных параметров модели и пользовательских данных во время процессов обучения и вывода (inference). Это не только снижает проблемы конфиденциальности, но и снижает вычислительную нагрузку на сеть блокчейна, что делает zkML перспективным кандидатом для децентрализованных приложений машинного обучения.

- [2502.02387] SoK: Understanding zk-SNARKs: The Gap Between Research and Practice

> Систематизация знаний о zk-SNARK, включая их теоретические основы, практические реализации и ограничения. Рассматриваются проблемы масштабируемости, настройки (trusted setup) и оптимизации для реальных приложений, таких как блокчейн и ML.

- [2506.20915] ZKPROV: A Zero-Knowledge Approach to Dataset Provenance for Large Language Models

> ZKPROV предлагает метод доказательства происхождения данных для LLM с использованием ZKP. Это позволяет верифицировать, что модель обучена или выполняет inference на авторизованных данных, не раскрывая их содержимое. Применение: защита интеллектуальной собственности и конфиденциальности данных.

Cryptographic Primitives

> Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (*zk-SNARK*). A *zk-SNARK* is a cryptographic proof system that allows a prover to convince a verifier that a statement $x \in L_R$ is valid with respect to a relation R, without revealing any auxiliary information (i.e., the witness ω).

Примитивы, из которых строится схема доказательства применения нейросети:

1. $SETUP(\lambda, W) \rightarrow \{C, \Omega\}$ - компиляция модели со своей архитектурой и весами дает два набора векторов (C)-публичный набор и приватный (Ω).
2. $Prove(C, \Omega, p) \rightarrow \{r, \pi\}$ - запуск модели является доказательством над промптом пользователя (p) генерирует ответ (r) и π - доказательство. Доказательство - это совместно результат вывода модели inference и генерация проверочного вектора (π).

3. $Verify(p, r, \pi, C) \rightarrow Accept, Reject$ -- верификация выполняется с использованием публичных компонент доказательства, над промптом и выводом модели.

Этап компиляции $SETUP()$ можно разложить, на две процедуры: генерацию приватного ключа $pp = KeyGen(\lambda)$ и публичного набора для данной модели $C = Commit(W; \Omega, pp)$

Definition 2.2 (NIZK). A NIZK proof consists of three algorithms (Setup,Prove,Verify) that are defined as follows:

1. $Setup(pp) \rightarrow (pk, vk)$: On input a public parameter pp, it outputs a proving and verification key pk and vk.
2. $Prove(pk, x, w, R) \rightarrow \pi$: On input pk, an instance and witness pair (x,w), and the relation R, it outputs a proof π.
3. $Verify(vk, x, \pi) \rightarrow 0, 1$: On input vk, x, and π, it outputs 1 or 0 to show if π is accepted or not.

{переделать протокол под SNARK}

Важным параметрами построения схемы доказательства является возможность верификации доказательства на стороне клиента и объем данных необходимых для выполнения этой проверки. Важно, чтобы сложность верификации позволяла выполнять проверку налету в процессе загрузки результатов запуска сети. Важно, чтобы сигнатура сети C занимала ограниченный объем данных, а доказательство было сравнимо по длине с вектором семантических признаков `m_embed` .

Операция генерации публичного набора основано на Гомоморфизме приватной модели нейросети. Гомоморфные преобразования сохраняют структуру. Каждой операции сопоставляется другая. При этом есть разногласие в представлении макро-операций таких как MLP, FFN и Attention и нелинейных функций. Тензорные макро-операции можно представлять, как составные или неделимые. Каждой нелинейной операции необходимо сопоставить линейное (полиномиальное) представление.

Наравне с ZKP и SNARK следует рассмотреть принципы построения криптосистемы FHE (Fully Homomorphic Encryption scheme), такой как Ring-LWE, которая обеспечивает шифрование и дешифрацию с использованием принципов удовлетворяющих требованиям PQC (Post-Quantum Cryptography).

Рассмотрение можно начать с принципа поля. Например, RSA так же является HE (гомоморфным преобразованием), строится на операции умножения в конечном поле с использованием модульной арифметики с числами большой разрядности. Мы рассматриваем две операции в модульной арифметике типа умножения и сложения, на которых можно

построить расчеты в полиномиальном приближении. Можно доказать, что в некоторой области пространства непрерывную функцию можно аппроксимировать полиномами с заданной точностью. Таким образом любую функцию мы стремимся преобразовать в полиномиальную функцию для возможности построения схемы доказательства.

Принципы гомоморфного шифрования (HE)

- CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy Homomorphic Encryption, 2016
- [1412.6181] Crypto-Nets: Neural Networks over Encrypted Data

> For our purpose, a (secret key) Homomorphic Encryption scheme consists of four algorithms: encryption (E_k), decryption D_k, addition ($\oplus$) and multiplication ($\otimes$).
> The encryption algorithm takes as input a message and a secret key k. The decryption takes as input an element from the ciphertext space and a key, while the algorithms $\oplus$ and $\otimes$ do not depend on the secret key and only take two ciphertexts as input. Let m_1 and m_2 be integer messages and let k be a secret key.

Представленные алгоритмы удовлетворяют следующим свойствам:

1. Функция шифрования $E_k(m)$, такая что m практически невозможно восстановить обратно без использования приватного ключа k.
2. Существует обратная функция: $m_1 = D_k(E_k(m_1))$.
3. Выполняется свойство линейности: $m_1 + m_2 = D_k(E_k(m_1) \oplus E_k(m_2))$.
4. Выполняется свойство линейности: $m_1 \times m_2 = D_k(E_k(m_1) \otimes E_k(m_2))$.
5. Алгоритмы $\oplus$ и $\otimes$ не используют закрытый ключ шифрования.

Алгоритмы $\oplus$ и $\otimes$ позволяют многократное каскадное применение, так что полиному над m_i соответствует аналогичный полином с операциями $\oplus$ и $\otimes$. Операции обладают коммутативностью.

Let $m_1, ..., m_n$ be messages. Представленные алгоритмы позволяют составить такую полиномиальную функцию, что удовлетворено равенство:

$$P(m_1, ..., m_n) = D_k(\tilde{P}(E_k(m_1), ..., E_k(m_n))) \ .$$

Криптосистема *Homomorphic Encryption* состоит из четырех алгоритмов. Для компиляции схемы из вычислительного графа нейросети используются полиномиальные приближения нелинейных тензорных операций, все коэффициенты строятся из целых чисел по модулю простого числа и операции сдвига с редуцированием (XTIME - вместо масштабирования используется уполовинивание или удвоение позволяет ввести операции умножения и деления полиномов).

Операции умножения полиномов выполняются с использованием нескольких вариантов умножения, которые хорошо разобраны. Один из вариантов умножения находит широкое применение для полиномов с высокой степенью N=10..15 с использованием NTT (аналога быстрого преобразования Фурье).

Схема Ring-LWE использует кольца полиномов. По сути мы говорим про модульную арифметику (алгебру над коммутативным кольцами) или арифметику Галуа в конечном поле поверх тех же полиномов. Однако, могут существовать и более сложные варианты - композитные поля и группы Ли. Во всех случаях требуется представлять нелинейные функции их полиномиальной аппроксимацией, чтобы все операции сводились к умножению и сложению в поле.

В двух словах остановимся на арифметике Галуа. Арифметика галуа GF(2^8) используется при вычислении кодов Рида-Соломона, которые так же построены на полиномах. В системе команд Intel GFNI можно эффективно обрабатывать полиномы. Центральная операция это аппаратная реализация умножения, которая может быть эффективно реализована в FPGA с использованием композитных полей, и операция аффинного преобразования. Но в системе команд GPU аппаратная поддержка отсутствует и акцент делается на параллельные тензорные вычисления в группе с пониженной разрядностью весовых коэффициентов матриц. Таким образом эффективно будут выполняться операции с квантизацией FP8 или INT8, метод эффективного вычисления на GPU должен строится на основе этих форматов. Перспективным можно считать метод, позволяющий уменьшать разрядность операций вплоть до FP4 и тернарной логики. К таким методам можно отнести схему CKKS, работающую с вещественными числами произвольной разрядности.

Заметим, если для операции умножения существует обратная, то существует возможность представления в рациональных функциях (аппроксимация Падé). Использование рациональных аппроксимаций при доказательстве с нулевым разглашением (ZKP) к настоящему времени не применяется, не достаточно изучено. Следует заметить, что запуск моделей нейросетей на различном оборудовании GPU, CPU, NPU не детерминирован иза отсутствия стандартизации методов приближенного вычисления функций активации, таких как: softmax, sigmoid, tanh, swish, SiLU, GeLU. Вычисления сопровождаются квантовым шумом (ошибкой квантизации), который необходимо учитывать при построении схемы доказательства.

Fully Homomorphic Encryption (FHE)

Brakerski/Fan-Vercauteren [Bra12, FV12] scheme, a Ring-Learning With Errors (Ring-LWE)-based crypto-system. Позволяет восстановить данные после шифрования и обладает свойствами пост-квантовой криптографии.

- (https://people.csail.mit.edu/rivest/Rsapaper.pdf) R.L. Rivest, at al., A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, 1978
- (https://crypto.stanford.edu/craig/craig-thesis.pdf) Craig Gentry. A FULLY HOMOMORPHIC ENCRYPTION SCHEME. 2009
- (https://www.cs.cmu.edu/~odonnell/hits09/gentry-homomorphic-encryption.pdf)
- [2009/547] Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers

-- эти три работы с участием Craig Gentry лежат в основе последующих работ по ZKP. Авторы формулируют принципы шифрования для распределенных вычислений с использованием четырех функций

1. $KeyGen(\lambda, F) \to pk, sk$ - генерация двух ключей с использованием значния о функции
2. $\mathrm{ProbGen}_{sk}(x) \to \{\sigma_x, \tau_x\}$ - генерация зашифрованного вектора данных и проверочных данных.
3. $\mathrm{Compute}_{pk}(\sigma_x) \to \sigma_y$ - вычисления выполняются удаленно с использованием зашифрованных входных данных.
4. $\mathrm{Verify}_{sk}(\sigma_y, \tau_x) \to y = F(x)$ в результате проверки восстанавливается результат или устанавливается, что результат не является валидным значением функции.

- (https://cims.nyu.edu/~regev/papers/pqc.pdf) D. Micciancio, O. Regev. Lattice-based Cryptography
- [2011/277] Zvika Brakerski, Craig Gentry, Vinod Vaikuntanathan. Fully Homomorphic Encryption without Bootstrapping, 2011
- [2011/344] Zvika Brakerski, Vinod Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE, 2011
- [2012/078] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP, 2012
- [2012/144] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption, 2012
- [2013/340] Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based, 2013

> GSW (Gentry-Sahai-Waters) — это схема гомоморфного шифрования, основана на проблеме Learning With Errors (LWE). GSW использует матричный подход, где шифротекст представляет собой матрицу, а гомоморфные операции (сложение и умножение) выполняются через матричные вычисления.

- [2014/816] FHEW: Bootstrapping Homomorphic Encryption in less than a second

> В контексте схем гомоморфного шифрования *bootstrapping* (перезапуск или самозапуск) — это процесс, который позволяет увеличить глубину вычислений над зашифрованными данными. Он используется в полностью гомоморфных схемах шифрования (FHE), где после определённого количества операций над шифротекстом уровень шума (noise) может превысить допустимый предел, что делает данные недекодируемыми.

- [2016/421] J.H. Cheon, at al. Homomorphic Encryption for Arithmetic of Approximate Numbers

> CKKS (Cheon-Kim-Kim-Song) — это схема полностью гомоморфного шифрования (FHE), предназначенная для эффективных вычислений с вещественными числами. Она позволяет выполнять операции сложения, умножения и другие над зашифрованными данными, не раскрывая исходную информацию.

- [2016/837] J.H. Cheon and D. Stehle. Fully Homomorphic Encryption over the Integers Revisited, 2016
- [2018/117] An Improved RNS Variant of the BFV Homomorphic Encryption Scheme
- [2018/931] J.H. Cheon, at al. A Full RNS Variant of Approximate Homomorphic Encryption

> В этой работе представлен вариант приближенного гомоморфного шифрования, который оптимален для реализации на стандартных компьютерных системах. Вводится новая структура модуля шифротекста, которая позволяет использовать как разложение циклотомических многочленов в RNS, так и преобразование NTT на каждом из компонентов RNS.

Residue Number System (RNS) — это система представления чисел, в которой целое число выражается набором его остатков от деления на несколько взаимно простых модулей.

- [2018/039] Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography
- [2018/421] TFHE: Fast Fully Homomorphic Encryption over the Torus

> четко даются определения и математические основы полностью гомоморфного преобразования на единичном торе.

Обозначения. In the rest of the paper, we denote the security parameter as λ.
We denote as $\mathbb{B}$ the set $\{0,1\}$ without any structure and by $\mathbb{T}$ the real Torus $\mathbb{R}/\mathbb{Z}$, the set of real numbers modulo 1. We denote by $\mathbb{Z}_N[X]$ the ring of polynomials $\mathbb{Z}[X]/(X^N+1)$. $\mathbb{T}_N[X]$ denotes $\mathbb{R}[X]/(X^N+1) \mod 1$ and $\mathbb{B}_N[X]$ denotes the polynomials in $\mathbb{Z}_N[X]$ with binary coefficients.

Определение ($\mathcal{R}$-module). Let $(\mathcal{R},+,\times)$ be a commutative ring. We say that a set M is a $\mathcal{R}$-module when is an abelian group, and when there exists an

external operation $\cdot$ (product) which is bi-distributive and homogeneous. Namely,
$\forall r, s \in R$ and $x, y \in M$,
$1_{\mathcal{R}} \cdot x = x$, $(r + s) \cdot x = r \cdot x + s \cdot x$, $r \cdot (x + y) = r \cdot x + r \cdot y$,
and $(r \times s) \cdot x = r \cdot (s \cdot x)$.

- [2018/828] Aurora: Transparent Succinct Arguments for R1CS
- [2019/317] Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation
- [2020/086] Bootstrapping in FHEW-like Cryptosystems
- [2021/1337] Large-Precision Homomorphic Sign Evaluation using FHEW/TFHE Bootstrapping
- [2022/198] Efficient FHEW Bootstrapping with Small Evaluation Keys, and Applications to Threshold Homomorphic Encryption
- [2022/915] OpenFHE: Open-Source Fully Homomorphic Encryption Library
- [2024/463] Security Guidelines for Implementing Homomorphic Encryption
- [2024/1881] THOR: Secure Transformer Inference with Homomorphic Encryption
- [2024/2025] Mira: Efficient Folding for Pairing-based Arguments
- [2025/263] Transparent SNARKs over Galois Rings
- [2025/882] Leveled Homomorphic Encryption over Composite Groups
- [2103.16400] Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52
- [2401.03703] On Lattices, Learning with Errors, Random Linear Codes, and Cryptography
- [2503.05136] The Beginner's Textbook for Fully Homomorphic Encryption (https://fhetextbook.github.io/)
- [2507.04501] LINE: Public-key encryption
- (https://faculty.kfupm.edu.sa/coe/mfelemban/SEC595/References/Introduction to the BFV FHE Scheme.pdf)
- [OpenFHE] (https://github.com/openfheorg/openfhe-development)

> Fully Homomorphic Encryption (FHE) is a powerful cryptographic primitive that enables performing computations over encrypted data without having access to the secret key. OpenFHE is an open-source FHE library that includes efficient implementations of all common FHE schemes:

- Brakerski/Fan-Vercauteren (BFV) scheme for integer arithmetic
- Brakerski-Gentry-Vaikuntanathan (BGV) scheme for integer arithmetic
- Cheon-Kim-Kim-Song (CKKS) scheme for real-number arithmetic

Software references for publicly available Homomorphic Encryption libraries:

- [cuFHE] (https://github.com/vernamlab/cuFHE)
- [cuHE] (https://github.com/vernamlab/cuHE)
- [HEAAN] (https://github.com/snucrypto/HEAAN)

- [HElib] (https://github.com/shaih/HElib)
- [NFLlib] (https://github.com/CryptoExperts/FV-NFLlib)
- [PALISADE] (https://git.njit.edu/groups/PALISADE)
- [SEAL] (http://sealcrypto.org)
- [TFHE] (https://tfhe.github.io/tfhe/)

Стандартизация

- Homomorphic Encryption Standardization
- [HESv1.1] Homomorphic Encryption Standard, 2018
- [NIST:fips.203] Module-Lattice-Based Key-Encapsulation Mechanism Standard. Tech. rep. National Institute of Standards and Technologies, 2024. (http://dx.doi.org/10.6028/NIST.FIPS.203)
- [NIST:fips.204] Module-Lattice-Based Digital Signature Standard. Tech. rep. National Institute of Standards and Technologies, 2024. (http://dx.doi.org/10.6028/NIST.FIPS.204)
- [NIST:fips.205] Stateless Hash-Based Digital Signature Standard (http://dx.doi.org/10.6028/NIST.FIPS.205)

Стандарты серии PQC применяют Ring-LWE для построения схемы цифровой подписи и схемы выработки ключей для симметричной криптографии. Используют

- The prime number $q = 2^{23} - 2^{13} + 1 = 8380417$
- кольцо полиномов $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^{256} + 1)$

Для быстрого вычисления произведения полиномов используются алгоритмы прямого и обратного NTT (Number Theoretic Transform).

$$g \circ h = \mathrm{NTT}^{-1}(\mathrm{NTT}(g) \odot \mathrm{NTT}(h))$$

- [34] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” Mathematics of computation, vol. 19, no. 90, pp. 297–301, 1965.
- [35] W. M. Gentleman and G. Sande, “Fast fourier transforms: for fun and profit,” in Proceedings of the November 7-10, 1966, fall joint computer conference, 1966, pp. 563–578. (https://doi.org/10.1145/1464291.1464352)
- [2021/1396] NTT software optimization using an extended Harvey butterfly
- [1205.2926] David Harvey. Faster arithmetic for number-theoretic transforms

Shoup modular multiplication. The most time-consuming primitive in NTT algorithms is modular multiplication between the coefficients of a and the fixed (precomputed) powers of ω.

- [37] Accelerating High-Precision Number Theoretic Transforms using Intel AVX-512 (https://spiral.ece.cmu.edu/pub-spiral/pubfile/PACT_2024_AVX_371.pdf)
- (https://arxiv.org/pdf/2404.13544)
- [2306.01989] Optimized Vectorization Implementation of CRYSTALS-Dilithium

XMSS: eXtended Merkle Signature Scheme

Стандарт FIPS.205 определяет режим eXtended Merkle Signature Scheme (XMSS) и допускают использование функций хеширования: SHA-256, SHA-512, SHAKE-128, SHAKE-256.

- [NIST:FIPS.202] SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (http://doi.org/10.6028/NIST.FIPS.202)

> The hash functions are two eXtendable Output Functions (XOF), namely `SHAKE-256` and `SHAKE-128`.
> XOF maps an arbitrary-length bit string to a string of infinitely many bits. These XOF functions are mainly used for generating random bytes of SHAKE-128 to sample matrix A and for generating random bytes of SHAKE-256 to sample s, e and y.

- [NIST SP 800-185] SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash
- [NIST SP 800-208] Recommendation for Stateful Hash-Based Signature Schemes
- [RFC8391] XMSS: eXtended Merkle Signature Scheme, 2018 (https://doi.org/10.17487/RFC8391)
- [RFC8554] Leighton-Micali Hash-Based Signatures, 2019 (https://doi.org/10.17487/RFC8554)

Функция хеширования на модульной арифметике

- [2019/458] POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems
- [2019/1107] On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy
- [2022/403] Horst Meets Fluid-SPN: Griffin for Zero-Knowledge Applications
- [2023/107] The Tip5 Hash Function for Recursive STARKs

-- еще более скоростная функция, основананя на модульной арифметике. Использует NTT для матриц перестановок.

- [2023/1025] Monolith: Circuit-Friendly Hash Functions with New Nonlinear Layers for Fast and Constant-Time Implementations
- [Skyscraper] Skyscraper: Fast Hashing on Big Primes, 2025

- [2025/058] Skyscraper-v2: Fast Hashing on Big Primes
- [2023/323] Poseidon2: A Faster Version of the Poseidon Hash Function

-- Цель данной работы упростить функцию хеширования и ускорить ее вычисление за счет параллельного вычисления функций. Требование - использование модульной арифметики совместимой с вчислениями на кольце, длина состояния хеша должна настраиваться на используемую систему полиномов и совмещаться с zk-SNARK по используемым модулям, а модули $\mathbb{F}_q$ выбираются совместимыми кольцом полинов $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^N \pm 1\rangle$ из Ring-LWE. Алгоритм должен быть arithmetization-frendly, т.е. основан на полиномиальных операциях, содержащих умноежние и сложение. Вектор состояния разбивается на блоки кратные 4. Линейная стадия использует умножение матрицы 4x4 на вектор, нелинейная стадия использует степенную функцию в качестве S-box (нелинейных подстановок): $x \to x^d$, где d - взаимно простое число $gcd(d, q-1) = 1$, например 3, 5 или 7. Дополнительный параметр выбора $gcd(d, p+1) \neq 1$ обеспечивает некоторую устойчивость к атакам на S-box.

Каждый слой определяется, как произведение матрицы вращения M на результат нелинейнго преобразования вектора x.

$$\mathcal{E}_i(x) = M_{\mathcal{E}} \cdot \left((x_0 + c_0^{(i)})^d, (x_1 + c_1^{(i)})^d, ..., (x_{t-1} + c_{t-1}^{(i)})^d\right)$$

Конструирование нелиной функции может быть выполнено с использованием бинарных операций XOR, NOT и AND на 8-битных словах.

- [2023/1025] Monolith: Circuit-Friendly Hash Functions with New Nonlinear Layers for Fast and Constant-Time Implementations

Тут применяется нелинейная функция S-Box:

$$\mathcal{S}_i(x) = (x \oplus ((\bar{x} \lll 1) \odot (x \lll 2) \odot (x \lll 3))) \lll 1 \text{ , над } \mathbb{F}_2^8$$
$$\mathcal{S}_i(x) = (x \oplus ((\bar{x} \lll 1) \odot (x \lll 2))) \lll 1 \text{ , над } \mathbb{F}_2^7$$

отмечается использование S-Box в SHA3, нелиненая операция $\mathbb{F}_2^7$:

$$\mathcal{S}_i(x) = x \oplus ((\bar{x} \lll 1) \odot (x \lll 2))$$

линейная операция:

$$\mathcal{S}_i(x) = x \oplus (x \lll 1) \oplus (x \lll 2)$$

Тенденция выбирать коэффициенты MDS матриц бездумно сохраняется. Матрицы генериуются как цикркулянты или негациклические матрицы по первому столбцу. Коэффициенты заполняются с использованием SHAKE128 с ограничением разрядности 16 бит.

Эффективные циркулянтные MDS-матрицы

Матрица $N \times N$, заданная циркулянтом $\mathrm{circ}(c_0, c_1, \ldots, c_{t-1})$, определяется следующим образом:

$$M = \begin{pmatrix} c_0 & c_1 & \cdots & c_{t-1} \\ c_{t-1} & c_0 & \cdots & c_{t-2} \\ \vdots & \vdots & \ddots & \vdots \\ c_1 & c_2 & \cdots & c_0 \end{pmatrix}.$$

Умножение плотной $N \times N$ матрицы на вектор размерности N может потребовать числа операций порядка $O(N^2)$. Однако, на кольце полиномов $\mathcal{R}_q = \mathbb{F}_q[x]/(x^N - 1)$ существует изоморфизм между $\mathcal{R}_q$ и циркулянтными матрицами $N \times N$.

Эта связь также описана в [2022/1577, раздел 4], где авторы предлагают использовать циркулянтные матрицы для повышения эффективности хеш-функции *Rescue*. В частности, изоморфизм задается следующим образом:

$$a_{N-1}X^{N-1} + a_{N-2}X^{N-2} + \cdots + a_1 X + a_0 \longleftrightarrow \begin{pmatrix} a_0 & a_{N-1} & \cdots & a_1 \\ a_1 & a_0 & \cdots & a_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_{N-1} & a_{N-2} & \cdots & a_0 \end{pmatrix},$$

где $a_i \in \mathbb{F}_q$. Следовательно, для циркулянтов может использоваться метод NTT полиномиального умножения по модулю $X^N - 1$. Алгоритм умножения полиномов NTT обладает асимптотической сложностью $O(N \log N)$. Возможность применния NTT ограничивается выбором модуля $q = 2^L - \alpha 2^s + 1$ и полинома $x^N - 1$, где $s \geq 2N$. Для оптимизированной функции *Rescue* выбрано простое число $q = 2^{64} - 2^{32} + 1$.

- [2022/1577] Rescue-Prime Optimized
- [2020/1143] Rescue-Prime: a Standard Specification

В отличие от оптимизированной функции *Rescue*, стандартная использует два варианта S-box: x^α и $x^{\bar{\alpha}}$. α and $\bar{\alpha}$ удовлетворяют условию $\left(x^\alpha\right)^{\bar{\alpha}} = x$ для всех $x \in \mathbb{F}_q$. Для нахождения обратной величины рекомендуется воспользоваться расширенным алгоритмом Евклида.

Таким образом функция пермутации для SPONGE (губки) основана на модульной арифметике и двух операциях: умножении и сложении. S-box - нелиненый элмент определяется, как поэлементное возведение в степень. Матричные операции 4x4 (32-бит) хорошо оптимизируются на современных процессорах. Матричные перестановки MDS синтезируются на основе циркулянтов. Операция матричного умножения реализуется с использованием прямого и

обратного NTT (Number Theoretic Transform). Вот основные идеи построения функции хеширования для ZKP.

Сюда следует добавить два варианта использования функции пермутации: в составе губки (SPONGE) для генерации XOF и в составе функции компрессии для уменьшения размера контекста.

- [SPONGE] Cryptographic sponge functions

{Осмысление} Я представляю, что схема нейросети построена по тому же принципу, что и функция SPONGE в стандарте SHA3 или в новых разработках, ориентированных на интеграцию со схемами zk-SNARKs и zk-STARKs. Работа сети состоит из двух этапов: 1- впитывание `absorb` и 2- выдавливание `squeeze` - генерации. При этом на первом этапе происходит накопление данных в контексте, а на втором генерация и заполнение контекста. Так же можно представить нейросеть, как некоторую функцию, подобную функци SPONGE, которая в качестве параметра принимает один или два вида функций.

Среди этих идей я выделяю одну - использование NTT для линенйной стадии смешивания. Это означает необходимость реализации NTT в библиотеке. И необходость реализации операции нелинейной векторной операции "S-box" возведения в степень минимального простого числа - константы, которая определяется выбором простого числа q и является генератором ультипликативной группы.

Например, Для модуля $q = 2^{31} - 1$ (число Mersenne 31) значение показателя степени нелинейной функции $\alpha = 5$.

Значение $\alpha = -1$ принятое в первоначальной версии *Poseidon* подходит для модулей $p = a\beta - 1$, но небезопасное для модулей вида $p = a\beta + 1$.

Поиск простых чисел для NTT
Для простого числа должен существовать корень степени n из единицы, то есть $\omega^n \equiv 1 \mod q$. Для расчетов NTT понадобится корень $\gamma^2 \equiv \omega \mod q$. Условием существования корня степени n из единицы по модулю простого числа q является условие $n|(q-1)$.

Вероятностным критерием пригодности простого числа является возможность извлечения квадратного корня из -1, то есть $x^2 \equiv -1 \mod q$.
Числа вида $q = a\beta + 1$ при $a < \beta = 2^s$ -- простые числа Прота. {См. теорему Прота и тестирование чисел Прота на простоту.}

Теорема Прота
Если p — это число Прота вида $A \cdot 2^n + 1$, где A — нечётно и

$A < 2^n$, то p — простое (называемое простым Прота) тогда и только тогда, когда для некоторого квадратичного невычета g выполняется сравнение:
$g^{(p-1)/2} \equiv -1 \pmod{p}$

Тестирование на простоту выполняется по теореме Прота для случайной или последовательной выборки квадратичных невычетов:

1. Найти квадратичный невычет (генератор мультипликативной группы) $\operatorname{jacobi}(g, p) = -1$.
2. Проверить для квадратичного невычета $g^{(p-1)/2} \equiv -1 \pmod{p}$, тогда p — простое по теореме Прота.

Мне удалось извлечь корень сепени n=16,32, 256,512 из некоторых простых чисел вида $A \cdot 2^{16} + 1$ или $2^{32} - a0 \cdot 2^{16} + 1$. В литературе упоминаются числа вида $p = A \cdot 2^s + 1$, где A - простое нечетное число. Для таких чисел существует корнеь степени 2^s из единицы.

- [2012/470] Some Connections Between Primitive Roots and Quadratic Non-Residues Modulo a Prime

-- дан алгоритм выбора чисел ω и γ для NTT через квадратичные не-вычеты. Для работы алгоритма требуется соблюдение условия $p = q \cdot 2^s + 1$, $2N|(p-1)$, q - простое нечетное число. Если на кольце $\mathcal{R}_p = \mathbb{F}_p[x]/\langle x^N - 1\rangle$ заявлена степень $N = 2^8$, то необходимо чтобы $s \geq 9$.

Проверил ряд простых чисел:

- $2^{23} - 2^{13} + 1$
- $2^{31} - 2^{27} + 1$
- $2^{31} - 2^{25} + 1$
- $2^{31} - 2^{24} + 1$
- $2^{31} - 2^{19} + 1$
- $2^{31} - 2^{17} + 1$
- $2^{31} - 2^{9} + 1$

Theorem (Generalized Pepin Test)
Let $p = a\beta + 1$, $a < \beta = 2^s$, $s \geq 2$ be a Proth number defined in such that $g \nmid a$. Then p is a prime if and only if $g^{(p-1)/2} \equiv -1 \pmod{p}$.

- [0812.2596] Deterministic Primality Proving on Proth Numbers

Теорема (обобщенная теорема Прота)
Пусть $p = a \cdot r^s + 1$ для некоторого простого

r и целых $s, a \geqslant 1$, для $r^e > a$.
Если для g - квадратичного невычета выполняектся $g^{p-1} \equiv 1 \pmod p$ и $g^{(p-1)/r} \not\equiv 1 \pmod p$ для некоторого a, то p — простое.

Тест простоты мы можем сформулировать так, для некоторого g - квадратичного невычета по модулю $p = a\beta + 1$, $a < \beta = 2^s$, $s \geq 2$:

1. $g \leftarrow 3$
2. while $\operatorname{jacobi}(g, p) \neq -1$ do $g \leftarrow g + 2$
3. $g^{p-1} \equiv 1 \pmod p$ и $g^{(p-1)/2} \equiv -1 \pmod p$

Помимо теста простоты должен соблюдаться критерий выбора простого числа p для NTT, который требует $2N|(p-1)$. Среди модулей, которые прошли проверку на простоту и совместимость с NTT следует выбрать те, которые дают максимальный порядок генератора. Это достигается, в частности, если a - простое число.

Тест простоты Миллера-Рабина
Пусть p — простое число и $p = a \cdot 2^s + 1$, где a — нечётно. Тогда для любого $g \in \mathbb{Z}_p$ выполняется хотя бы одно из условий:

1. $g^a \equiv 1 \pmod p$
2. Существует целое число $r < s$ такое что $g^{2^r a} \equiv -1 \pmod p$

На практике нет смысла использовать вероятностный тест простоты Миллера-Рабина для простых чисел, которые проходят тест простоты Прота. Но в тест Прота я внес некоторое изменение, которое согласуется с тестом Миллера-Рабина.

1. $g \leftarrow 3$
2. while $\operatorname{jacobi}(g, p) \neq -1$ do $g \leftarrow g + 2$
3. $g^{p-1} \equiv 1 \pmod p$ и $g^{(p-1)/2} \equiv -1 \pmod p$

В пункте (3) мы видим условия теста Ферма и теста Миллера-Рабина. Тест может завершиться раньше, если при возведении в степень найдено число 1. Вероятностный тест простоты Соловея — Штрассена проверяет тождество:
$g^{(p-1)/2} \equiv \left(\frac{g}{p}\right) \pmod p$. Таким образом, пункт (3) одновременно удовлетворяет условиям теста Ферма, теста Прота, теста Миллера-Рабина и теста Соловея — Штрассена.

Нашел несколько подходящих простых чисел для N=256, s=9:

```
prime=7ffd5601 ord 7ffd55ff
prime=ffffca01 ord ffffc9ff
prime=fffe2601 ord fffe25ff
prime=fff55601 ord fff555ff
prime=fff30a01 ord fff309ff
prime=ffefc201 ord ffefc1ff
prime=ffea2201 ord ffea21ff
prime=ffe6da01 ord ffe6d9ff
```

Основные выводы по разделу. Мы можем предложить функцию возвращающую некриптографический хэш от тензора используя модульную арифметику на кольце с тем же модулем и полиномом. В тоже время есть возможность составить криптографический хэш на модульной арифметике, примером является Poseidon2. При переводе матрицы циклических перестановок MDS в модульную арифметику, мы можем использовать NTT для ускорения операции умножения и выпонении операций перестановок, в том случае, когда матрица MDS задается как циркулянт. NTT возможно использовать, когда модуль имеет корень N-степени. Есть сомнение что выбор модулей вида $p = q \cdot 2^s + 1$ - целесообразен. В отличие от модулей $p = q \cdot 2^s - 1$, где можно получить период повтора генератора $(p/2 - 1)$, период повтора (порядок мультипликативной группы) может быть короткий. Среди модулей, которые прошли проверку на простоту и совместимость с NTT следует выбрать те, которые дают максимальный порядок генератора.

{перенести}

Обозначения

Мы вводим обозначения $\mathbb{Z}_q$ как множества целых чисел $(-q/2, q/2]$ где $q > 1$ - целые числа. Все целочисленные операции выполняются по модулю $(\mod q)$ если не сказано обратное. Для упрощения, you will see me mostly deal with positive integers in $[0, q)$, but keep in mind that it's the same as our $\mathbb{Z}_q$, as $-x \equiv q – x (\mod q)$ where x is a positive integer (e.g. $-1 \equiv 6 (\mod 7)$). Каждый вектора $v \in \mathbb{Z}_q^n$ можно рассматривать как вектор элементов из класса $\mathbb{Z}_q$.

We will use $[\cdot]_m$ to specify that we are applying modulo m, and $\lfloor \cdot \rceil$ for rounding to the nearest integer.

Угловыми скобками $\langle a, b \rangle$ обозначим скалярное произведеление (inner product) двух векторов $a, b \in \mathbb{Z}_q^n$ и определим операцию через умножение и сложение по модулю q

$$\langle a, b \rangle = \sum_{i}^{n} a_i \cdot b_i \ (\mod q)$$

Learning With Error

Learning With Error (LWE) was introduced by Regev in 2009 and can be defined as follows:

Для целых чисел $n \geq 1$ и $q \geq 2$, let's consider the following equations

$$\begin{aligned}
\langle s, a_1 \rangle + e_1 &= b_1 \ (\mod q) \\
\langle s, a_2 \rangle + e_2 &= b_2 \ (\mod q) \\
&\dots \\
\langle s, a_m \rangle + e_m &= b_m \ (\mod q)
\end{aligned}$$

where s and ai are chosen independently and uniformly from $\mathbb{Z}_q^n$, and e_i are chosen independently according to a probability distribution over $\mathbb{Z}_q$, and $b_i \in \mathbb{Z}_q$. The LWE problem state that it's hard to recover s from the pairs (a_i, b_i), and it's on such hardness that cryptography generally lies. On the list of candidate algorithms for the post-quantum cryptography standardization are some that are based on LWE, so you would probably hear more about it when it would be used in key- establishment and public-key encryption.

Ring Learning With Error

Ring-LWE is a variant of LWE, it's still based on the hardness of recovering s from the pairs (a_i, b_i), and the equations are mainly the same, however, we go from the world of integers ($\mathbb{Z}_q^n$) to the world of polynomial quotient rings ($\mathbb{Z}_q[x]/\langle x^n + 1 \rangle$), this means that we will deal with polynomials with coefficients in $\mathbb{Z}_q$, and the polynomial operations are done (mod) some polynomial that we call the polynomial modulus (in our case: $\langle x^n + 1 \rangle$), so all polynomials should be of degree $d < n$, and $x^n \equiv -1 (\mod \langle x^n + 1 \rangle)$.

Let's now use a more formal definition of Ring-LWE by Regev:
Let n be a power of two, and let q be a prime modulus satisfying $q = 1 (\mod 2n)$. Define R_q as the ring $\mathbb{Z}_q[x]/\langle x^n + 1 \rangle$ containing all polynomials over the field $\mathbb{Z}_q$ in which x_n is identified with -1. In Ring-LWE we are given samples of the form $(a, b = a \cdot s + e) \in R_q \times R_q$ where $s \in Rq$ is a fixed secret, a∈Rq is chosen uniformly, and e is an error term chosen independently from some error distribution over Rq.

So if we want to build an HE scheme using Ring-LWE, then our basic elements won't be integers, but polynomials, and you should be familiar with basic polynomial operations (addition, multiplication and modulo). I cooked up a quick refresher of polynomial operations to avoid getting off on the wrong foot, but you can just skip it if it's a trivial thing for you.

Polynomial Arithmetic и Аппроксимация Падé

(https://mathworld.wolfram.com/PadeApproximant.html)
Прежде всего нас может интересовать метод вычисления аппроксимации Падé для экспоненциальных функций, таких как `SiLU` , `GELU` , `tanh` и `exp` , в составе функции `softmax` .

Однако конкретное представление (аппроксимация) нелинейной функции будет определяться настройкой криптосистемы и степенью полиномов в числителе и знаменателе рациональной функции.

Аппроксимация Падé для экспоненциальной функции, пример:

$$\exp_{3/3}(x) = \frac{\exp(+x/2)}{\exp(-x/2)} \approx \frac{120 + 60x + 12x^2 + x^3}{120 - 60x + 12x^2 - x^3} \ .$$

Определив экспоненту, как ряд или как отношение рядов, можно применить такое определение и к вектору и к матрице (Матричная экспонента). Стоит отметить, что в системе команд x86 отсутствует векторная инструкция расчета экспоненты. Все такие функции эмулируются, за исключением логарифма. Таким образом, просто вводя правило вычисления экспоненты можно устранить неопределенность.

Для простоты понимания материала, я предпочитаю представлять функцию шифрования $E_k(.)$, как матрицу ортогонального (Аффинного) преобразования размером `n_embed`, для которой существует обратное преобразование $D_k(.)$ - обратная матрица. Такое представление интуитивно понятно для CNN сетей, но для каждой функции следует определить аппроксимацию и степень полинома. С другой стороны аппроксимация может быть основана на ортогональных и базисных полиномах, таких как полиномы Чебышева и Якоби, Базисные полиномы С.Н.Бернштейна. При использовании систем базисных и ортогональных полиномов для аппроксимации MLP (Feed-forward Network) возникает представление в виде сетей KAN.

{Тут следует определить правила преобразования для рациональных функций и для полиномов данной степени и перенести их в раздел Арифметизация нелинейных функций}. Выбрать правила арфиметизации для функций активации. Нелинейные функции порождают ошибку, которая сильно зависит от правил округления. В этой связи мы вводим рассмотрение правил округления при переводе вещественных чисел в модульную арифметику. Правила включают эмуляцию методов RNE, RTN, RTZ, режимов stochastic rounding и компенсацию квантовой ошибки на потоке данных, обратное распространение ошибки, диффузия ошибки. Сравнить правила округления принятые в с схеме CKKS. Определить переход между системами базисных полиномов данной степени через матричные преобразования.

Функции активации

- [1606.08415] Gaussian Error Linear Units (GELUs)
- [2002.05202] GLU Variants Improve Transformer

Базовые аппроксимации вводятся для функций активации таких SwiGLU и GELU. Эти функции могут быть выражены через функцию сигмоид.

$$\mathrm{SwiGLU}_\beta(z, z\prime) := \mathrm{Swish}_\beta(z) \cdot z\prime$$
$$\mathrm{Swish}_\beta(x) = x\mathrm{Sigmoid}(\beta x) = \frac{x}{1 + e^{-\beta x}}$$
$$\mathrm{GELU}(z) := z\Phi(z) \approx z \cdot \mathrm{Sigmoid}(1.702z)$$

Все подобные функции активации можно представить в виде полиномиальной аппроксимации (exp, sigmoid, tanh).

Lemma 1. Let N be a neural network in which all non-linear transformations are continuous. Let $X \subset \mathbb{R}^n$ be the domain on which N acts and assume that X is compact, then for every $\epsilon > 0$ there exists a polynomial P such that

$$\sup_{x \in X} \|N(x) - P(x)\| < \epsilon$$

Основное в этом утверждении непрерывность и компактность, которые позволяют обосновать применение полиномов. Я бы отослал к аппроксимационной теореме Вейерштрасса и доказательству Бернштейна с выводом системы базисных полиномов.

Аппроксимационная теорема Вейерштрасса утверждает, что любую непрерывную функцию на отрезке [0,1] (на компактном множестве) можно сколь угодно точно аппроксимировать многочленами (то есть подобрать рекурсивную последовательность многочленов, равномерно сходящихся по норме к данной функции).

- (https://github.com/microsoft/CryptoNets)

> Encrypting data is a prominent method for securing and preserving privacy of data. Homomorphic encryption (HE)
> (Rivest et al., 1978) adds to that the ability to act on the data while it is still encrypted. In mathematics, a homomorphism is a structure-preserving transformation.

Существует ряд библиотек предназначенных для выполнения операций типа ZKP (zero-knowledge proof),

- (https://docs.ezkl.xyz/)

Доказательство целостности модели выполняется с использованием описания архитектуры (графа тензорных операций) и весов модели в формате ONNX (Open Neural Network eXchange). На первом этапе строится схема доказательства с использованием графа тензорных операций нейросети, нелинейные операции в графе вычисления подменяются на полиномиальную аппроксимацию. Все вычисления выполняются в числах с фиксированной точкой.

```
import ezkl
# Настройка параметров
settings = ezkl.gen_settings()
ezkl.compile_model("llama.onnx", "compiled_llama.ezkl", settings)
# Генерация доказательства
proof = ezkl.prove("input.json", "compiled_llama.ezkl", "pk.key")
# Верификация
ezkl.verify(proof, "compiled_llama.ezkl", "vk.key")
```

Тезис. Мы хотим сформулировать современный метод компиляции графа вычислительной сети с использованием принципов zero-knowledge, который бы являлся ключом для построения сложных схем доказательства использования сложной функции и тензоров весов с использованием цифровой подписи на основе merkle-tree и применялся практически к любым вычислительным графам.

Применительно к нейросетям LLM Компиляция графа доказательства должны выполняться с использованием коэффициентов в открытых форматах `GGUF` или `Safetensors` и канонического представления графа в формате протокола RPC. Тензоры изначально представленные в формате BF16 должны быть квантизованны в числах с фиксированной точностью пригодные для модульной арифметики. Метод квантизации в доказательствах следует выбирать исходя из методов квантизации оптимизированных для обучения на GPU, таких как MXFP8, MXINT8.

Необходимо разработать метод канонизации описания графа тензорных операций и представления в бинарном формате основанном на `CBOR` кодировании, работающий на множестве операций поддерживаемых в `llama.cpp` и `ONNX`. При вычислении графа над результатом каждой тензорной операции выполняется функция квантования построенная по принципу модульной арифметики и сдвига в конечном поле (умножение и редуцирование полиномов). Должны быть разработаны методы для квантизации 2-бит, 4-бит, 8-бит, а также квантизация в тернарную логику. Методы квантизации должны быть оптимизированы под современную архитектуру тензорных ядер GPU.

В схему компиляции предлагается добавить подбор *nonce* (квантовой ошибки используемой при округлении результата операции) для операции хеширования тензора на каждой тензорной операции или на слое сети. Для достижения критерия вычислительной сложности составления схемы помимо валидности хеш, подбор выполняется при вычислении каждого хеша после каждой операции в дереве. Набор *nonce* полученных таким образом включают в схему проверки. Что это дает? Проверка поверх известной сети будет выполняться на любом оборудовании в достаточно короткое время, в то время, как расчет подобной схемы требует большого количества вычислений, что делает практически невозможным составление второй

схемы с подменой весов и идентичным результатом. Каждому узлу в дереве будет соответствовать свой *nonce*, валидность вычисления будет даваться не только значением вектора но соблюдением критерия сложности подбора *nonce*.

При построении более сложных схем мы предлагаем использовать функцию двойного хеширования sha256d подобно тому, как это делается в хорошо зарекомендовавших себя принципах технологии цепной записи данных (blockchain). Использование одиночной функции создает возможность для снижения вычислительной сложности компиляции схемы доказательства с подменой одного из значений в векторе.

Схема доказательства должна быть разработана с учетом возможности модификации модели с использованием технологии LoRa. Доказательство целостности модели является целостность базовой модели и целостность производной модели с учетом изменений.

Стандартизация блокчейн

- [MP 26.4.001-2018] «Термины и определения в области технологий цепной записи данных (блокчейн) и распределенных реестров»
- [ISO/TC307] Blockchain and distributed ledger technologies

Описание комплекса задач при работе с LLM, требующих ZKP

Как доказать, что модель применена к нашим данным. Как доказать не видя архитектуры, что в нее не внесли изменения и не выполнили какую-то модификацию, включая квантование модели, которая приводит к иным результатам на некоторой итерации работы модели. Даже объединение промпта в батч задание способно дать другой результат. Мы понимаем, что одинаковые условия запуска модели должны давать одинаковый результат. Общая идея - использовать Commit для публикации в блокчейне параметров для последующего доказательства использования модели. При этом схема доказательства не должна раскрывать веса и архитектуру модели.

Доказательством целостности модели является генерация определенной последовательности самой моделью. Однако модели строятся с использованием "температуры генерации" и начальных условий "seed" которые могут влиять на генерацию. Считается, что повышение *температуры* увеличивает креативность генерации. Прежде всего следует убедиться, что современные модели возможно разделить на Encoder-only часть, как в случае BERT, SigLIP и Embedding моделей и выявить промежуточные данные которые могут влиять на результат в семантическом пространстве, такие как квантизация модели и температура. Нужен

математически точный критерий, который можно использовать для сравнения результатов двух моделей.

Модель не изменилась, если при заданной *температуре* и векторе начальных условий *seed* выход в точности совпадает. Однако сравнение в словах токенах, после декодера не вполне корректно. Более корректным является сравнение вектора embedding и вектора выходных признаков output в пространстве семантических признаков. Две модели, даже обладающие разными словарями являются идентичными, если на входной вектор embedding получается идентичный выходной вектор признаков. Такое может достигаться при разных архитектурах сети.

Точность вычислений может порождать ошибку, по этой причине должен существовать некоторый критерий сравнения сетей с разной квантизацией. Например, если в целях ускорения применена квантизованная модель, она может давать результат отличающийся от данного на какую-то среднюю величину квантовой ошибки. Ошибки могут накапливаться, не вполне ясно как это можно измерить. Интуитивно понятно, что бинараная ошибка с однородным или гауссовым распределением не должна приводить к накоплению ошики. Ошибки можно фильтровать и компенсировать. Квантовый шум можно изучать и в тех случаях, когда его можно представить в виде нормального распределения вероятности, можно использовать в методах шифрования. Современные методы шифрования используют добавление квантовых ошибок или детерминированное округление, как метод добавления ошибки. Так вводятся схемы LWE и LWR.

Про модели LLM известно, что они генерируют один вектор семантических признаков за цикл работы модели. Если не менять позицию чтения в ассоциативной памяти (KV-кэше), то LLM должна генерировать одинаковый ответ. Таким образом LLM это функция, которая полностью зависит от входных данных и не имеет встроенных циклов, которые могут изменить состояние непредсказуемым образом. Но при этом понятие времени отсутствует. Где гарантия, что позиция генерации не сдвинулась, не не была выполнена инъекция каких-то контекстных данных от имени третьего участника (наблюдателя). Гарантия может выражаться только в том, что сохраненный контекст дает ту же генерацию. Сравнению подлежит сохраненный контекст и тестовая фраза (в векторном пространстве семантических признаков), которые должны давать прогнозируемый результат.

Далее могут быть варианты: можно рассматривать контекст как сумму двух разряженных контекстов, будет ли при этом результат также суперпозицией двух ответов? Это утверждение верно вблизи некоторого вектора (точки в пространстве семантических признаков) систему можно считать линейной. На этом принципе построено обучение модели, при обучении используются градиенты.

Один из критериев сравнения, принятый в математике - построение матрицы Грамма от сложной функции. Предлагаю провести некоторую аналогию.

Матрица Грама является инструментом для описания структуры и взаимосвязей в сложной системе, представляя собой матрицу скалярных произведений векторов или собственных функций, описывающих элементы системы. Она позволяет выявить зависимость между этими элементами, показывая, насколько они коллинеарны или ортогональны друг другу. В гильбертовом пространстве (в вероятностном пространстве, в аналитической геметрии, в n-мерном пространстве с определенной мерой и скалярным произведением) матрица Грамма полностью характеризует систему.

Характеристикой системы будет реакция на множество ортогональных векторов, число линейно-независимых векторов для описания системы будет равняться размерности вектора слоя `n_embed` . Изучение линейной системы может быть основано на дельта-функции или изучении реакции на ступенчатую функцию во времени. Изучение системы с памятью основано на временных реакциях на ступенчатое возбуждение. Линейной система считается, если реакция системы на сумму воздействий является суммой реакций. Это может не выполняться для моделей LLM, но таков критерий для теста. Следуя этому пути, можно попробовать сопоставить данной модели сети некоторое линейное приближение. Этим обусловлено стремление представить нелинейные функции полиномиальным приближением в надежде, что полиномиальное приближение гарантирует линейные свойства.
Привнесенная идея аппроксимации приводит к ортогональным полиномам, т.е. все та же идея разложения системы по ортогональным функциям и переход к матрице грамма.

Мы предполагаем, что никакие изоморфные преобразования (умножение на матрицу ортогональных преобразований) не меняют модель. Изоморфные преобразования обратимы, если система линейная. Гомоморфные преобразования также строятся на матрице ортогональных преобразований, не обратимы но сохраняют алгебраические свойства системы. Проверка доказательства ZKP будет сроится на паре - функции, которая была преобразована в гомоморфную и результате применения. Именно такой способ проверки используется в схеме Ring-LWE.

Мы предполагаем, некоторое множество реакций системы на тестовые воздействия подобно матрице Грамма могут охарактеризовать нашу систему, вернее её линейное приближение. На базе математического анализа и аналитической геометрии так можно характеризовать системы дифференциальных уравнений.

Защита данных клиента

Inference Privacy и Differencal Privacy

{откуда взялась идея Difference Privacy. Постановка задачи должна исключать возможность манипуляции контекстом. Если запуск модели генерирует контекст, то возможность вырезать фрагменты контекста и заменить на внедренные значения, тем самым можно влиять на результат генерации}.

В целом идеи Difference Privacy мне напоминают шпионские фильмы, где в досье секретного агента все фамилии, адреса и даты зачеркнуты ченым для пущей секретности или заменяются на "Агент Смит". Можно ли такую информацию вообще считать секретной, если контекст можно дополнять и накапливать данными из других источников. Я сомневаюсь, что такое вообще можно называть словом приватность.

> Мне представляется эпизод из фильма "Бегущий по лезвию", где *репликант* отвечает на серию вопросов построенных с применением двусмысленных ключевых слов и контекстных фраз, анализируется реакция *репликанта* в контексте и отклонение реакции от референсной модели. Это примерно тот способ, с использованием которого можно анализировать изменилась ли настройка сети в сравнением с базовой. Видимо для этого требуется отдельное искусство составления синтетических тестов для сравнения сетей. Помимо самой реакции изучается стабильность, т.е. как тест воспринимается в контексте при периодическом повторе. Для этого надо представить, что важна не только реакция на ключевую фразу, но и реакция в контексте и при усилении, повторном использовании синтетичесикого теста или отдельных фраз из контекста. Если проводить аналогии, то необходимо добавлять синтетические тесты способные вызвать зацикливание вывода нейросети или приводящие к неадекватным действиям в контексте. Мне лично проще представить тест когда общение строится между двумя репликантами и запись диалога наполняется периодическими последовательности из синтетических тестов с потерей смысла в контексте. Модель сети при нормальной работе не должна порождать повторяющиеся последовательности при общении с другими нейросетями также как периодические последовательности не должны вызывать периодических ответов и агрессивных (эмоциональных, усилленных) реакций.

Другой подход позволяет выявить граф вычислений, как хеш-функцию по дереву графа. Это возможно если рассматривать каждую операцию в дереве, как некоторый *nonce*, а тензоры операции рассматривать как хеши на входе операции. Такой подход возможен, только если архитектура сети открыта, но это не всегда так. Если бы любую нелинейную операцию в дереве графа можно было бы представить, как полином над множеством целых чисел или чисел с фиксированной точкой, то перейдя к модульной арифметике можно было создать схему шифрования подобно эллиптической криптографии или криптографии на торе. Для построения такой схемы нужно приближение в виде полиномиальных (рациональных) функций. Такое приближение может быть естественным для сетей KAN, в которых используются полиномы в

вероятностном пространстве, полиномы должны образовывать базис. В современной литературе появились работы с использованием для построения сетей KAN классических ортогональных полиномов Чебышева и Якоби, помимо сплайновых сетей с полиномами Бернштейна. Но надо понимать, что построить такую схему можно только зная архитектуру и матрицы весов.

Общая постановка задачи - доказать что к даннм пользователя применили сложную нелинейную функцию множества переменных с нулевым разглашением, без предоставления информации о самой функции. Схема должна включать: 1 - некоторая опубликованная сигнатура модели (Commit); 2 - с воможностью построения проверки доказательтсва использования этой функции (Prove и Verify). Практическая ценность такой схемы достигается при использовании обратимых преобразований, которые не изменяют модель. Если бы доказательство можно было бы свести к запуску шифрованного промпта поверх преобразованной модели и последующему декодированию результата с восстановлением результата в исходный формат в процессе верификации, то такая схема могла бы быть использована для проверки идентичности моделей и безопасного запуска моделей на приватных данных.

Критерий идентичности моделей
Следует уточнить критерий сравнения моделей на основе векторов эмбеддингов. Например, можно использовать критерий косинусной схожести между выходными векторами:

$$\text{cosine\_similarity}(v_1, v_2) = \frac{\langle v_1, v_2 \rangle}{\|v_1\| \cdot \|v_2\|} \, .$$

Если $1 - \text{similarity}(v_1, v_2) < \epsilon$, модели считаются идентичными.

Косинусная схожесть и дистанция в векторном пространстве семантических признаков - хороший начальный критерий, на базе которого строятся более сложные критении сравнения, использующие динамическое центрирование функции распределения, маски контрастности и весовые коэффициенты. В конечном счете может быть подобран критерий подобный функции Cross-Entropy-Loss, используемой при обучении модели.

см. отдельный обзор по Методам сравнения моделей по семантичесим признакам

Предлагается рассмотреть использование матрицы Грамма для сравнения моделей в пространстве семантических признаков:

$$G_{ij} = \langle u_i, v_j \rangle, \quad u_i, v_j \in \text{embeddings}$$

Пространство семантических признаков можно охарактеризовать набором линейно-независимых и нормированных embedding-векторов. Таким образом, матрица Грамма может

быть использована для оценки сходства между моделями, каждая операция эквивалентна косинусной схожести пары векторов.

Это позволяет выявлять линейные зависимости между выходами моделей и изучать стабильность моделей.

{Добавить обсуждение, как фиксировать температуру и seed для воспроизводимости результатов, и как это влияет на ZKP-верификацию. Куда входит настройка температуры генерации. Как влияет квантизация модели на сравнение по семантической схожести.}

Мы исходим из того, что клиент может подавать данные на вход с определенным контекстом, сохранять и восстанавливать контекст используя прореживание и читать вектор состояния на выходе системы. Идеально для безопасности запуска было бы разделить данные на фрагменты, преобразовать входные данные с использованием матриц, известных только клиенту и таким образом получить безопасный способ запуска сети. Опять таки, это возможно только для линейных LTI-систем и систем построенных на рациональных функциях (аппроксимация Падé).

{Я вполне осознаю, что часть моих тезисов не обоснована должным образом - это скорее интуитивное понимание основанное на курсе математического анализа и курсе аналитической геометрии и методов лежащих в основе машинного обучения. Сетей построенных на рациональных функция, на рациональных функциях в State-Space пространстве, одновременно отвечающем всем требованиям вероятностного пространства, как и методов отображения LLM сетей в сети State-Space в настоящее время не существует. На сегодня это можно считать футуристическим прогнозом.}

- [2303.00654] How to DP-fy ML: A Practical Guide to Machine Learning with Differential Privacy
- [2407.12108] Private prediction for large-scale synthetic text generation
- [2506.04566] Clustering and Median Aggregation Improve Differentially Private Inference

> Differentially private (DP) language model inference is an approach for generating private synthetic text. A sensitive input example is used to prompt an off-the-shelf large language model (LLM) to produce a similar example. Multiple examples can be aggregated together to formally satisfy the DP guarantee.

Semantic Web наравне с приложениями технологии zkp распространяется концепция построения безразмерной сети ориентированной прежде всего на машинные методы обработки информации, как развитие сети Internet WWW. Дело в том, что ресурсы и поиск в сети в настоящее время смещают акцент на пригодность ресурсов сети для машинного обучения и

генерации. Вместе с тем развивается идея децентрализации функций поиска в сети с использованием технологий LLM и RAG.

В сочетании слов *Semantic Web* я вижу возможность поиска и сравнения контента по семантическим признакам, который позволяет подбирать подходящий контент под запрос пользователя. Понятным на сегодняшний день является технология сравнения векторов полученных с использованием Embedding и Reranking LLM и (RAG) поиска. Что это дает? Поисковая технология может быть делегирована в форме модели LLM с открытыми весами.

ZKP/FHE для RAG:
Retrieval-Augmented Generation (RAG) требует верификации результатов поиска. ZKP может использоваться для доказательства, что результаты поиска соответствуют запросу, не раскрывая содержимое базы данных и весов модели.

Использование ZKP (Zero-Knowledge Proofs, доказательства с нулевым разглашением) и FHE (Fully Homomorphic Encryption, полностью гомоморфное шифрование) в контексте RAG (Retrieval-Augmented Generation) позволяет обеспечить конфиденциальность данных и безопасное выполнение вычислений в системах генерации текста с использованием внешних данных.

RAG — это архитектура, которая комбинирует извлечение релевантной информации из внешнего источника данных (например, базы знаний) с генерацией текста с помощью языковой модели. Выборка данных выполняется с использованием embedding векторов от документов и запроса пользователя в пространстве семантических признаков. Для генерации векторов для сравнения используется Embedding и Reranking LLM.

Более детальные RAG системы могут учитывать графы связей между документами и запросом, что позволяет улучшить точность поиска.

- [2501.00309] Retrieval-Augmented Generation with Graphs (GraphRAG)

FHE позволяет выполнять поиск по зашифрованным данным, что защищает конфиденциальность запросов и базы данных. {надо разъяснить на уровне схемы} Если не углубляться в математические принципы, сравнение ембеддингов при выборке данных выполняется с использованием скалярного произведения (косинусной схожести в пространстве семантических признаков) от зашифрованных векторов. Зашифрованные эмбеддинги могут работать, как вектор хешей для фильтрации данных при распределенном хранении. {тезис надо обосновать}

Vector Search. A retrieval method that finds data similar to a query by comparing high-dimensional numerical vectors

These vectors are stored in a vector index using libraries such as FAISS (Facebook AI Similarity Search), ElasticSearch or Annoy, which allow for efficient searching in high-dimensional space. Vector search finds similar data using approximate nearest neighbor (ANN) algorithms.

Векторный поиск и алгоритмы ANN (Approximate Nearest Neighbors, приблизительный поиск ближайших соседей), включая kNN (k-ближайших соседей), являются ключевыми технологиями в обработке больших объемов данных.
алгоритмы и структуры данных:

- *KD-деревья*: Разделяют пространство на регионы для быстрого поиска. Хорошо работают в низкоразмерных пространствах, но теряют эффективность при высокой размерности.
- *Ball-деревья*: Улучшение KD-деревьев, использующее гиперсферы для разделения пространства.
- *LSH (Locality-Sensitive Hashing)*: Хэширует векторы так, что близкие точки с высокой вероятностью попадают в одну "корзину". Подходит для высокоразмерных данных.
- *HNSW (Hierarchical Navigable Small World)*: Графовый алгоритм, использующий многослойные графы для эффективного поиска. Один из самых популярных методов в современных системах (например, в библиотеке FAISS).
- *Product Quantization (PQ)*: Сжимает векторы, разделяя их на подмножества и кодируя каждое подмножество. Используется для экономии памяти и ускорения поиска.
- *ANNOY (Approximate Nearest Neighbors Oh Yeah)*: Использует случайные проекции и бинарные деревья для построения индекса.

Locality Sensitive Hashing (LSH) is a technique that efficiently approximates similarity search by reducing the dimensionality of data while preserving local distances between points.

Мы выбираем технику для высокой размерности векторов, соответствующей Embedding векторам, порядка тысячи элементов. В качестве хэш функций предлагается использовать модульную арифметику. Множество хэш функций при этом различается только начальной точкой и модулем. Модульная арифметика локально сохраняет дистанцию и может использоваться как основа LSH.

- [1603.09320] Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs
- [1702.08734] Billion-scale similarity search with GPUs
- [2309.15479] Fast Locality Sensitive Hashing with Theoretical Guarantee, 2023
- (https://github.com/erikbern/ann-benchmarks)

LSH for ℓ_2 Norm

The hash function is defined as follows:

$$h_{\mathbf{a},b}(\mathbf{v}) = \left\lfloor \frac{\mathbf{a}^T\mathbf{v} + b}{w} \right\rfloor$$

where $\mathbf{a}$ is a n-dimensional vector with each entry chosen independently from $\mathcal{N}(0,1)$ and b is a real number chosen uniformly from the range $[0, w]$.
w is an important parameter by which one could tune the performance of E2LSH.

Хэш функция определяется через скалярное произведение - проекцию. Множество хэш-функций образуют базис, по которому раскладывается embedding вектор.

Библиотеки для ANN:

- *FAISS* (Facebook AI Similarity Search): Высокоэффективная библиотека для поиска в больших наборах векторов, поддерживает HNSW, PQ и другие методы.
- *ANNOY*: Легковесная библиотека от Spotify, оптимизированная для быстрого поиска.
- *HNSWlib*: Специализируется на HNSW, высокая производительность.
- *Elasticsearch* и *OpenSearch*: Поддерживают векторный поиск для задач обработки текстов и мультимедиа.
- *Milvus*: Распределенная система для векторного поиска, интегрируется с HNSW, PQ и другими алгоритмами.

Проекция вектора на гиперплоскость в семантическом пространстве эмбеддингов — это процесс нахождения ближайшей точки на гиперплоскости к заданному вектору. В контексте эмбеддингов, которые часто представляют семантическую информацию (например, слова или предложения в пространстве признаков), проекция может использоваться для выделения определённых аспектов или упрощения данных.

Математическое описание
Пусть *Гиперплоскость* в n-мерном пространстве определяется уравнением:

$$\langle \mathbf{w}, \mathbf{x} \rangle + b = 0,$$

где $\mathbf{w}$ — вектор нормали к гиперплоскости, $\mathbf{x}$ — точка на гиперплоскости, b — свободный член.

Формула проекции: Проекция вектора $\mathbf{v}$ на гиперплоскость определяется как:

$$\mathbf{v}' = \mathbf{v} - \frac{\langle \mathbf{w}, \mathbf{v} \rangle + b}{\|\mathbf{w}\|^2} \mathbf{w},$$

где $\langle \mathbf{w}, \mathbf{v} \rangle$ — скалярное произведение, $\|\mathbf{w}\|^2 = \langle \mathbf{w}, \mathbf{w} \rangle$ — квадрат ℓ_2-нормы.

- Метод главных компонент (разложение Карунена-Лоева, principal component analysis, PCA)
- Метод главных компонент через критерий максимизации разброса
- (http://www.machinelearning.ru/wiki/images/a/a4/MOTP11_5.pdf)

Алгоритм (Схема метода главных компонент)

1. Построение матрицы ковариаций (дать вероятностное определение):

$$\mathbf{Q} = \frac{1}{n-1}\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T = \frac{1}{n-1}\mathbf{X}^T\mathbf{X}$$

2. Приведение симметричной матрицы к диагональному виду $\mathbf{Q} \propto \mathbf{X}^\mathsf{T}\mathbf{X} = \mathbf{W}\mathbf{\Lambda}\mathbf{W}^\mathsf{T}$. $\Lambda = diag(\lambda_1 ... \lambda_n)$ - диагональная матрица собственных значений.
3. Выбор главных компонент: выбираются собственные векторы, соответствующие наибольшим собственным значениям.
4. Проекция данных на главные компоненты: $\mathbf{Y} = \mathbf{X}\mathbf{W}_k$, где k — число главных компонент.

Центральный вопрос, возможно ли использовать вектор зашифрованных данных обладающий свойствами LSH для семантического поиска. В этом случае следует преобразвать алгоритм разложения к комплексным числам (используя каноническое вложение или в пространство полиномов $\mathcal{R}_q^2$) и использовать его для построения проекции на гиперплоскость. Второй вопрос, можно ли привести построение матрицы ковариаций к KV-кэшу LLM. Для этого нужно сравнить метод заполнения KV-кэша с методом построения матрицы ковариаций.

Заметим, что в линейной алгебре основной метод диагонализации симметричной матрицы - это метод LDL^T-разложения Холецкого.

The Cyclotomic Ring and Canonical Embedding

Довольно простая интуитивно понятная схема - криптография с использованием часовой стрелки. Операции сложения и умножения выполняются по часовой стрелке, модулем является количество делений на циферблате. Мы измеряем время в минутах и секундах, по модулю q=60. Каноническое вложение (canonical embedding) переводит секунды (целые числа по модулю) в синусы и косинусы углов - проекции угла, на которые отклонилась стрелка, или комплексные числа. Такой способ отображения целых чисел в комплексные числа называется каноническим вложением (canonical embedding). Между вещественными числами и целыми числами по модулю q существует необратимое отображение, гомоморфизм. Отображение может быть обратимо, если для декодирования сохранять число полных оборотов стрелки.

Гомоморфизм — это отображение между двумя алгебраическими структурами (например, группами, кольцами, векторными пространствами), которое сохраняет операции.

Определение:
Пусть $(A, +, \cdot)$ и $(B, \oplus, \otimes)$ — два кольца. Отображение $\sigma : A \to B$ называется гомоморфизмом колец, если:

- $\sigma(a_1 + a_2) = \sigma(a_1) \oplus \sigma(a_2)$ (сохранение сложения),
- $\sigma(a_1 \cdot a_2) = \sigma(a_1) \otimes \sigma(a_2)$ (сохранение умножения),
- $\sigma(1_A) = 1_B$ (сохранение единицы, если кольца с единицей).

Если рассматривать только группы (например, с одной операцией, скажем, сложением), то достаточно первого условия.

В общем случае каноническое вложение $\sigma : \mathcal{R} \to \mathbb{C}^n$ (где $n = \phi(m)$ - Euler totient function) для m- циклического кольца $\mathcal{R} = \mathbb{Z}[x]/\langle \Phi_m(x) \rangle$ является гомоморфизмом колец. Оно отображает элементы кольца в векторы, сохраняя операции сложения и умножения.

Основное свойство, которые выводится из канонического вложения - это сохранение нормы. Для любого $a \in \mathcal{R}$ и $b \in \mathbb{C}^n$ выполняется $\|\sigma(a)\|_2 = \|a\|_2^{can}$. Это свойство позволяе сравнивать эмбединги полученные в результате преобразования по норме и применять шифрованные данные в обучении или сортировке данных.

Каноническое вложение:
Рассмотрим циклотомическое поле $K = \mathbb{Q}(\zeta)$, где ζ — примитивный m-й корень единицы, а $\Phi_m(x)$ — m-й циклический полином. Кольцо целых чисел поля — $\mathcal{R} = \mathbb{Z}[x]/\langle \Phi_m(x) \rangle$, степень поля $n = \phi(m)$. Каноническое вложение $\sigma : R \to \mathbb{C}^n$ отображает полином $a(x) \in \mathcal{R}$ в вектор:

$$\sigma(a) = (a(\zeta_1), a(\zeta_2), \ldots, a(\zeta_n)) \ ,$$

где ζ_i — примитивные m-е корни единицы. Вектор $\sigma(a)$ лежит в пространстве $\mathcal{H} \subset \mathbb{C}^n$, которое учитывает комплексное сопряжение (например, $\sigma_{j+n/2}(a) = \overline{\sigma_j(a)}$), и $\mathcal{H} \cong \mathbb{R}^n$.

Сохранение Нормы
Норма, о которой обычно говорят в контексте канонического вложения, — это евклидова норма вектора $\sigma(a)$ в $\mathcal{H}$:

$$\|\sigma(a)\|_2 = \sqrt{\sum_{j=1}^{n} |a(\zeta_j)|^2} \ .$$

Эта норма связана с алгебраическими свойствами полинома $a \in \mathcal{R}$ и используется для анализа ошибок в Ring-LWE. Сохранение нормы означает, что $\|\sigma(a)\|_2$ даёт информацию о "размере" элемента a, которая согласуется с его представлением в кольце, хотя напрямую норма в кольце и в пространстве $\mathcal{H}$ не идентичны, а связаны через определённые коэффициенты.

Для циклических колец с $m = 2^k$ (где $\Phi_m(x) = x^{m/2} + 1$) каноническое вложение имеет особое свойство: норма $\|\sigma(a)\|_2$ пропорциональна норме полинома $a(x)$ в смысле его коэффициентов. {утверждение требует доказательства}

{Я бы хотел в явном виде определить прямое и обратное отображение из $\mathbb{R}^n \to \mathbb{Z}_q/\langle\Phi_m(x)\rangle$. Раскрыть как понятие canonical embedding используется в схеме CKKS. В определении канонического вложения есть множество рациональных чисел и множество комплексных чисел. Как это связано с кольцом целых чисел по модулю q.}

По ссылке можно разобрать разделы A-7.Roots of Unity и A-8.Cyclotomic Polynomial:

- [2503.05136] The Beginner's Textbook for Fully Homomorphic Encryption (https://fhetextbook.github.io/)

Корень n-й степени $\omega^n \equiv 1 \mod q$. и $\omega^{n/2} \neq 1 \mod q$.

Если число ω является корнем n-й степени, то $\omega^{n/k}$, является корнем k-й степени. {мне удалось воспользоваться этим утверждением, так родился алгоритм поиска корня.}. Корень 2^s-ой степени найти можно методом подбора, исопльзуя критерий $\omega^{n/2} \neq 1 \mod q$. Условием существования 2^s-ого корня является существование квадратного корня из -1.

Схема CKKS: Approximate FHE

Пусть $m \in \mathcal{R}$ - не шифрованное сообщение (plaintext). Заданный секретный ключ $sk \in \mathcal{R}$, и некоторый R-LWE шифротекст (ciphertext) ct шифруют сообщение m как пару $ct = (b, a) \in \mathcal{R}_Q^2$ так чтобы $[ct \cdot sk]_Q = m$.

Схема работает в двух пространствах: (plaintext) сообщение $m(x) \in \mathcal{R} = \mathbb{Z}[x]/\langle x^N + 1\rangle$ и $c(x) \in \mathbb{C}^{n/2}$. Для перехода между пространствами используется каноническое вложение $\sigma : \mathcal{R} \to \mathbb{C}^{n/2}$ и метод DFT (Discrete Fourier Transform).

Пусть DFT представляет собой преобразование $\mathbb{R}[x]/\langle x^N + 1\rangle \to \mathbb{C}^{N/2}$.

$$\mathsf{DFT}(p(x)) = (p(\zeta_i))_{0 \le i < N/2}$$

где $\zeta_i = \zeta^{5^i}$, ζ - корень $2N$-й степени из единицы.

На базе DFT определяется операция декодирования $\mathsf{Dec} : \mathcal{R} \to \mathbb{C}^{N/2}$

$$\mathsf{Dec}(m) = \frac{1}{\Delta}\mathsf{Dec}(m)$$

где Δ - масштабный множитель (scaling factor). Масштабный множитель используется для нормализации и дескретизации значений.

Пусть iDFT обратное преобразование $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{R}[x]/\langle x^N + 1\rangle$.
Операция кодирования $\mathsf{Ecd} : \mathbb{C}^{N/2} \to \mathcal{R}$ определяется как

$$\mathsf{Ecd}(\vec{z}) = \lfloor \Delta \cdot \mathsf{iDFT}(\vec{z}) \rceil$$

Заметим что Ecd является обратной операцией к Dec и устанавливают соотвествие между $\mathcal{R}$ и $\mathbb{C}^{N/2}$. Операции DFT и iDFT сохраняют норму с фиксированным коэффициентом (свойство изометрии по норме ℓ_2):

$$\|DFT(p(x))\|_2 = \sqrt{\frac{N}{2}} \cdot \|p(x)\|_2$$

для любого $p(x) \in \mathcal{R}$.

Используется представление в модульной арифметике $\mathcal{R}_Q$ такое, что выбранный модуль делит $q|Q$. Это представление позволяет управлять разрядностью чисел. Для управления разрядностью применяется операция *Rescaling* и *Modulus Switching*.

Inference Privacy {перенести в свой раздел}

Одним из способов защиты систем машинного обучения от широкого спектра существующих атак является построение систем конфиденциального машинного обучения с использованием схем гомоморфного шифрования.

- [2411.18746] Inference Privacy: Properties and Mechanisms

{Добавить описание атак на inference (например, model inversion, membership inference) и как FHE/ZKP их предотвращают. Например, указать, что FHE позволяет выполнять inference на зашифрованных данных, не раскрывая входные данные или параметры модели.}

Форматы данных, Сериализация

Построение схем *ZKP* и *FHE* требует стандартизации. Прежде всего должен быть выбран способ канонизации графа вычислений (computation graph) и канонического представления тензоров, бинарный формат сериализации. Слендует отметить определенность сериализации (deterministic) не означает каноническое (canonical) представление графа, поскольку

сериализация допускает варианты кодирования одного и того же содержимого разными методами или разными длинами данных.

Сериализация важна для обмена данными по сети. В частности в контексте удаленного запуска процедур и обмена массивами данных (тензорами). В ходе обмена возникают задачи проверки целостности данных, и кэширования тензоров, для исключения повторной передачи больших объемов информации.

Для майнинга крипто-валют применяется JSON-RPC протокол Stratum.

gRPC (Google Remote Procedure Call) — это современная, высокопроизводительная, открытая система удаленного вызова процедур, разработанная Google. Она позволяет приложениям взаимодействовать друг с другом, вызывая функции на удаленных серверах, как если бы они были локальными. gRPC использует HTTP/2 для передачи данных и Protocol Buffers для сериализации сообщений.

- Google Protobufs Protocol Buffers

> Protocol Buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data.

С другой стороны существуют структурированные форматы данных предназначенные для обмена данными в сети интеренет и в частности CBOR, являющийся интернет стандартом [STD94]. Мы предлагаем рассматривать формат CBOR в сочетании с протоколом HTTP и CoAP. CoAP позволяет реализовывать подписку на события и предлагает механизм уведомлений в течение сессии.

Особенностью форматов пригодных для RPC является возможность однозначного преобразования из бинарного в текстовый формат (JSON) и представление в текстовом структурированном виде пригодном для отладки сообщений, а также возможность описания и верификации схемы документа.

Формат сериализации Protobuf имеет представление в виде JSON и тесктовое представление. Однако ни одно из этих представлений не является каноническим. Так же как и в случае XML-Security текстовый формат должен приводиться к каноническому представлению до применения процедуры подписи или хеширования.

Форматы сериализации плохо предназначены для квантизованных данных. В CBOR определены правила сериализации типизованных массивов. Правила сериализации графов не стандартизованы и в большинстве случаев описание графа не является частью формата. Можно выделить формат ONNX, который содержит описание графа с использованием

множества тензорных операций и имеет под собой кодирование protobuf. Среди форматов для представления тензоров моделей LLM следует отметить два: GGUF и Safetensors. GGUF - бинарный формат допускающий типизацию тензорных данных оптимизированный под загрузку данных без предварительной обработки. Safetensors включает текстовые метаданные в формате JSON и не является достаточно детерминированным. Оба формата не предназначены для RPC и не содержат представление графа вычислений. Чаще всего модели рождаются в формате PyTorch (с сериализацией picle) и затем конвертируются в один из представленных форматов для запуска моделей или публикации весов.

GGUF формат

Для эффективного распределения заданий с использованием RPC формат должен содержать каноническое сериализованное описание графа предназначенное для передачи по сети. В существующей реализации сериализация графа выполняется налету исходя из внутреннего представления данных в GGML. Это не оптимальный путь - требуется обработка данных и сопоставление идентификаторов. Каждому тензору необходимо сопоставить криптографический хеш в выбранном формате `SHA256`, `SHA3-256` или `SHAKE256` от merkle-tree образованного из хешей тензоров - аргументов операции. Кроме того в нашей реализации для целей кэширования и проверки целостности больших объемов данных необходим некрипторграфический хеш. Для проверки целостности тензорных данных в GGML выбран хеш `xxh64` и `FNV-1`. Мы предлагаем ввести вектор хешей одновременно пригодный для сравнения embedding векторов и для поиска локальных копий на серверах.

В нашей постановке здачи, для безопасного запуска модели нейросети с использованием RPC, следует подобрать хеш функцию удовлетворяющую свойствам $\mathbb{Z}_q^n$. Для этих целей нами было предложено использовать простые числа и методы *multiply-with-carry*: `MWC32`, `MWC64`, `MWC128`, которые эффективно могут быть реализованы на GPU. Изначально алгоритм MWC предназначен для детерминированной генерации псевдослучайных чисел PRNG. По сути метод MWC представляет собой умножение чисел по модулю простого числа. Простое число выбирается исходя из возможности оптимизации операции редуцирования. Алгебраические свойства MWC позволяют считать хеши параллельно с заданным смещением в массиве данных (множителем), подобно методике `folding` для CRC. Таким образом хеши MWC могут быть использованы для реализации фильтра Блума при поиске тензоров в локальном кэше и для проверки целостности тензорных данных с высокой степенью параллельности вычислений. Алгоритмы MWC основаны на модульной арифметике и гарантируют последовательность без повторов.

- (https://prng.di.unimi.it/)
- (https://github.com/AnatolyGeorgievski/MWC128)

Выбор генератора псевдослучайных чисел PRNG

В методике LWE и Ring-LWE используются генераторы псевдослучайных чисел PRNG с нормальным (гауссовым) распределением вероятности ошибки. В качестве PRNG нами используется генератор `MWC64` с однородным распределением вероятности при случайном выборе seed. Генератор дает однородное распределение вероятности, которе преобазуется в нормальное распределение вероятности с помощью преобразования Box-Muller.

- Box-Muller Трансформация Бокса-Мюллера
- (https://github.com/AnatolyGeorgievski/MWC128/blob/main/Gaussian.md)

Преобразование Box-Muller позволяет получить ошибку с гауссовским распределением вероятности. А такаже можно получить начальные условия в задачах статистики и молекулярной динамики, распределение Максвелла. Я упоминаю это распределение, как частный случай позволяющий создавать нормальные распределения векторов в n-мерном векторном пространстве с заданной *температурой* распределения.

Дискретное гауссово распределение

Пусть $n \in \mathbb{Z}^+$, $\sigma \in \mathbb{R}^+$ представляют стандартное отклонение, и среднее $c \in \mathbb{R}^n$. Определим

$$\rho_{\sigma,c}(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{c}\|^2}{2\sigma^2}\right)$$

где $\mathbf{x} \in \mathbb{R}^n$. Тогда дискретное гауссово распределение определяется как

$$D^n_{\sigma,c} = \rho_{\sigma,c}(\mathbf{x})/\rho_{\sigma,c}(\mathbb{Z}^n),$$

где $\rho_{\sigma,c}(\mathbb{Z}^n) = \sum_{\mathbf{x}\in\mathbb{Z}^n} \rho_{\sigma,c}(\mathbf{x})$. Если $c = 0$, мы опускаем c и обозначаем распределение как D^n_σ.

Алгоритм (CBD) Sempling polynomial from the centered binomial distribution

Параметры: $q \in \mathbb{Z}^+$, $\eta \in \mathbb{Z}^+$
Вход: Массив байтов $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in \mathbb{B}^{64\eta}$
Выход: Полином $f \in \mathcal{R}_q$

1. $(b_0, \ldots, b_{512\eta-1}) := BytesToBits(B)$
2. for i from 0 to N do
3. $\quad a_i := \sum_{j=0}^{\eta-1} b_{2i\eta+j}$
4. $\quad b_i := \sum_{j=0}^{\eta-1} b_{2i\eta+\eta+j}$

5. $f_i := a_i - b_i \pmod q$
6. end for
7. return $f_0 + f_1X + f_2X^2 + \cdots + f_{N-1}X^{N-1}$

Sampling from the centered binomial distribution. Алгоритм работает на потоке бинарных данных с однородным распределением вероятности нулей и единиц и выдает центрированный биномиальный вектор с заданной дисперсией значений $[-\eta, \eta]$.

$$CBD_\eta = \sum_{i=0}^{\eta-1} b_i - \sum_{i=0}^{\eta-1} b_{i+\eta}$$

Например, для $\eta = 1$ функция выдает тернарные вектора $\{-1, 0, 1\}$.
Функция CBD используется в ML-KEM и других криптографических схемах PQC.

Альтернативно можно рассмотреть обобщенный алгоритм генерациии центрированного биномиального распределения, используя k бит и усреднение (фильтр).

Алгоритм (GCBD Sample Algorithm)

Параметры: $k, \ell \in \mathbb{Z}^+$
Выход: выборка $GCBD_{k,\ell}$

```
x := 0
for i = 1 to \ell do
    t, s \leftarrow B^{(k)}
    x = x + (t - s)
end for
return x
```

Представление графа тензорных операций

Что из себя представляет граф тензорных операций. Сериализованный граф - это таблица, массив данных, содержит идентификаторы операций, контекстные идентификаторы тензоров, входных и выходных тензоров. Мы можем ограничить число тензоров на входе и рассматривать только бинарные или унарные тензорные операции. Каждая тензорная операция может содержать список параметров операции. Могут быть определены макро операции такие как MLP, FFN, GRU, LSTM, Attention и др. Каноническое представление графа тензорных операций должно регламентировать использование макро операций и списки параметров операций.

Для записи параметров тензорных операций следует использовать список полей, в котором используется позиционное кодирование идентификаторов параметров, а некоторые парметры принимают значения по-умолчанию исключаются из списка. В этом смысле подхожит способ кодирования параметров protobuf или MAP в формате CBOR.
Для каждой тензорной операции должна быть определена схема данных для формальной проверки и канонизации сериального представления параметров операции. В языке Protobuf для этого используется текстовое описание схемы данных.

Описание графа тензорных операций допускает нарушение порядка выполнения операций, порядок будет определяться структурой графа, а не порядком следования записей в таблице графа. Это согласуется с принципами FHE и обеспечиватся коммутативными свойствами операций.

Сериальное представление параметров тензорной операции формируют `extranonce` для фнукции хеширования. Результатом расчета является криптографический хеш от `extranonce` и от хешей полученных от аргмуентов тензорной операции. Параметром операции может являться поле флагов. Я использую понятие экстранонс из блокчейна, как поля для фиксации настроек модели, которое наравне с хешами образуют сообщение для расчета финального хеша. Под `nonce` я понимаю вектор бинарных случайных чисел, который используется для критерия сложности подбора хеша.

Под тензором в графе понимается выходное значение тензорной операции или многомерный типизованный массив данных без операции. Тензор без операции на входе графа может быть результатом предудущего цикла расчета графа тензорных операций, может содержать данные клиента или содержать *замороженные* веса модели. Каждый тензор в блокчаине имеет критографический хеш, который используется для доказательства целостности данных и может содержать вектор некриптографических хешей для контроля операций с кэшем и выполнения поиска.

При соблюдении принципов blockchain (децентрализованный реестр с цепной записью данных), можно потребовать определенной сложности формирования криптографического хеша от списка хешей - аргументов операции, и ввсети `nonce`, удовлетворяющий критерию сложности. В качестве основной функции хеширования мы рассматриваем операцию SHA256 и двойного SHA256d при расчете с использованием `nonce`. В будущих реализациях мы допускаем замену операции SHA256 на другие алгоритмы хеширования удовлетворяющие критерию сложности и принципам *blockchain* в условиях пост-квантовой эпохи. Одним из таких алгоритмов является SHAKE256 [FIPS202].

Математическое описание:
Пусть T — тензор, $H(T)$ — криптографический хеш (например, SHA256), а $nonce$ —

случайное значение, выбираемое для удовлетворения критерия сложности (аналогично PoW в блокчейне). Тогда для тензорной операции с двумя аргументами $T_{out} = op(T_1, T_2)$, результат хеширования определяется как:

$$H_{op} = SHA256d(H(T_1) \,||\, H(T_2) \,||\, H(T_{out}) \,||\, nonce \,||\, extranonce) \ ,$$

где $extranonce$ — параметры операции, а $||$ — конкатенация байтовых строк. Критерий сложности может быть задан через выбор *nonce*, как требование, чтобы H_{op} начинался с k нулевых битов.

{Сравнить схему с XMSS.}

Тензор в RPC протоколе может содержать хеш SHA256, а также вектор некриптографических хешей для контроля операций с кэшем. Локальный Идентификатор тензора используется для привязки в графе тензорных операций. Локальный идентификатор тензора является уникальным в течение сессии работы с графом тензорных операций. Принцип формирования локального идентификатора тензора должен удовлетворять правилам кодирования SDNV для представления в бинарных структурированных форматах и иметь отображение в пространство имен текстовых идентификаторов тензоров весов и смещений с учетом индексов слоев модели. Длина бинарного SDNV идентификатора должна быть ограничена 8-ю байтами.

Броадкаст тензорных данных

В сушествующей практике тензорных вычислений, операции могут работать с аргументами - тензорами кратной размерности. В таких случаях выполняется броадкаст операции тензора - расширение тензора до требуемой размерности. Каноническое представление тензора в операции с броадкастом - это тензор минимальной кратной размерности, которая может быть использована в качестве аргумента операции.

- Numpy: General broadcasting rules
- ONNX: Broadcasting

Заметим, что в реальности размерность тензорных данных не превышает 4-х. Обчыно мы работаем с двумерными тензорами. Две другие размерности возникают в процессе предобработки данных, например, в случае выделения фрагментов изображения, патчей для независимой обработки. Четвертая компонента возникает в пакетных заданиях, при объединении нескольких массивов или нескольких каналов данных в одну операцию, для параллельной обработки. При обработке промптов данные могут быть скомпонованы в пакетное задание, в то время как генерация текста выполняется по одному токену за цикл.

Мы акцентируем внимание на представлении тензора в виде многомерного массива данных с возможностью броадкаста поскольку это создает варианты получения одного результата с использованием модели представленной в виде графов с различной структурой, только потому что данные и тензоры нарезаны и перемешаны по-разному. Так например данные могут быть перемешаны на входе и к ним применятся матрица весов с перестановкой столбцов. В идеале все модели, которые дают одинаковый результат должны быть представлены в виде графов с одинаковой структурой в результате канонизации графа.

Манифест

Хранение тензоров модели сопровождается манифестом - отсоединенным текстовым файлом, который содержит информацию о тензорах и хешах модели. Каждому тензору может быть сопоставлен криптографический хеш типа `SHA256` и список некриптографических хешей, например `xxh64` , `fnv-1` или `mwc64` . Привязка к тензору выполняется с использованием канонического локального имени тензора или соответствующего `SDNV` идентификатора [RFC6256].

{Расширить понятие Manifest до CMS, допустить возможность хранения обязательств и подписей,сравнить с форматом PGP с акцентом на XMSS}

Типы тензоров low-precision**

Тензоры в формате представляются в виде типизованных многомерных массивов данных (contiguous, без перемешивания permute и без выборки view). Порядок передачи тензоров по сети не имеет значения и может быть определен в процессе расчета от графа тензорных операций. Необходимость пересылки тензора по-сети определяется с использованием хешей.

- [2206.02915] 8-bit Numerical Formats for Deep Neural Networks
- [2209.05433] FP8 Formats for Deep Learning
- [2302.08007] With Shared Microexponents, A Little Shifting Goes a Long Way

-- Вводится характеризация квантового шума (QSNR), возникающего при квантизации и представлен compute flow graph используемый при обучении на квантизованных данных. По сути тут вводится методика компенсации квантовой ошибки, которую мы хотим реализовать в наших алгоритмах. Мы проводили свои тесты с использованием принципов квантизации Microscaling (с общей экспонентой) и пришли к выводу, что квантизация MXINT8 и MXFP8 дает хорошие показатели при квантизации весов моделей.

- [2310.10537] Microscaling Data Formats for Deep Learning

-- Помимо самих форматов в статье рассматриваются результаты обучения на квантизованных данных и результаты пост квантизации с диффузией ошибки (PTQ).

- [2506.08027] Recipes for Pre-training LLMs with MXFP8
- [OCP:MX] OCP Microscaling Formats (MX) Specification Version 1.0

Тернарная квантизация

В настоящее время тернарные типы данных не стандартизированы и применяются в двух основных направлениях: светочные сети с тернарной квантизацией (TNN) и модели языковых сетей на тернарной квантизации [BitNET][BitCPM4] Для языковых моделей BitNET предложен ряд форматов для представления тензоров в формате GGUF.

В данном контексте мы выделяем квантовые нейронные сети (QNN) для которых характерно использование нормировки и тернарных квантизаторов с компенсацией ошибки округления после каждой активации и после каждой тензорной операции.

Определение тернарной квантизации.

- [2405.03144] PTQ4SAM: Post-Training Quantization for Segment Anything

-- Квантизация PTQ определяется примерно так:

$$x_q = \text{clamp}(\left\lfloor \frac{x}{s} \right\rceil + z, 0, 2^N - 1)$$
$$\tilde{x} = s \cdot (x_q - z) \approx x$$

При этом получаем целочисленный метод квантизации подобный `Q8` .
Немного иначе будет выглядеть квантизация для MX - Microscaling Formats, для которых выделяется общая экспонента `E8M0` и используется операция округление RTE с насыщением, финитная арифметика. Подобные операции хорошо согласованы с архитектурой тензорных ядер GPU.

Тернарная квантизация определяется так:

$$\tilde{x} = \min(\max(\left\lfloor \frac{x}{s} - z \right\rceil, 0), Q)$$
$$s \cdot (\tilde{x} + z) \approx x$$

Квантизованные значения $\tilde{x} \in \mathbb{T} : \{-1, 0, 1\}$ для тернарной квантизации и
$\tilde{x} \in \mathbb{B} : \{-1, 1\}$ для бинарной квантизации.
При этом значения $\tilde{x}$ кодируются как пара (x^+, x^-): $-1 = (0, 1)$; $0 = (0, 0)$; $+1 = (1, 0)$.

Арифметические операции в тернарной логике

Операция умножения может быть определена с использование побитовых логических операций

- тернарно-тернарная операция умножения:

$$(c^+, c^-) = (a^+ \wedge b^+) \vee (a^- \wedge b^-)\,, \quad (a^+ \wedge b^-) \vee (a^- \wedge b^+)$$

- тернарно-бинарная операция умножения:

$$(c^+, c^-) = (a^+ \wedge b) \vee (a^- \wedge \bar{b})\,, \quad (a^+ \wedge \bar{b}) \vee (a^- \wedge b)$$

Заметим, что возможно иное представление тернарных данных в виде пары (знак, экспонента), `E1M0` .

Операцию сложения мы рассматриваем в составе операции *dot product* и она сводится к суммированию значений числа положительных и отрицательных значений (+1) и (-1) на векторе результата умножения. Базовой является операция подсчета ненулевых бит в бинарном векторе, операция `popcount` . Операция в языке программирования C представлена псевдо функцией `__builtin_popcount()` , поддерживается компиляторами, включая LLVM и GCC. Поддерживается в реализациях OpenCL C. Заметим, что в системе команд Intel AVX512-VPOPCNT присутствуют операции подсчета числа бит в векторах. В системе команд ARM Aarch64 присутсвтует инструкция `CNT` (Count non-zero bits), в NEON - `VCNT` . В системе команд Intel GPU так же присуствует инструкция подсчета бит `CBIT` .

Операция *dot product* (скалярное произведение):

$$c = \sum_{i=0}^{n-1} (a_i b_i)^+ - \sum_{i=0}^{n-1} (a_i b_i)^-$$

Требует определения векторных операций над квантизованными данными с использованием побитовых операций AND, XOR и POPCNT.

- [US2024/0054180A1] Fast matrix multiplication for binary and ternary convolutional neural networks on arm central processing unit

Матричное умножение с учетом квантизации запишется:

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} B_{k,j} \approx s_a s_b \sum_{k=0}^{n-1} (\tilde{A}_{i,k} + z_A)(\tilde{B}_{k,j} + z_B) = s_a s_b \tilde{C}_{i,j}$$

$$\tilde{C}_{i,j} = \sum_{k=0}^{n-1}(\tilde{A}_{i,k}\tilde{B}_{k,j}) + z_B \sum_{k=0}^{n-1}\tilde{A}_{i,k} + z_A \sum_{k=0}^{n-1}\tilde{B}_{k,j} + nz_Az_B$$

-- Это выражение определяет алгоритм для тензорных операций. Показано что использование логических операций позволяет выполнять умножение матриц быстрее чем в операциях с плавающей точкой.

- [2310.11453] BitNet: Scaling 1-bit Transformers for Large Language Models
- [2402.17764] The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits
- [2502.11880] Bitnet.cpp: Efficient Edge Inference for Ternary LLMs
- [2504.18415] BitNet v2: Native 4-bit Activations with Hadamard Transformation for 1-bit LLMs
- [2506.07900] MiniCPM4: Ultra-Efficient LLMs on End Devices

Квантизация и ZKP/FHE
Квантизация (такая как MXFP8, MXINT8) снижает вычислительные затраты, но вводит квантовый шум (QSNR), который необходимо учитывать при построении схемы ZKP. Предлагается использовать методы диффузии ошибки (PTQ) из [2310.10537] для компенсации шума.

Для LWE/Ring-LWE квантизация должна быть совместима с модульной арифметикой. Например, `MXFP8` с общей экспонентой (`E4M3`) позволяет эффективно представлять данные в $\mathbb{Z}_q$.

ONNX использует правила сериализации Google Protobufs и позволяет определять тензоры как многомерные массивы данных различных типов:

```
 1: onnx.TensorProto.FLOAT
. . .
16: onnx.TensorProto.BFLOAT16
17: onnx.TensorProto.FLOAT8E4M3FN
18: onnx.TensorProto.FLOAT8E4M3FNUZ
19: onnx.TensorProto.FLOAT8E5M2
20: onnx.TensorProto.FLOAT8E5M2FNUZ
21: onnx.TensorProto.UINT4
22: onnx.TensorProto.INT4
23: onnx.TensorProto.FLOAT4E2M1
24: onnx.TensorProto.FLOAT8E8M0
```

Заметим, что использование идентификаторов типов в других форматах основанных на кодировании protobuf отличается.

Действующая редакция ONNX:Proto позволяет выполнять сериализацию типов FloatN, включая FP8, FP4. Присутствует поддержка *microscaling* MX-форматов с общей экспонентой (E8M0). Формат ONNX охватывает типы данных используемых при обучении такие как AWQ и Post Training Quantization (PTQ) с диффузией ошибки.

Принцип обратного распространения (диффузия) ошибки - ключевой компонент обучения с понижением разрядности. Я бы обратил внимание на возможность использования (или компенсации) квантовой ошибки для построения схемы LWE - шифрования.

В статье [2310.10537, Fig.2] приводится схема (data flow), которая поясняет принцип обучения на квантизованных данных формата MX с общей экспонентой.

- [2405.07135] Post Training Quantization of Large Language Models with Microscaling Formats

Тернарная логика $\{-1, 0, 1\}$ перспективна для ZKP, так как минимизирует размер доказательств. Ожидаем появляения типа предназначенного для эффективного кодирования тензоров тернарного типа FP2E1M0: $\{-1, 0, 1\}$ с общей экспонентой.

- [1605.04711] Ternary Weight Networks

В формате ONNX остуствует представление квантизованных типов данных, широко представленных в GGUF. Однако диапазон форматов может быть расширен при необходимости, принципиальных ограничений нет.

Table 1: Concrete MX-compliant data formats and their parameters.

Name	Block	Scale Fmt	Element Format	Bit-width
MXFP8	32	E8M0	FP8 (E4M3 / E5M2)	8
MXFP6	32	E8M0	FP6 (E2M3 / E3M2)	6
MXFP4	32	E8M0	FP4 (E2M1)	4
MXINT8	32	E8M0	INT8	8

- Общую экспоненту E8M0, обозначают как UE8M0, который принимает значения в интервале [-127,127] или NaN.

Подробнее описание формата ONNX и тензорных операций в графе см. документацию ONNX

- ONNX:Opertors

Каждая тензорная операция содержит флаги типа `differentiable`, `optional` по каждому аргументу. Флаг `differentiable` определяет, можно ли вычислить градиент по аргументу. Аргумент может ссылкаться на список тензорных типов допустимых для данной операции.

Бинарное кодирование Protobuf

Формат определяется как последовательность tag-value пар. value может быть переменной длины с префиксом длины. Поле `tag` и `len-prefix` кодируются оп правилу `VARINT`. Типы данных со знаком кодируются "зигзагом"(ZigZag-encoded), знак при этом размещается в младшем бите.

В документации правила кодирования *wire-format protobuf* не достаточно четко определены. Формат использует little-endian кодировние для много-байтовых полей и полей переменной длины.

```
message    := (tag value)*

tag        := (field << 3) bit-or wire_type;
                encoded as uint32 varint
value      := varint      for wire_type == VARINT,
              i32         for wire_type == I32,
              i64         for wire_type == I64,
              len-prefix  for wire_type == LEN,
              <empty>     for wire_type == SGROUP or EGROUP

varint     := int32 | int64 | uint32 | uint64 | bool | enum | sint32 | sint64;
                encoded as varints (sintN are ZigZag-encoded first)
i32        := sfixed32 | fixed32 | float;
                encoded as 4-byte little-endian;
                memcpy of the equivalent C types (u?int32_t, float)
i64        := sfixed64 | fixed64 | double;
                encoded as 8-byte little-endian;
                memcpy of the equivalent C types (u?int64_t, double)

len-prefix := size (message | string | bytes | packed);
                size encoded as int32 varint
string     := valid UTF-8 string (e.g. ASCII);
                max 2GB of bytes
bytes      := any sequence of 8-bit bytes;
                max 2GB of bytes
packed     := varint* | i32* | i64*,
                consecutive values of the type specified in `.proto`
```

Форматы данных построенные с использованием кодирования wire-format protobuf сопровождаются файлами `.proto` или `.proto3` с описанием структуры данных. Текстовое описание форматов страдает четкостью, поскольку текстовый формат не предусматривает ряд определений, таких как множественность и `CHOICE` - вбор варианта кодирования. Многие ограничения формата невозможно передать средствами текстового языка Protobuf и они даются в комментариях.

Имея опыт работы с другими бинарными форматами данных, я прихожу к необходимости моделирования данных, которое бы учитывало задачи протокола RPC и внедрения схем документов на другом промежуточном языке.

Encoding rules and MIME type for Protocol Buffers

Правила кодирования Protobuf не были стандартизованы как RFC и это осложняет использование в Internet. Рекомендация по использованию protocol buffers в протоколе HTTP могла бы выглядеть так:

MIME Types:
Standard MIME types are defined for Protobuf serializations to ensure proper handling in various contexts, especially in HTTP communication:

- `application/protobuf`:
  This is the primary MIME type for binary-encoded Protobuf data. It implies a binary encoding by default.
- `application/protobuf+json`
  This MIME type is used for Protobuf data encoded in the JSON format (ProtoJSON). It implies a json encoding by default.

Parameters for MIME Types:

- `encoding`:
  This parameter specifies the Protobuf encoding format. It can be binary or json. For application/protobuf+json, encoding defaults to json and cannot be set to binary. Conversely, for application/protobuf, encoding defaults to binary and cannot be set to json.
- `charset`:
  For JSON or Text Format encodings, charset should be set to utf-8. It should not be set for binary encodings. If unspecified, UTF-8 is assumed for JSON/Text formats.
- `version`:
  This parameter is reserved for potential future versioning of Protobuf wire formats and should not be set unless a wire format is officially versioned. Unversioned wire encodings are treated as version 1.

Кодирование тензоров в TensorFlow

Protocol Buffers (Protobuf) can be used to encode tensors, particularly within frameworks like TensorFlow. TensorFlow specifically defines a TensorProto message in its `tensorflow/core/framework/tensor.proto` file to represent tensors in a Protobuf format.

Here's how Protobuf encodes tensors:

TensorProto Structure:
The TensorProto message includes fields to describe the tensor's metadata and its data:

- *dtype*: Specifies the data type of the tensor elements (e.g., DT_FLOAT, DT_INT32).

- *tensor_shape*: A TensorShapeProto message that defines the dimensions and sizes of the tensor.
- *Data Fields*: The actual tensor data is stored in specialized fields based on the dtype. For example, float_val for float data, int_val for integer data, and so on. These are typically repeated fields, allowing for arrays of values.

Protobuf Encoding Mechanism:
When a *TensorProto* is serialized using *Protobuf*, it follows the standard Protobuf encoding rules:

- *Field Numbers and Wire Types*: Each field in TensorProto (like dtype, tensor_shape, float_val) is assigned a field number and encoded with a wire type (e.g., VARINT, LENGTH_DELIMITED).
- *Variable-Length Encoding (Varints)*: Integers, including field numbers and dtype values, are often encoded using varints, which are space-efficient for small numbers.
- *Length-Delimited Encoding*: Fields containing variable-length data, such as the actual tensor values in float_val or int_val, are typically length-delimited, meaning their length is encoded before the actual data.

Кодирование графа в ONNX

- ONNX:IR Open Neural Network Exchange Intermediate Representation (ONNX IR) Specification

Кодирование типизованных массивов в CBOR

CBOR (Concise Binary Object Representation) typed arrays refer to a specific extension of the CBOR data format that allows for the efficient and unambiguous representation of arrays containing numeric data of a specific type (e.g., arrays of 16-bit unsigned integers, or 32-bit floating-point numbers). This is achieved through the use of CBOR tags.
RFC 8746 определяет типизованные массиыв CBOR и набор тэгов для стандартных типов данных (tags 64 to 87)

- [RFC 8259] [STD90]: The JavaScript Object Notation (JSON) Data Interchange Format
- [RFC 8746] Concise Binary Object Representation (CBOR) Tags for Typed Arrays
- [RFC 8949] [STD94]: Concise Binary Object Representation (CBOR)
- [IANA:CBOR-Tags] Concise Binary Object Representation (CBOR) Tags

Кодирование выглядит следующим образом: тэг типизованного массива - размерность, тег прикладного типа, и массив данных, который может быть упакован в пригодном для запуска модели виде.

```
<Tag 40> # multi-dimensional array tag
   82       # array(2)
     82      # array(2)
       02     # unsigned(2) 1st Dimension
       03     # unsigned(3) 2nd Dimension
     <Tag 65> # uint16 array
       4c     # byte string(12)
         0002 # unsigned(2)
         0004 # unsigned(4)
         . . .
```

Кодирование в таком виде допускает применение application-specific tags для определения типа элемента данных массива. Списки параметров различного типа в CBOR обозначаются словом `array` , а объекты (с именами полей, как в JSON) обозначаются словом `map` . В примере шапка это список из двух элементов - размерности и тег типа массива. Сам массив кодируется как байтовая строка. Tag - это превикс определяющий тип кодирования.

Protobuf vs CBOR

{Дополнить сравнением производительности и размера сериализованных данных для Protobuf и CBOR в контексте RPC. CBOR может быть более компактным за счет тегов и бинарного формата при описание структурированных данных. CBOR лучше стандартизирован для криптографических приложений и для использования в протоколах промышленных IoT устройств. Кодирование CBOR следует рассматривать как неотъемлемую часть протокола CoAP. Protobuf - это только метод бинарного кодирования, используется совместно с текстовым описанием структуры сообщений. Разбор структуры пакета данных без схемы не представляется возможным.}

Критерии сравнения

- Zero-Copy: возможность использования тензорных данных без копирования буферов
- компактность и детерминированное представление метаданных
- типизация и разбор структурированных данных без схемы данных.

Protobuf:

- Плюсы: Поддержка в gRPC, широкая экосистема, возможность описания схемы через `.proto` .
- Минусы: Более громоздкий формат, отсутствие встроенной поддержки типизованных массивов, неканоническое кодирование.

CBOR:

- Плюсы: Компактность, поддержка типизованных массивов (RFC 8746), стандартизация для IoT (CoAP).
- Минусы: Меньшая поддержка в ML-экосистеме, необходимость дополнительной обработки для сложных графов.

Рекомендация: Использовать CBOR для Edge-устройств и IoT, Protobuf — для серверных приложений с gRPC. Для ZKP/FHE предпочтительнее CBOR из-за компактности и стандартизированных тегов.

Циклотомические полиномы

- Cyclotomic polynomial

Криптосистемы на циклотомических полиномах — это тип симметричных криптосистем, где для шифрования и расшифрования используется один и тот же ключ. Эти системы опираются на свойства кольца многочленов над конечным полем, которые могут быть связаны с циклотомическими полиномами, которые невозможно разложить на множители (неприводимые) в определенном поле.

Циклотомическое поле степени n — это расширение поля рациональных чисел $\mathbb{Q}$, полученное присоединением примитивного корня n-й степени из единицы, обозначаемого ζ_n, где $\zeta_n = e^{2\pi i/n}$ и $\zeta_n^n = 1$. Поле обозначается как $\mathbb{Q}(\zeta_n)$.

Циклотомическое кольцо — это кольцо целых чисел циклического поля, то есть $\mathbb{Z}[\zeta_n]$, которое состоит из всех целочисленных комбинаций степеней ζ_n. Если n является степенью двойки, то есть $n = 2^k$, где $k \geq 1$, то циклическое поле и кольцо имеют особые свойства, которые мы рассмотрим ниже.

Циклотомические полиномы

Циклотомический полином (cyclotomic polynomial) степени n, обозначаемый $\Phi_n(x)$, — это минимальный полином примитивного корня n-й степени из единицы над $\mathbb{Q}$. Он определяется как:

$$\Phi_n(x) = \prod_{\substack{1 \leq a \leq n \\ \gcd(a,n)=1}} (x - \zeta_n^a) \,,$$

где произведение берется по всем a, взаимно простым с n.

Для степеней двойки циклические полиномы имеют явный вид. Например:

- Для $n = 2$: $\Phi_2(x) = x + 1$.
- Для $n = 4$: $\Phi_4(x) = x^2 + 1$.
- Для $n = 8$: $\Phi_8(x) = x^4 + 1$.
- Для $n = 2^k$: $\Phi_{2^k}(x) = x^{2^{k-1}} + 1$.

Эти полиномы неприводимы над $\mathbb{Q}$, и циклическое поле $\mathbb{Q}(\zeta_{2^k})$ изоморфно $\mathbb{Q}[x]/\langle \Phi_{2^k}(x) \rangle$.

В постквантовой криптографии (PQC) на основе решеток (lattice-based cryptography) используются полиномиальные кольца $\mathcal{R}_q = \mathbb{Z}_q[x]/f(x)$ с циклическими полиномами $f(x)$. Традиционно применяются полиномы $x^n + 1$ (где $n = 2^k$, минимальная ошибка расшифровки, но ограниченный выбор степени) и $x^n + \cdots + x + 1$ (где $n + 1$ — простое, гибкий выбор степени, но большая ошибка расшифровки). В общем виде могут так же использоваться циклотомические полиномы вида $x^n - x^{n/2} + 1$.

LWE (Learning With Errors) и LWR (Learning With Rounding) — это две фундаментальные проблемы в криптографии на основе решеток, которые лежат в основе многих пост-квантовых криптосистем. Они используются для построения безопасных схем шифрования, подписи и других криптографических примитивов. Рассмотрим их различия, преимущества и недостатки.

LWE (Learning With Errors) — это проблема, основанная на добавлении случайного шума к линейным уравнениям. Для заданного модуля q, матрицы $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, секретного вектора $\mathbf{s} \in \mathbb{Z}_q^n$ и вектора шума $\mathbf{e}$, распределенного по некоторому закону (обычно гауссовому распределению), LWE-задача заключается в восстановлении $\mathbf{s}$ по набору пар:

$$(\mathbf{A}, \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \mod q) .$$

Ключевые характеристики схемы:Шум. Шум $\mathbf{e}$ добавляется к $\mathbf{A}\mathbf{s}$, что делает задачу вычислительно сложной.

LWR (Learning With Rounding) — это модификация LWE, где вместо добавления случайного шума используется округление значений. Для матрицы $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, секретного вектора $\mathbf{s} \in \mathbb{Z}_q^n$, и модуля q, LWR-задача заключается в восстановлении $\mathbf{s}$ по парам:

$$(\mathbf{A}, \mathbf{b} = \lfloor \mathbf{A}\mathbf{s} \rceil_p) ,$$

где $\lfloor \cdot \rceil_p$ обозначает округление по модулю $p \leq q$, обычно p значительно меньше q. Округление означает, что вместо $\mathbf{A}\mathbf{s} \mod q$ берутся только старшие биты результата, масштабированные к модулю p.
Ключевые характеристики: Округление используется вместо шума.

- [2013/098] Learning with Rounding, Revisited. New Reduction, Properties and Applications

Вместо добавления случайного шума LWR использует детерминированное округление, что снижает вычислительные затраты. Схему LWR возможно интегрировать в схему ZKP, и использовать совместно с PTQ - пост-квантизацией при запуске нейросетей.

Приложение A. Алгоритмы Ring-LWE для ZKP

Принятые скращения:

```
**PKC** -- сокр. от Public-Key Cryptography
**PQC** -- сокр. от Post-Quantum Cryptography
**HE**  -- сокр. от Homomorphic Encryption
**FHE** -- сокр. от Fully Homomorphic Encryption
**FFT** -- сокр. от Fast Fourier Transform
**NTT** -- сокр. от Number-Theoretic Transform
**RNS** -- сокр. от Ring Number System
**MPC** -- сокр. от Multi-Party Computation
**BGV** -- сокр. от авторов Brakerski-Gentry-Vaikuntanathan
**BFV** -- сокр. от авторов Brakerski-Fan-Vercauteren
**CKKS** -- сокр. от авторов Cheon-Kim-Kim-Song
**AIR**  -- Arithmetic Intermediate Representation
**IOP**  -- Interactive Oracle Proof
**FRI**  -- Fast Reed-Solomon IOP
**PCS**  -- Polynomial Commitment Scheme
**R1CS** -- Rank-1 Constraint System
**QAP**  -- Quadratic Arithmetic Program
**LWE**  -- сокр. от Learning with Errors
**LWR**  -- сокр. от Learning with Rounding
**GLWR** -- сокр. от Generalized Learning with Rounding
**R-LWE**-- сокр. от Ring Learning with Errors
**ZKP**  -- сокр. от Zero-Knowledge Proof
**NIZK** -- Non-Interactive Zero-Knowledge
**zk-SNARK** -- Zero-Knowledge Succinct Non-Interactive Argument of Knowledge
**zk-STARK** -- Zero-Knowledge Succinct Transparent Argument of Knowledge
```

Современные реализации гомоморфного шифрования (HE) в значительной степени зависят от арифметики полиномов в конечном поле. Это особенно актуально для схем HE, таких как BGV, BFV и CKKS. Двумя основными узкими местами в производительности примитивов и приложений HE являются полиномиальное модульное умножение и прямое и обратное теоретико-числовое преобразование (NTT, сокр. от Number-Theoretic Transform).

- [2103.16400] Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52

В статье даны математические принципы и алгоритмы, реализованные в библиотеке:

- (https://github.com/IntelLabs/hexl) Intel Homomorphic Encryption (HE) Acceleration Library

Основные оптимизации в арифметике на кольце полиномов:

- NTT for fast multiplication: The number-theoretic transform (described below) is used to convert polynomials into a point-value representation, reducing the complexity of polynomial multiplication from $O(N^2)$ to $O(N \log N)$.
- Barrett or Montgomery reduction: These techniques optimize modular arithmetic by avoiding expensive division operations.
- Residue Number System (RNS): For large q, RNS decomposes operations into smaller, parallelizable computations over coprime moduli.

Коэффициенты полинома представляются, как элементы вектора. Длина вектора ограничена N элементами.

Полиномиальная арифметика $\mathcal{R}_q = \mathbb{Z}_q / \langle x^N + 1 \rangle$:

- сложение $c_i = a_i + b_i \pmod q$
- обратный элемент по сложению - вычитание $c_i = q - a_i \pmod q$
- поэлементное умножение для данных $a, b \in \mathcal{R}_q$, расчитать $c = a \odot b : c_i = a_i \cdot b_i \pmod q$
- произведение вектора на скаляр $c_i = a_i \cdot b \pmod q$
- произведение векторов $a, b \in \mathcal{R}_q$, с редуцированием по полиному $(x^N + 1)$

$$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j} - \sum_{j=i+1}^{N-1} a_j b_{N+i-j} \pmod q$$

где второе слагаемое получается при редуцировании полинома по правилу

$$x^N \equiv -1 \mod (x^N + 1)$$

- ротация полинома $f(x) \cdot x^{-h}$

Умножение полиномов следует рассматривать как композицию двух элементарных операций: сложение векторов коэффициентов полинома и операцию циклического сдвига (polynomial rotation).

$$f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \in \mathcal{R}_q$$

$$f(x) \cdot x^{-h} = a_h + a_{h+1}x + ... + a_{n-1}x^{n-1-h} - a_0x^{n-h} ... - a_{h-1}x^{n-1}$$

Операция редуцирования по модулю. Для 64 битных машин, x86_64 в частности, действует операция $(a \cdot b) \mod q$

```
uint64_t y =  ( uint128_t (a) * b ) % q;
```

Для длинных чисел можно использовать редукцию Баррета для модульной арифметики и идеи отложенного редуцирования для операций сложения.

$$x \mod q = x - \lfloor x/q \rfloor q(\mod q)$$

Редукция Баретта использует тот факт, что выражение истинно для вычисления $\lfloor x/q \rfloor$ с произвольной точностью.
В алгоритме используется обратная величина по отношению к делению. Вместо деления выполняется умножение на константу Баррета для данного q и сдвиг.

Algorithm 1 Barrett Reduction
Require: $q < 2^Q$, $d < 2^D$, $k = \lfloor \frac{2L}{q} \rfloor$, with $Q \leq D \leq L$
Ensure: Returns $d \mod q$

```
1: function Barrett Reduction(d, q, k, Q, L)
2:   c1 ← d ≫ (Q − 1)
3:   c2 ← c1k
4:   c3 ← c2 ≫ (L − Q + 1)
5:   c4 ← d − qc3
6:   if c4 ≥ q then
7:      c4 ← c4 − q
8:   end if
9: return c4
```

При переходе к векторной реализации операцию `if` $c4 \geq q$ `then` $c4 \leftarrow c4 - q$ можно заменить на `min(x-q, x)` над целыми числами без знака.

В библиотеке Intel HEXL для реализации операций с полиномами используются простые числа $q < 2^{64}$ и векторные операции AVX-512 Integer Fused Multiply Add (IFMA) – fused multiply add of integers using 52-bit precision.
Система команд AVX512IFMA52 добавлена в акхитектуре x86-64 в процессоры Intel начиная с Xeon Scalable Gen 3. Показано, что использование инструкций IFMA52 ускоряет работу алгоритма умножения полиномов в 3 раза в сравнении с AVX512-DQ [HEXL].

В библиотеке верхнего уровня такой, как OpenFHE, используется операции умножения полиномов с использованием аналога фурье преобразования. Фурье преобразование имеет преимущество на больших N и дает $O(N \log N)$ сложность вычислений, в сравнении с $O(N^2)$ для операций умножения. Метод NTT (Number Theoretic Transform) работает аналогичным образом как FFT. NTT пребставляет собой обратимое линейное преобразование, в пространство где операция умножения действует аналогично операции сложения - поэлемнтно.

Библиотека Intel HEXL предлагает векторную реализацию алгоритма NTT для умножения полиномов $\mathbb{Z}_q[x]/\langle x^N + 1\rangle$.

Модульное умножение полиномов

Модульное умножение полиномов включает умножение двух полиномов в полиномиальном кольце (например, $\mathbb{Z}_q[X]/\langle X^N + 1\rangle$) с последующим приведением результата по модулю простого числа q и полинома (например, $X^N + 1$). Эта операция является центральной в схемах HE, поскольку шифротексты представляются в виде полиномов, а операции, такие как шифрование, расшифровка строятся на умножении и сложении полиномов.

Полиномы в схемах HE имеют большие степени (например, $N = 2^{10}$ до 2^{15}), что делает умножение вычислительно затратным.

Прямое и обратное теоретико-числовое преобразование (NTT)

NTT — это специализированная форма быстрого преобразования Фурье (FFT), адаптированная для конечных полей, используется для умножения полиномов в конечном поле. Оно преобразует полином из представления по коэффициентам в представление по точкам (прямое NTT) и обратно (обратное NTT). Это полезно, поскольку умножение полиномов в точечной области является поэлементным (и выполняется гораздо быстрее, чем свёртка по коэффициентам).

Let ω be a primitive N'th root of unity in $\mathbb{Z}_q$

$$\mathrm{NTT}_\omega^N(a)_i = \sum_{j=0}^{N-1} a_j \omega^{ij} \pmod q\,, \quad \mathrm{InvNTT}_\omega^N(\tilde{c})_i = \frac{1}{N}\sum_{j=0}^{N-1} \tilde{c}_j \omega^{-ij} \pmod q$$

The NTT can be used to speed up polynomial-polynomial multiplication in $\mathcal{R}_q$. However, using $\odot$ to indicate element-wise multiplication, the straightforward usage $InvNTT(NTT(a) \odot NTT(b))$ corresponds to polynomial-polynomial multiplication in $\mathbb{Z}_q^N/\langle X^N - 1\rangle$, в то время как HE работает в кольце $\mathcal{R}_q = \mathbb{Z}_q^N/\langle X^N + 1\rangle$. В этом случае умножение принимает вид

$$\tilde{c} = InvNTT(NTT(\tilde{a}) \odot NTT(\tilde{b}))$$

где элементы векторов $\tilde{a}, \tilde{b}, \tilde{c}$ поэлементно домножаются на вектор корней степени 2N из единицы, $\tilde{a} = \{a_0, a_1\psi, a_2\psi^2, ...a_{N-1}\psi^{N-1}\}$, $\psi^2 = \omega$ - примитивный корень полинома 2N-й степени.

$$c = \{\tilde{c}_0, \tilde{c}_1\psi^{-1}, \tilde{c}_2\psi^{-2}, ...\tilde{c}_{N-1}\psi^{-N+1}\}$$

В таком виде число операций порядка $O(N^2)$, но может сокращаться до $O(N \log N)$ за счет рекурсивного разложения полиномов $f(x) = f_0(x^2) + x \cdot f_1(x^2)$, где $f_0(x), f_1(x)$ - полиномы степени $N/2$. Эта идея лежит в основе алгоритма Cooley-Tukey NTT.

$$\begin{aligned} f(x) &= a_0 + a_1x^1 + a_2x^2 + ... + a_{N-1}x^{N-1} \\ &= f_0(x^2) + x \cdot f_1(x^2) \\ f_0(x) &= a_0 + a_2x^1 + a_4x^2 + ... + a_{N-2}x^{N/2-1} \\ f_1(x) &= a_1 + a_3x^1 + a_5x^2 + ... + a_{N-1}x^{N/2-1} \end{aligned}$$

Имеется свойство перидичности и симметрии корней степени 2N из единицы $\psi^{2N+k} = \psi^k$ и $\psi^{N+k} = -\psi^k \mod q$, что позволяет сократить количество вычислений. Используя свойства $\omega^N = 1$ и $\omega^{N/2} = -1 \mod q$ получаем выражение:

$$\begin{aligned} c_i &= f_0(\omega^{2i}) + \omega^i f_1(\omega^{2i}) \\ c_{i+N/2} &= f_0(\omega^{2i}) - \omega^i f_1(\omega^{2i}) \end{aligned}$$

Это выражение, его визуальное представление, называется "бабочка" (CT NTT butterfly). Бабочка может применяться рекурсивно с понижением степени N/2 на каждой итерации. Линейная запись рекурсивного алгоритма называется Cooley-Tukey NTT.

- [2509.05884] Introduction to Number Theoretic Transform
- [2024/585] A Complete Beginner Guide to the Number Theoretic Transform (NTT)

-- Описан вывод "бабочек" и выражений для прямого и обратного преобразования NTT для случая циклических полиномов $(x^N - 1)$ и негациклических полиномов $(x^N + 1)$. Для негациклических полиномов выражение для обратного преобразования дается через корнеь 2N-й степени из единицы $(\psi^2 = \omega)$:

$$\mathrm{NTT}_{\psi}^{N}(x)_i = \sum_{j=0}^{N-1} x_j \psi^{(2i+1)j} \pmod{q},$$
$$\mathrm{invNTT}_{\psi}^{N}(x)_i = \sum_{j=0}^{N-1} x_j \psi^{-(2i+1)j} \pmod{q}.$$

Эти выражения мы используем в качестве референсного алгоритма, для отладки работы параллельной версии и версий с "бабочками". Используя свойства периодичности и симметрии корней, получаем выражения для CT-"бабочек" прямого преобразования:

$$c_i = A_i + \psi^{2i+1} B_i \pmod{q}$$
$$c_{i+N/2} = A_i - \psi^{2i+1} B_i \pmod{q}$$

и для GS-"бабочек" обратного преобразования:

$$a_{2i} = (A_i + B_i)\psi^{-2i} \pmod{q}$$
$$a_{2i+1} = (A_i - B_i)\psi^{-2i} \pmod{q}$$

Возможность применения быстрых алгоритмов NTT обусловлено требованием существования корней степени N из единицы для данного модуля. Модули могут определяться подходящими для NTT.

- NWC-NTT friendly - простые числа q для которых существует корень степени 2N из единицы $q \equiv 1 \mod 2N$.
- PWC-NTT friendly - простые числа q для которых существует корень степени N из единицы $q \equiv 1 \mod N$.

Для композитных полей, $q = q_1^{m_1} \cdot q_2^{m_2} \cdot ... \cdot q_k^{m_k}$, существует корень степени N из единицы, если для всех простых q_i существует корень степени N из единицы $q_i \equiv 1 \mod N$.

Заметим, при использовании матричной записи умножения полиномов, возникают матрицы с циклической структурой -- циркулянты и негациклические матрицы, что позволяет использовать алгоритмы NTT для быстрого умножения вектора на матрицу. Можно говорить об изоморфизме между полиномами и матрицами - циркулянтами. Это свойство используется при разработке криптографических алгоритмов, хеш-функций и функций шифрования, на кольце полиномов.

Циклическая свёртка (Positive Wrap Convolution, PWC) {дать определение}

{связь с матрицами циркулянтами}

Негациклическая свёртка (Negacyclic Wrap Convolution, NWC) — это операция, связанная с умножением полиномов в кольце $\mathbb{Z}[x]/(x^N+1)$, где результат умножения двух полиномов берётся по модулю x^N+1. Формально, для двух полиномов $a(x) = a_0 + a_1 x + \cdots + a_{N-1}x^{N-1}$ и $b(x) = b_0 + b_1 x + \cdots + b_{N-1}x^{N-1}$ с коэффициентами из $\mathbb{Z}$, их негациклическая свёртка определяется как коэффициенты полинома $c(x) = a(x)b(x) \mod (x^N+1)$.

Коэффициенты результирующего полинома $c(x) = c_0 + c_1 x + \cdots + c_{N-1}x^{N-1}$ вычисляются следующим образом:

$$c_k = \sum_{i+j=k \mod N} a_i b_j - \sum_{i+j=k+N \mod N} a_i b_j,$$

где второе слагаемое учитывает отрицательный знак, возникающий при редуцировании $x^N \equiv -1 \mod (x^N+1)$.

Умножение полиномов методом KEM

- [ML-KEM] Module-Lattice-Based Key-Encapsulation Mechanism Standard, 2024 (https://doi.org/10.6028/NIST.FIPS.203)

-- в стандарте предложен метод умножения полиномов с использованием NTT, который использует корни N степени из единицы. На выходе алгоритма образуются сопряженные пары чисел. Умножение после NTT выполняется не поэлементно, а с использованием пары элементов. В стандарте используется простое число $q = 13 \cdot 2^8 + 1$ для которого существует корень степени $N = 256$ из единицы $q \equiv 1 \mod N$ и не существует корней степени $2N$. Метод умножения [ML-KEM, Alg. 12] можно связать с методом умножения Карацубы, можно назвать умножением в поле комплексных чисел со странной мнимой единицей, или назвать "бабочкой" специального вида.

Представим разложение по четным и нечетным коэффициентам полиномов, назвем его "бабочкой" Карацубы:

$$\begin{aligned} f(x) &= f_0(x^2) + x \cdot f_1(x^2) \\ g(x) &= g_0(x^2) + x \cdot g_1(x^2) \\ h(x) &= f(x)g(x) = h_0(x^2) + x \cdot h_1(x^2) \\ &= f_0(x^2)g_0(x^2) + x(f_0(x^2)g_1(x^2) + f_1(x^2)g_0(x^2)) + x^2 f_1(x^2)g_1(x^2) \end{aligned}$$

Вот из этого и получается

$$\begin{aligned} h_0(x) &= f_0(x^2)g_0(x^2) + x^2 f_1(x^2)g_1(x^2) \mod (x^2 - \zeta^{2i+1}) \\ h_1(x) &= f_0(x^2)g_1(x^2) + f_1(x^2)g_0(x^2) \end{aligned}$$

А для полинома $x^N + 1$ получается разложение вида $\prod(x^2 - \zeta^{2i+1})$, откуда можно получить правила умножения в парах, которое применяется в ML-KEM:

$$\begin{aligned} \mathrm{c}_0 &= a_0 b_0 + \zeta^{2i+1} a_1 b_1 \\ \mathrm{c}_1 &= a_0 b_1 + a_1 b_0 \end{aligned}$$

Умножение методом Карацубы (Karatsuba)
Другой метод понижения сложности вычисления произведения полиномов - рекурсивный алгоритм умножения Анатолия Карацубы.

$$\begin{aligned} (f \circ g)(x) &= (f_0(x) + x^{N/2} \cdot f_1(x))(g_0(x) + x^{N/2} \cdot g_1(x)) \pmod{x^N \pm 1} \\ &= f_0(x)g_0(x) + x^{N/2}(f_0(x)g_1(x) + f_1(x)g_0(x)) \mp f_1(x)g_1(x) \end{aligned}$$

Реализация будет содержать метод сложения с ротацией одного из полиномов. Метод умножения Карацубы может использоваться совместно с NTT и в тех случаях, когда простое число не имеет корня степени N.

Аппаратное ускорение:
Специализированное оборудование (например, GPU, FPGA или ASIC) может ускорить NTT за счёт параллелизации операций "бабочка" (Harvey NTT butterfly) или оптимизации модульной арифметики.

- [1205.2926] David Harvey. Faster arithmetic for number-theoretic transforms

{Обсуждение} Я вижу возможность ускорения операции NTT за счет отложенного редуцирования при суммировании и транспонировании 16x16 при редуцировании группы векторных операций.

Поиск примитивного корня k-й степени по модулю простого числа

Алгоритм поиска примитивного корня k-й степени по модулю простого числа q заключается в нахождении элемента ω - в мультипликативной группе $\mathbb{Z}/p\mathbb{Z}$, такого что порядок ω равен k, где k — делитель $p - 1$.

Шаги алгоритма:

1. Проверка исходнях данных:

- Убедитесь, что q — простое число.
- Проверьте, что k делит $(q - 1)$. Если нет, примитивного корня k-й степени не существует.

2. Поиск примитивного корня:

Примитивный корень k-й степени — это элемент $\omega \in \{2, 3 \ldots, q-1\}$, для которого порядок равен k, то есть $\omega^k \equiv 1 \pmod{q}$, и k — наименьшая степень, удовлетворяющая этому условию.

Для этого:

1. Выполнить перебор элементов $\omega \in \{2, 3 \ldots, q-1\}$.
2. Для каждого ω проверьте, является ли $\omega^{(q-1)/q_i} \not\equiv 1 \pmod{q}$ для всех простых делителей q_i из факторизации $q-1$, если q_i не входит в факторизацию k. Это гарантирует, что порядок ω не меньше k.
3. Проверьте, что $\omega^k \equiv 1 \pmod{q}$. Если это выполнено, и предыдущее условие для всех q_i также выполнено, то ω — примитивный корень k-й степени.

Практически, для использования NTT, нам нужно найти примитивный корень 2^s-й степени по модулю простого числа вида $q = a_0 \cdot 2^s + 1$. При выборе простого числа q можно потребовать, чтобы a_0 - было простым числом. При выборе элемента ω на первом шаге следует проверять $\omega^{(q-1)/2} \not\equiv 1 \pmod{q}$.

- [2012/470] Some Connections Between Primitive Roots and Quadratic Non-Residues Modulo a Prime

-- в статье предложен способ поиска корней степени k по модулю простого числа q. Метод основан на подборе и анализе делителей (q-1) из числа квадратичных не-вычетов. g выбирается как квадратичный не-вычет, используя символ Якоби $jacobi(g, q) = -1$.

For an odd prime p and an integer a not divisible by p, Euler's criterion states that a is a quadratic residue modulo p if and only if $a^{(p-1)/2} \equiv 1 \pmod{p}$, and a is a quadratic non-residue modulo p if and only if $a^{(p-1)/2} \equiv -1 \pmod{p}$. This criterion provides a way to determine if a number has a square root modulo a prime, and it is often expressed using the Legendre-Jacobi symbol: $\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}$.

Для случая 2^s-й степени, и модулей вида $q = a_0 \cdot 2^s + 1, s \geq 1$, где a_0 - нечетное простое число, предложен простой метод:

1. Найтия генератор g мультиплиткативной группы, являющийся квадратичным не-вычетом по модулю q.
2. Если $\omega^{2^s} \neq 1$ то ω - примитивный корень 2^s-й степени.

Алгоритм поиска примитивного корня модуля q:

Require: g - квадратичный не-вычет по модулю $q = a_0 \cdot 2^s + 1$, $s \geq 1$, где a_0 - нечетное простое число
Ensure: α - примитивный корень 2^s-й степени по модулю q

1. if $g^{2^s} \neq 1 \pmod q$ return $\alpha \leftarrow g$.
2. $\gamma \leftarrow 2$
3. while $(\gamma^2)^{2^s} = 1 \pmod q$ do $\gamma \leftarrow \gamma + 1$
4. return $\alpha \leftarrow (g\gamma^2)$

Алгоритм поиска корня k-ой степени по модулю q

Require: g - квадратичный не-вычет по модулю $q = a_0 \cdot 2^s + 1$, $s \geq 1$, где a_0 - нечетное простое число
Ensure: γ - примитивный корень 2^s-й степени по модулю q

1. $\gamma \leftarrow 2$
2. while $(\gamma^2)^k \neq 1 \pmod q$ or $(\gamma)^k = 1 \pmod q$ do $\gamma \leftarrow \gamma + 1$
3. return γ

Заметитм в пункте (2) $(\gamma)^{2^s} = \pm 1 \pmod q$ - принимает одно из двух значений. Поиск значения γ выполняется последовательным перебором γ начиная с 2 или методом умножения $\gamma = \gamma \cdot g$ начиная с g.

Генератр мультипиликативной группы g выбирается, как квадратичный не-вычет по модулю q используется в качестве начального значения поиска.
Значения γ, и $\omega = \gamma^2 \pmod q$, $\omega^{2^s} \equiv 1 \pmod q$, а также их обратные значения γ^{-1} и ω^{-1} используются в алгоритме NTT.
Обратные значения находятся по малой теореме Ферма, путем возведения в степень: $\gamma^{-1} = \gamma^{q-2} \pmod q$.

Algorithm A.1 Barrett Reduction

Require: $q < 2^Q, d < 2^D$, $k = \lfloor 2^L/q \rfloor$, with $Q \leq D \leq L$
Ensure: Returns $d \mod q$

1. function Barrett Reduction(d, q, k, Q, L)
2. $c_1 \leftarrow d \gg (Q-1)$
3. $c_2 \leftarrow c_1 k$
4. $c_3 \leftarrow c_2 \gg (L-Q+1)$
5. $c_4 \leftarrow d - qc_3$
6. if $c_4 \geq q$ then
7. $\quad c_4 \leftarrow c_4 - q$
8. end if
9. return c_4
10. end function

{обсуждение} Что такое не полное редуцирование (lazy reduction). Мы берем переполнение, старшую часть числа $z = z_1 2^{32} + z_0$, умножаем по модулю на специальную константу и вычитаем из числа. Получаем остаток в младшей части числа z_0. По сути мы выполняем операцию деления с остатком. В операции не учитывается значение младшей части числа z_0. В результате остаток может превышать q, выполняется компенсация в следующем шаге (6)-(7). При работе в цикле, когда на каждом цикле выполняется суммирование остатков, компенсацию можно исключить и выполнять одну операцию по модулю q за пределами цикла. Бывают случаи когда компенсацию можно исключить при последовательном выполнении операций редуцирования, в том случае когда операция редуцирования допускает на входе значения больше $q \cdot \beta$.

Помимо редцирования Барретта, существует редуцирование Монтгомери. Если в первом случае мы выполняли преобразование над старшей частью числа, в алгоритме Монтгомери мы выполняем преобразование над младшей частью числа.

Montgomery's reduction

Определение. Пусть $R > q$ и $\mathrm{GCD}(R, q) = 1$. Метод вычисления $zR^{-1} \mod q$ для входных значений $z < qR$ называется редукцией Монтгомери. Если $q' = -q^{-1} \mod R$, то $c = zR^{-1} \mod q$ может быть получен:

1. $c \leftarrow (z + (zq' \mod R)q)/R$
2. $c \leftarrow c - q$ if $c \geq q$

Метод Монтгомери эффективен для работы в цикле, например при возведении в степень или умножении полиномов.
Преобразование из $\tilde{x} = xR \mod q$ в $x = xR^{-1} \mod q$ выполняется один раз при выходе из цикла. В вычислениях используется обратная величина $q^{-1} \mod R$, которая вычисляется один раз предварительно. Для простоты надо понимать что R выбирается исходя из разложения удобного для данной вычислительной платфрмы, например $R = 2^{32}$.

Algorithm A.2 Signed Montgomery Reduction

Require: $0 < q < 2^{31}$, $-2^{31}q \leq z = z_1\beta + z_0 < 2^{31}q$$ where $0 \leq z_0 < 2^{32}$
Require: precomputed $q' = q^{-1} \mod^{\pm} \beta$
Ensure: Returns $r \equiv \beta^{-1}z \mod q$; $-q < r < q$

1. $m \leftarrow z_0 q' \mod^{\pm} \beta$ -- signed low product
2. $t_1 \leftarrow \lfloor (mq)/\beta \rfloor$ -- signed high product
3. $r \leftarrow z_1 - t_1$

Замечание: редуцирование Монтгомери со знаком используется совместно с умножением со знаком и это приводит к изменению в алгоритме CT- и GS-batterfly. Коррекция при этом считается отложенной, что позволяет уменьшить количество операций при выполнении модульного умножения до трех процессорных инструкций.

1. Signed Multiply Long, multiplies two 32-bit signed values to produce a 64-bit result.
2. MUL. Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.
3. Signed Multiply Accumulate Long, multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

- [2018/039] Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography

-- Предложено редуцирование со знаком для плаформы AVX2: signed Montgomery и signed Barrett.

- [2020/1278] Compact Dilithium Implementations on Cortex-M3 and Cortex-M4
- [2023/1955] Barrett Multiplication for Dilithium on Embedded Devices
- [2404.13544] Faster Post-Quantum TLS 1.3 Based on ML-KEM: Implementation and Assessment -- интересная работа в каждом алгоритме намеренно внесена ошибка.

Алгоритм редуцирования Монтгомери оптимизируется до трех инструкций AVX512 на x86-64 (см. реализацию qnn_hexl.c и ml_kem.c).

Algorithm A.1.1 Signed Barrett modular reduction

Require: $0 < q < \beta/2$, $-\beta q \leq z = z_1\beta + z_0 < 2^{31}q$ where $-2^{31} \leq z_0 < 2^{31}$
Require: precomputed $u = \lfloor (2^{\lfloor \log(q) \rfloor - 1}\beta)/q \rceil$
Ensure: Returns $r \equiv a \bmod^{\pm} q$; $-q < r \leqslant q$

1. $t \leftarrow \lfloor au/\beta \rfloor \gg 2^{\lfloor \log(q) \rfloor - 1}$ -- signed low product and arithmetic shift
2. $t \leftarrow t \cdot q \bmod^{\pm} \beta$ -- signed low product
3. $r \leftarrow z - t$ -- signed remainder

Algorithm A.3 Modular exponentiation

Require: $0 \leq z < q$
Ensure: Returns $r \equiv z^e \mod q$

1. $r \leftarrow 1$
2. while $e \neq 0$ do
3. $\quad z \leftarrow z^2 \mod q$
4. $\quad r \leftarrow rz \mod q$ if $e \bmod 2 \neq 0$
5. $\quad e \leftarrow \lfloor e/2 \rfloor$
6. end while

{обсуждение} Алгоритмов возведения всепень несколько вариантов. Подробнее я бы отослал к учебнику Guide to Elliptic Curve Cryptography [GECC]. Хотелось бы выделить несколько

вариантов, которые используются в криптографии. В примере дан алгоритм бинарный, от младшего к старшему разряду. Можно предложить вариант алгоритма от старшего к младшему разряду. Алгоритм может быть построен на редуцировании Барретта или Монтгомери.

В нашем теме основное применение алгоритма возведения нахождение констант для прямого и обратного NTT.

Алгоритмы возведения в степень надо рассматривать как шаблон метода. Медод одинаково подходит для умножения и для возведения в степень в качестве параметра используются два метода: удвоение (dubling) и сложение (addition). В случае возведения в степень, удвоение - возведение в квадрат, сложение - умножение. Алгоритм одинаково выглядит для разных математических классов, включая кольца полиномов и эллиптические кривые.

Возведение в степень используется для нахождения обратного числа $a^{-1} \equiv a^{p-2} \mod p$. Однако существуют алгоритмы нахождения обратного числа, основнные на принципе расширенного алгоритма Евклида.

Algorithm A.4 Shoup's modular multiplication

Require: $b \in [0, \beta); w \in [1, p), p < \beta/2$
Require: $\bar{w} \leftarrow \lfloor (w \cdot \beta)/p \rfloor$, $\beta \leftarrow 2^{32}$ or 2^{64}
Ensure: Returns $r = b \cdot w \mod p$

1. $q \leftarrow \lfloor (b \cdot \bar{w})/\beta \rfloor$ -- (quotient)
2. $r \leftarrow b \cdot w - q \cdot p \pmod \beta$ -- (remainder)
3. $r \leftarrow r - p$ if $r \geq p$

Оптимизированные алгоритмы умножения в кольце полиномов:
Техники, такие как алгоритмы Кули-Тьюки (Cooley-Tukey NTT) и Джентльмена-Санда (Gentleman-Sande (GS) InvNTT), а также выбор параметров (например, N как степень 2 и q, поддерживающее эффективное NTT), улучшают производительность. Алгоритмы используют "бабочки" (Harvey butterfly) с редукцией Баррета или редукцией Монтгомери для ускорения вычислений.

Бабочкой Харви называется операция вычисления пары полиномов вида:

$$(X_0, X_1) \mapsto (X_0 + WX_1, X_0 - WX_1) \mod q).$$

Algorithm A.5 Harvey NTT butterfly.

β is the word size, e.g. $\beta = 2^{64}$
on typical modern CPU platforms.
Require: $q < \beta/4; 0 < W < q$
Require: $W' = \lfloor W\beta/q \rfloor, 0 < W' < \beta$
Require: $0 \leq X_0, X_1 < 4q$
Ensure: $Y_0 \leftarrow X_0 + WX_1 \mod q; 0 \leq Y_0 < 4q$
Ensure: $Y_1 \leftarrow X_0 - WX_1 \mod q; 0 \leq Y_1 < 4q$

1. function HarveyNTTButterfly($X_0, X_1, W, W', q, \beta$)
2. if $X_0 \geq 2q$ then
3. $\quad X_0 \leftarrow X_0 - 2q$
4. end if
5. $Q \leftarrow \lfloor W'X_1/\beta \rfloor$
6. $T \leftarrow (WX_1 - Qq) \mod \beta$
7. $Y_0 \leftarrow X_0 + T$
8. $Y_1 \leftarrow X_0 - T + 2q$
9. return Y_0, Y_1
10. end function

Algorithm A.5 Shoup's invNTT butterfly

Require: $X, Y, W \in [0, p), p < \beta/2$
Ensure: $X' = X + Y \mod p, Y' = W(X - Y) \mod p$

1. $X' \leftarrow X + Y$
2. $X' \leftarrow X' - p$ if $X' \geq p$
3. $T \leftarrow W(X - Y)$
4. $T \leftarrow T + p$ if $T < 0$
5. $Q \leftarrow \lfloor W'T/\beta \rfloor$
6. $Y' \leftarrow (WT - Qp) \mod \beta$
7. $Y' \leftarrow Y' - p$ if $Y' \geq p$

Вклад Harvey в том чтобы убрать коррекцию в строке (4) и (7). Но при этом увеличивается требование к $p < \beta/4$.

Algorithm A.5 Harvey invNTT butterfly

Require: $X, Y \in [0, 2p)$; $W \in [0, p)$; $p < \beta/4$
Require: precomputed $W' = \lfloor W\beta/p \rfloor$
Ensure: $X' \equiv X + Y \mod p$, $Y' \equiv W(X - Y) \mod p$; $X', Y' \in [0, 2p)$

1. $X' \leftarrow X + Y$
2. $X' \leftarrow X' - 2p$ if $X' \geq 2p$
3. $T \leftarrow X - Y + 2p$
4. $Q \leftarrow \lfloor W'Y/\beta \rfloor$
5. $Y' \leftarrow (W'T - Qp) \mod \beta$

{обсуждение} О бабочках. Алгоритм скопирован из Intel HEXL. Они очевидно используют редукцию Соупа. Возможно ли подобрать условия таким образом, чтобы в строке (6) был точный ответ ($T < q$), без коррекции?

Мы хотим заменить операцию деления на константу на операцию умножения и сдвиг.

Т.е. для операции $\lfloor A/B \rfloor$ подобрать такое число C и n, чтобы на всем множестве чисел $0 \leq A < 2^{32}$ или на множестве $0 \leq A < B$ выполнялось тождество:

$\lfloor A/B \rfloor = \lfloor (A \cdot C)/\beta \rfloor \gg n$; -- целочисленное деление с округлением к меньшему.

все операции выполняются в целых 32-битных числах:

```
mul_hi(A,C)>>n;
```

Операцию обозначим $\lfloor (A \cdot C)/\beta \rfloor \gg n$.
Редукция Соупа использует $n = 0$.

Операция mul_hi - старшая часть результата умножения двух чисел. Такая оптимизация может выполняться на GPU, в языке OpenCL C.

Замена работает не для всех чисел.
Если бы число C было в диапазоне 2^32 <= C < 2^33, на один разряд больше, то метод

вероятно работал бы на всех числах.

Как преобразовать алгоритм, чтобы работало для всех чисел $0 < b < 2^{32}$?

- $C = 2^{32} + C_0$ -- используется такая замена.

$$\lfloor A/B \rfloor = (\lfloor (A \cdot C_0)/\beta \rfloor + A) \gg n$$

Алгоритм бабочки примет вид:

1. $q \leftarrow (\lfloor \bar{w}_0 x_1/\beta \rfloor + x_1) \gg n$
2. $r \leftarrow W x_1 - q \cdot p \pmod{\beta}$
3. $y_0 \leftarrow x_0 + r \pmod{p}$
4. $y_1 \leftarrow x_0 - r \pmod{p}$

Дополнительные ссылки:

- (https://github.com/IntelLabs/hexl/tree/main)
- [2021/1396] NTT software optimization using an extended Harvey butterfly

-- рассматривают оптимизиации NTT под векторные инструкции.

- (https://web.maths.unsw.edu.au/~davidharvey/talks/fastntt-2-talk.pdf)

-- описаны алгоритмы NTT с использованием редукции Соупа и Монтгомери. Однако, есть отличие с представлением, какой из алогоримов NTT называется прямым, какой обратным.

Algorithm A.6 Modular multiplication by constant

Require: $b, w \in [0, p), p < \beta$
Require: $\bar{w} \leftarrow \lfloor ((w \ll n) \cdot \beta)/p \rceil$, where $\bar{w} = \beta + \bar{w}_0$, $\beta \leftarrow 2^{32}$ or 2^{64}
Ensure: Returns $r = b \cdot w \mod p$

1. $q \leftarrow (\lfloor (b \cdot \bar{w}_0)/\beta \rfloor + b) \gg n$
2. $r \leftarrow b \cdot w - q \cdot p \pmod{\beta}$
3. $r \leftarrow r - p$ if $r \geqslant p$

Заметим, что способ вычисления $\bar{w} = (1. + \bar{w}_0 \cdot \beta^{-1}) \cdot 2^n$ совпадает с представлением вещественных чисел в формате `double` и методом округления RTN, используемым по

умолчанию. Для вычисления константы можно использовать функцию `frexp` из стандартной библиотеки C.

Algorithm A.7 Division by constant

Require: $0 < B < \beta$, $\beta = 2^L$
Ensure: Return (C_0, n); $C_0 \in [\beta/2, \beta)$, $n \in [0, L)$
Ensure: $\lfloor a/B \rfloor = (\lfloor (a \cdot C_0)/\beta \rfloor + a) \gg n$, for $\forall a \in [0, \beta)$
Ensure: $\lfloor a/B \rfloor = \lfloor (a \cdot C)/\beta \rfloor \gg n$, for $\forall a \in [0, \beta)$, $C = \beta + C_0$

1. $k \leftarrow \mathrm{ctz}_L(b)$
2. $b \leftarrow b \gg k$
3. if $b = 1$ then
4. $\quad C_0 \leftarrow 0, n \leftarrow 0$
5. else
6. $\quad n \leftarrow L - \mathrm{clz}_L(b)$
7. $\quad C_0 \leftarrow \lfloor ((2^L - b) \ll n)/b \rfloor + 2^n + 1$
8. end if
9. $n \leftarrow n + k$
10. return (C_0, n)

Упрощенный вариант деления можно предложить для случая $0 < a < 2^{L-1}$:

$$\lfloor a/B \rfloor = \lfloor (a \cdot C)/\beta \rfloor \gg n\,, \quad C \in [2^{L-1}, 2^L)$$

Вычисление произведения $a \cdot B \pmod p$:

1. $q = \lfloor (a \cdot B)/p \rfloor$ (quotient)
2. $r = a \cdot B - q \cdot p$ (remainder)

-- такой алгоритм просто следует из определения. В нем операция деления заменяется на операцию умножения со сдвигом. Можно подставить конкртеное выражение для модуля $p = 2^{32} - q_0 2^s + 1$, $n = 32$:

$$C_0 = \lfloor ((q_0 2^s - 1) \ll n)/p \rfloor + 2^n.$$

1. $q = (\lfloor (a \cdot C_0)/\beta \rfloor + a) \gg n$ -- quotient

2. $r = a \cdot B - q \cdot p$ -- remainder

```
/*! вычисляем константу `C0` и сдвиг `n` для замены деления:
    A/B == ((A*C0)>>32 + A)>>n  */
uint32_t div_c0(uint32_t b, int *nd)
{
        uint32_t C;
    int n, k;
        k = ctz(b);// count trailing zeros
        b>>=k;
        if (b==1) {
                C = 0; n = 0;
        } else {
                n = 32 - clz(b);// count leading zeros
                C = (uint64_t)(((1uLL<<32)-b)<<n)/b + (1uLL<<n) + 1;
        }
        *nd=n+k;
        return C;
}
```

Оптимизированные алгоритмы опираются на сдвиг 32-, 52- или 64 бит и отложенное редуцирование Баррета при сложении и вычитании.

Algorithm A.8 Montgomery's modular multiplication

Require: $b, w \in [0, p); p < \beta/2$
Require: precomputed $J = p^{-1} \mod \beta, w' = \beta w \mod p$
Ensure: Returns $r = b \cdot w \mod p$

1. $U = u_1\beta + u_0 \leftarrow bw'$ -- (wide product)
2. $q \leftarrow u_0 J \mod \beta$ -- (low product)
3. $h \leftarrow \lfloor qp/\beta \rfloor$ -- (high product)
4. $r \leftarrow u_1 - h$
5. $r \leftarrow r + p$ if $r < 0$

последнюю строчку можно исключить, если принять $r \in (-p, p)$. Две последовательные операции *low product* в строчке (1) и (2) можно объединить.

Algorithm A.8.1 Signed Montgomery's modular multiplication

Require: $b, w \in [0, p)$; $p < \beta/2$
Require: precompute $J = p^{-1} \bmod^{\pm} \beta$; $w' = w\beta \bmod p$;
Require: precompute $\tilde{w} = w' \cdot J \bmod^{\pm} \beta$ -- (signed low product)
Ensure: Returns $r \equiv b \cdot w \bmod^{\pm} p$

1. $u_1 \leftarrow \lfloor bw'/\beta \rfloor$ -- (signed high product)
2. $q \leftarrow b\tilde{w} \bmod^{\pm} \beta$ -- (signed low product)
3. $t \leftarrow \lfloor qp/\beta \rfloor$ -- (signed high product)
4. $r \leftarrow u_1 - t$

Для алгоритма важен метод вычисления обратного значения p^{-1}. Мы его вычисляем в 32 битных числах, потом приводим к знаковому типу $p^{-1} \bmod^{\pm} \beta$.

Listing Функция для вычисления $q^{-1} \bmod 2^s$, для $s < 32$:

```
uint32_t mod_inverse(uint32_t q, int s) {
    uint32_t q_inv = 1;
    for (int i = 1; i < s; i++) {
        if (((q * q_inv) & ((~0uL)>>(31-i))) != 1) {
            q_inv += (1u<<i);
        }
    }
    return q_inv;
}
```

Алгоритм -авторский, так и родился в виде программы. Базовая идея - инверсия по модулю должна давать единицу для любых s. Аналогичный метод может быть использован для нахождения константы при замене деления. Подобный метод возникал в конечных полях Галуа и все они, вероятно, могут быть выражены через расширенный алгоритм Евклида.

Algorithm A.9. Montgomery-Friendly multiplication

Require: $0 \leqslant a, b < p$, $p = K \cdot \beta - 1$, $\beta = 2^s$, $\beta/2 < K < \beta$
Ensure: $r = ab2^{-2s} \pmod p$

1. $r \leftarrow a \cdot b$
2. $r \leftarrow (r \bmod \beta) \cdot K + \lfloor r/\beta \rfloor$
3. $r \leftarrow (r \bmod \beta) \cdot K + \lfloor r/\beta \rfloor$
4. $r \leftarrow r - p \,\text{if}\, r \geqslant p$

Метод основан на операции сдвига на β на пол- слова и тождестве:

$$r\beta^{-1} \pmod p \equiv (r \bmod \beta) \cdot \lfloor (p+1)/\beta \rfloor + \lfloor r/\beta \rfloor, \text{ где } \beta = 2^{w/2} \leqslant 2^s \ .$$

$$(r \bmod \beta) + \lfloor r/\beta \rfloor \cdot \beta \equiv r \ , \quad \beta^{-1} \pmod p \equiv (p+1)/\beta \equiv a$$

$$\beta \cdot a \equiv 1 \pmod{\alpha\beta - 1}$$

Если сдвиг выполняется на полину разрядности, то операция выполняется дважды.

$$r\beta^{-2} = r\alpha^2 \pmod p$$

Операция состоит из двух вращений (2) и (3), каждое вращение сдвигает на β^{-1} и понижает разрядность на β. Заметим, что в случае когда модуль выравнен на слово и может быть прведен к виду $p = 2^n - \alpha_1\beta - 1$, то можно использовать цикллический сдвиг:

$$(r \bmod \beta) \cdot 2^{n-s} + \lfloor r/2^s \rfloor \equiv (r \ggg s) \ .$$

Операция циклического сдвига соответствует вращению по модулю $2^n - 1$.

Подобный метод мы использовали для редуцирования просле сдвига в алгоритме MWC64, для простых чисел Ризеля. На этом методе можно предложить функцию некриптографического MWC-хэша.

Нас в контексте ZKP интересуют модули вида $p = A\beta + 1$, пригодные для быстрого умножения с использованием NTT (NTT-friendly). Мы предлагаем такой алгоритм:

Algorithm A.9.1. NTT-Friendly modular multiplication

Require: $0 \leqslant ab < p2^{2s}$, $p = K \cdot \beta + 1$, $\beta = 2^s, \ 0 < K < \beta$
Ensure: $r = ab2^{-2s} \bmod p$

1. $r \leftarrow a \cdot b$ -- (wide product)
2. $r \leftarrow \lfloor r/\beta \rfloor - (r \bmod \beta) \cdot K$
3. $r \leftarrow \lfloor r/\beta \rfloor - (r \bmod \beta) \cdot K$
4. $r \leftarrow r + p \text{ if } r < 0$

Алгоритм основан на тождестве

$$r\beta^{-1} \pmod p \equiv \lfloor r/\beta \rfloor - (r \bmod \beta) \cdot \lfloor (p-1)/\beta \rfloor, \text{ где } \beta = 2^w \leqslant 2^s \ .$$

$$\beta \cdot a \equiv -1 \pmod{a\beta + 1}$$

Следующий алгоритм дополняет и обосновывает использование Хэш-функции на основе NTT-Friendly модулей.

Algorithm A.9.2. NTT-Friendly reduction

Require: $p = K \cdot 2^s + 1$, где $0 < K$, $\beta = 2^s$
Require: input $x = x_1\beta + x_0 \ , \ 0 \leqslant x < p\beta$
Ensure: output $r \equiv x\beta^{-1} \bmod p$

1. $r \leftarrow x_1 - x_0 \cdot K$
2. $r \leftarrow r + p \text{ if } r < 0$

Существует ещё ряд методов редуцирования специфичных для простых чисел и используемых в криптографии на эллиптических кривых, построенных на том же принципе что Алг.9. (см. [Plantard21] Efficient Word Size Modular Arithmetic).

В нашем контесте могут представлять интерес методы для чисел вида $p = 2^n - 2^k + 1$, где n и k кратны разрядности машинного слова, например $2^{64} - 2^{32} + 1$. Существует отдельный метод для модулей вида $p = 2^n - K$, где $K < 2^{n-w}$. Подобные модули надо рассматривать в другом контексте, в контексте multiprecistion, когда в результате операции сложения или умножения с накоплением возникает перенос разрядностью в слово и результат операции представлен в виде двух чисел $\{x, c\}$, $0 < x < 2^n$ и $c < 2^w$. Операция редуцирования сводится к обнулению остатка c, т.е найти такое число кратное модулю p, чтобы обнулить перенос. Это - не полное

редуцирование, т.н lazy reduction, после которого может потребоваться дополнительная операция коррекции. Коррекция может быть исключена, если следующая операция модульная.

Algorithm A.10. Multiprecision lazy reduction

Require: $p = 2^n - K$, где $K < 2^{n-w}$
Require: input $x < 2^n, c < 2^w$
Ensure: output $x \equiv \{x, c\} \bmod p$

1. $\{x, c\} \leftarrow \{x, c\} - c \cdot p$
2. $x \leftarrow x - p \text{ if } c > 0$ -- (overflow)
3. $x \leftarrow x - p \text{ if } x \geqslant p$ -- (optional correction)

Практически все модули в стандарте эллиптической криптографии удовлетворяют требованию этого алгоритма. Операция (1) может выглядеть проще в частном случае и умножение может сводиться к сдвигам и арифметическим операциям над машинным словом:

- $\{x, c\} \leftarrow x + c \cdot K$, для $p = 2^n - K$, $K < 2^{n-w}$
- $\{x, c\} \leftarrow x + c \cdot 2^s \mp \mathrm{c}$, для $p = 2^n - 2^s \pm 1$

Algorithm A.11. Plantard's word-size modular reduction

Require: $p < \beta/\phi, \phi = (1 + \sqrt{5})/2, \beta = 2^n$
Require: precompute: $R = p^{-1} \bmod \beta^2$
Require: input: $0 \leqslant ab \leqslant p\beta$
Ensure: output: $r \equiv ab(-\beta^{-2}) \bmod p$, $0 \leqslant r \leqslant p$

1. $r \leftarrow \lfloor (abR \bmod \beta^2)/\beta \rfloor$ -- одна операция high product?
2. $r \leftarrow \lfloor (rp + p)/\beta \rfloor$
3. $r \leftarrow r - p \text{ if } r = p$ -- optional

Proof: Пусть $q = abp^{-1} \bmod \beta^2$, где $\beta = 2^n$, тогда

$$qp - ab \equiv (abp^{-1})p - ab \pmod{\beta^2}$$
$$\equiv ab - ab \equiv 0 \pmod{\beta^2}$$

Мы получили $qp - ab$ делится на β^2, откуда получаем

$$ab(-\beta^{-2}) \bmod p \equiv \frac{qp - ab}{\beta^2}$$

Далее для вывода используется разложение $q = q_1\beta + q_0$

$$q_1 = \left\lfloor \frac{q}{\beta} \right\rfloor = \left\lfloor \frac{abp^{-1} \bmod \beta^2}{\beta} \right\rfloor$$

$$ab(-\beta^{-2}) \bmod p \equiv \frac{qp - ab}{\beta^2} \equiv ... \equiv \left\lfloor \frac{(q_1 + 1)p}{\beta} \right\rfloor$$

□

Это может оказаться самым эффективным методом умножения на константу в системе команд, где (2) - `madd_lo` - одна процессорная инструкция. Метод предназначен для использования в NTT. Константы вычисляются как $bp^{-1} \bmod \beta^2$. Возможно использование системы команд Intel AVX-IFMA.

- [2022/956] Improved Plantard Arithmetic for Lattice-based Cryptography
- [2309.00440] Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices

-- в работе представлен signed вариант модульного умножения и реализация алгоритма в инструкциях ARM Cortex-M3/-M4 и Risc-V.

Algorithm A.12. Signed Plantard multiplication

Require: input $a, b \in [-p2^{\alpha}, p2^{\alpha}]$, $p < 2^{w-\alpha-1}$
Require: precompute $R = q^{-1} \bmod^{\pm} \beta^2$
Ensure: output $r = ab(-\beta^{-2}) \bmod^{\pm} p$, где $r \in [-(p+1)/2, p/2)$

1. $r \leftarrow \lfloor (abR \bmod^{\pm} \beta^2)/\beta \rfloor$
2. $r \leftarrow \lfloor (rp + p2^{\alpha})/\beta \rfloor$

Алгоритм используется в NTT для умножения на константу [2309.00440]. Константы вычисляются методом $bp^{-1} \bmod {}^{\pm}\beta^2$.

Algorithm A.13. RNS to MRS conversion

Require: precompute $\gamma_{i,j} = p_i^{-1} \bmod p_j$
Require: input: $[x]_{\mathcal{B}}$ in RNS base $\mathcal{B} : \{p_i\}_{i=0}^{n-1}$
Ensure: output: x in MRS base

1. $\text{for } i \leftarrow 0 \text{ to } n-1 :$
2. $\quad \text{for } j \leftarrow i+1 \text{ to } n-1 :$
3. $\qquad x_j = (x_j - x_i) \bmod p_j$
4. $\qquad x_j = x_j p_i^{-1} \bmod p_j$
5. $\text{return } x$

Быстрое редуцирование по модулю $q = 2^{64} - 2^{32} + 1$

Метод редукции по модулю простого числа вида $\varphi^2 - \varphi + 1$. Обратите внимание, что это включает $p = 2^{64} - 2^{32} + 1$, где $\varphi = 2^{32}$. На интересует редуцирование после умножения, когда результат представлен в виде $x = x_0 + \varphi^2 x_1 + \varphi^3 x_2$.

Дано $\mathbb{F}_p$ для $p = \varphi^2 - \varphi + 1$, отдукда следует, что
$\varphi^2 = \varphi - 1 \Rightarrow \varphi^3 = \varphi^2 - \varphi = -1.$

Запишем значение x, как $x = x_0 + \varphi^2 x_1 + \varphi^3 x_2$,
где $x_0 \in Z_{2n}$ и $x_1, x_2 \in Z_n$, тогда
$x = x_0 + (\varphi - 1)x_1 - x_2 \pmod p.$

Специфика для использования в схемах BGV, BFV и CKKS

BGV и BFV: Эти схемы работают с целыми числами ($\mathbb{Z}/q\mathbb{Z}$) и в значительной степени зависят от NTT для умножения полиномов. Ядром умножения является операция модульного умножения. В наиболее эффективных алгоритмах используется модульное умножение Шоупа или модульное умножение Монтгомери.

GSW: Это схема работает с полиномами LWE. Шифротекст представляет собой матрицу, а гомоморфные операции (сложение и умножение) выполняются через матричные вычисления.

CKKS: Эта схема поддерживает приблизительную арифметику для вещественных чисел, что добавляет сложность в управлении точностью во время NTT и модульных операций. CKKS часто использует RNS для работы с большими модулями, а её операция пересчёта масштаба (для управления ростом шума) подчёркивает необходимость эффективного NTT.

Текущие тенденции и исследования
Аппаратное ускорение: Проекты, такие как Microsoft SEAL или IBM HElib, используют реализации на GPU/FPGA для NTT и арифметики полиномов. Специализированное оборудование для HE, например, в рамках программы DARPA DPRIVE, направлено на устранение этих узких мест.

Заключение
Модульное умножение полиномов с использованием NTT являются вычислительным ядром BGV, BFV и CKKS, и их оптимизация критически важна для практического внедрения HE. Достижения в алгоритмах, аппаратном ускорении и выборе параметров продолжают улучшать производительность, делая гомоморфное шифрование более практичным для реальных приложений.

R-LWE Cryptographic Algorithms

Algorithm 1: R-LWE Public Key Cryptosystem

Setup: Let $t = \lfloor \frac{q}{2} \rfloor$, $a, b \in \mathcal{R}_q$, and $s, e, r_0, r_1, r_2 \leftarrow \mathcal{X}$; then the public key encryption protocol between Alice and Bob:

Key Generation: Alice picks a random $a \in \mathcal{R}_q$ and samples $s, e \leftarrow \mathcal{X}$ to generate the public key $pk = \{a, b\}$ and the private key $sk = \{s\}$:

$$b = a \cdot s + e$$

where $\cdot$ is polynomial multiplication over the ring. Alice sends $\{a, b\}$ to Bob.

Encryption: Bob samples $r_0, r_1 \leftarrow \mathcal{X}$. He then converts his message into a binary vector (plaintext) m of length n, and generates the cipher $\{c_0, c_1\}$ as:

$$\begin{cases} c_0 = b \cdot r_0 + r_2 + tm, \\ c_1 = a \cdot r_0 + r_1 \end{cases}$$

Decryption: Alice decrypts the cipher by:

$$m = \lfloor (c_0 - c_1 \cdot s)/t \rceil,$$

where $\lfloor \cdot \rceil$ stands for taking the nearest binary integer.

{Algorithm 1.1 R-LWE Key Encapsulation}

Выбор параметров:

- For 128 bits of security, n = 512, q = 25601, and $\Phi(x) = x^{512} + 1$
- For 256 bits of security, n = 1024, q = 40961, and $\Phi(x) = x^{1024} + 1$
- n = 1024 and q = 12289

{Algorithm 1.1 R-LWE ML-KEM}

- [NIST:FIPS.203] Module-Lattice-Based Key-Encapsulation Mechanism Standard

Выбор параметров:

- For 256 bits of security, n = 256, q = 3329

Математическое определение.

1. Алиса генерирует матрицу $A \in \mathcal{R}_q^{k \times k}$ и выбирает случайные вектора $s, e \leftarrow \mathcal{X}^k$, чтобы сгенерировать открытый ключ (для зашифровывания) $ek = \{a, b\}$ и закрытый ключ (предназначеный для расшифровывания) $dk = \{s\}$:

$$\begin{cases} ek_{\text{PKE}} = (\langle \hat{A}, \hat{s} \rangle + \hat{e}, \hat{A}) \\ dk_{\text{PKE}} = \hat{s} \end{cases}$$

2. Боб выбирает случайный вектор $y, e_1 \leftarrow \mathcal{X}^k$ и генерирует шифратекст $c = \{c_0, c_1\}$ используя окртытый ключ ek_{PKE} и шум:

$$\begin{cases} c_1 = \langle \hat{A}^{\mathsf{T}}, \hat{y} \rangle + \hat{e}_1 \\ c_2 = \langle \hat{t}^{\mathsf{T}}, \hat{y} \rangle + \hat{e}_2 + \mu \end{cases}$$

где $(\hat{t} = \langle \hat{A}, \hat{s} \rangle + \hat{e}, \hat{A})$ - значение открытого ключа, а μ - сообщение, которое он хочет отправить, представленное в виде битового шума.

3. Алиса расшифровывает шифратекст c используя закрытый ключ dk_{PKE}:

$$m = \lfloor (c_2 - \langle \hat{s}^\mathsf{T}, c_1 \rangle)/t \rceil,$$

где $\lfloor \cdot \rceil$ обозначает ближайшее целое двоичное число.

Следует обратить внимание, что элементами векторов и матриц являются полиномы с коэффициентами в $\mathbb{Z}_q^N$. Генерция ошибок $\mathcal{X}$ определяется, как источник случайных чисел с "гауссовым" центрированным биномиальным распределением, для которого применяется преобразование в $\mathbb{Z}_q^N$. Источник случайных чисел может быть реализован с использованием криптографической хэш функции в качестве PRF или XOF. Источник шума реализуется как каскад из операций $\mathsf{CBD}_\eta(\mathsf{PRF}(\sigma))$. Стандарт предлагает использовать функцию shake256 как PRF. И shake128 для генерации XOF коэффициентов матрицы A в пространсвте точек NTT $\mathcal{T}_q$. Многообразие вариантов использования функций хеширования можно свести к единой функции пермутации, в режиме генерации XOF. Привязка может быть через использование XOF API и годится для генерации с исползованием SPONGE. Интересен вариант использования функции XOF на модульной арифметике в кольце $\mathcal{R}_q$.

Операция скалярного произведение выполняется с использованием правил умножения полиномов, которое может быть выполнено с использованием прямого и обратного NTT преобразования. В математической записи шляпа - это признак того, что операции выполняются в пространстве точек NTT: $\mathcal{R}_q \mapsto \mathcal{T}_q$.

Используя правила умножения матриц, можно записать
$\mathrm{c}_2 = \hat{s}^\mathsf{T} \circ \hat{A}^\mathsf{T} \circ \hat{y} + ...$
Остальные слагаемые связанные с шумом и произведением шума на шум, должны оказаться меньше единицы благодаря нормальному (гауссову) закону распрделения, но это не точно. Остается ненулевая вероятность, что схема не сойдется, и на выходе мы получим ошибку. Для этого в схему проверки надо встраивать значение хеш для контроля целостности данных.

В этой схеме матрица $\hat{A} \in \mathcal{T}_q^{k \times k}$ синтезируется с использованием криптографической хэш функции из $\rho \in \mathbb{B}^{32}$ - вектора случайных чисел 32 байта, который передается в составе ключа зашифровывания ek.

В стандарте точки в пространстве NTT $\mathcal{T}_q$ представлены парами со своим правилом умножения сопряженных пар. Эту схему я воспринимаю в *комплексных* числах. Операция скалярного произведения и транспонирования подразумевает сопряжение комплексных пар.

Algorithm 2: Oblivious Transfer Based on R-LWE

Public Key Encryption

Setup: Let KeyGen() be the Key Generation function of the sender Alice. Enc() the encryption function of the receiver Bob, and Dec() the decryption function of Alice as in Algorithm 1. Alice has N n bit binary messages $\{m_1, \cdots, m_N\}$ that Bob can choose from, and N n -byte random vectors $\{r_1, \cdots, r_N\}$ where $r_i \in \mathcal{R}_q$. Then the oblivious transfer between Alice the sender and Bob the receiver is as follows:

1. Alice sends $\{r_1, \cdots, r_N\}$ to Bob. Bob chooses the c^{th} vector r_c in order to acquire m_c. Then Bob generates a random binary vector $K \in \mathcal{R}_q^2$ and computes v to send to Alice:

$$v = r_c + \mathrm{Enc}_{pk}(K),$$

where r_c is added to both ciphertexts $\{c_0, c_1\}$.

2. For all $i \in \{1, 2, \cdots, N\}$, Alice computes the set $\{m_i'\}$ and sends it back to Bob:

$$m_i' = \mathrm{Dec}_{sk}(v - r_i) \oplus m_i,$$

where r_i is subtracted from both ciphertexts $\{c_0, c_1\}$ and $\oplus$ is bitwise XOR.

3. Bob computes his desired m_c while remaining oblivious to other m_i, where $i \neq c$:

$$m_c = m_c' \oplus K.$$

Algorithm 3: Zero-Knowledge Proof Based on R-LWE

Setup: Let $t = \lfloor \frac{q}{2} \rfloor$, $a, b, s \in \mathcal{R}_q$, and $e, r, u \leftarrow \mathcal{X}$.

Step: Suppose Alice has a secret s and needs to prove her ownership of it to Bob. It is notable that unlike the PKC scheme where $s \leftarrow \mathcal{X}$, in this ZKP scheme s can be any arbitrary value as $s \in \mathcal{R}_q$.

1. Alice picks a random $a \in \mathcal{R}_q$ and samples $e, r' \leftarrow \mathcal{X}$. Alice also selects an arbitrary binary vector m to compute:

$$\begin{cases} b = a \cdot s + e, \\ c = a \cdot r' + m + e' \end{cases}$$

where $\cdot$ is polynomial multiplication over the ring. Alice sends $\{a, b, c, m\}$ to Bob without revealing s.

2. Bob samples $u \leftarrow \mathcal{X}$, and interactively sends it to Alice.
3. Alice responds with x to Bob:

$$x = r + s \cdot u.$$

4. Bob verifies if:

$$\lfloor (c - a \cdot x + b \cdot u)/t \rfloor \stackrel{?}{=} m,$$

where $\lfloor \cdot \rfloor$ stands for taking the nearest binary integer. If the equality stands, then Alice has successfully proven her ownership of s to Bob.

Algorithm 4: NTT Polynomial Multiplication

Setup: let $a = \{a_0, \cdots, a_{n-1}\}$ and $b = \{b_0, \cdots, b_{n-1}\} \in \mathbb{Z}_q[x]/\langle f(x) \rangle$ be two polynomials of length n (with n coefficients), where
$f(x) = x^n + 1$ is an irreducible polynomial with a power of 2, and $q \equiv 1 \mod 2n$ is a large prime number.

Let ω be the n-th root of unity, ω^{-1} the inverse of ω, $\phi^2 = \omega \mod q$, and ϕ^{-1} the inverse of ϕ.

1. **Precompute**: $\{w, w^{-1}, \phi, \phi^{-1}\}$ for all $i \in \{0, 1, \cdots, n-1\}$
   /* negative wrap convolution of a and b */
2. for $i = 0$ to $n - 1$ do
   $\bar{a}_i \leftarrow a_i \phi^i$
   $\bar{b}_i \leftarrow b_i \phi^i$
3. end
   /* number-theoretic transformation a and b */
4. $\bar{A} \leftarrow \mathrm{NTT}_\omega^n(\bar{a})$
5. $\bar{B} \leftarrow \mathrm{NTT}_\omega^n(\bar{b})$
   /* component-wise multiplying A and B */
6. $\bar{C} = \bar{A} \cdot \bar{B}$
7. $\bar{c} \leftarrow \mathrm{invNTT}_\omega^n(\bar{C})$
8. for $i = 0$ to $n - 1$ do
   $c_i \leftarrow \bar{c}_i \phi^{-i}$
9. end
10. Return c

Заметим, что алгоритм использует версию NTT_ω^n для корня n-й степени из единицы $(\omega^n \equiv 1 \mod q)$. В алгоритме выполняется корректировка значений (negative wrap convolution, NWC) с использованием степеней корня 2n-й степени из единицы $(\phi^2 = \omega)$. Корректировку можно исключить, если использовать алгоритмы NTT_ϕ^n для негациклического полинома.

- [2024/585] A Complete Beginner Guide to the Number Theoretic Transform (NTT)

Приложение Б. Арифметизация тензорных операций

> Arithmetization techniques such as *R1CS* used in *Groth16*[37], *AIR* used for *FRI*-based commitments [11,12], and *Plonk* [26] and *Plonkstyle* arithmetizations (e.g., [25] used in halo2 [56]) make it possible to efficiently verify the correctness of a computation. Most of these proof systems internally use hash functions for the purpose of polynomial (Merkle tree) commitments.

Definition 1. (Arithmetic Circuit) A directed acyclic graph, with nodes as addition and multiplication gates, and edges as wires, over a finite field $\mathbb{F}_p$. Wires connect the outputs of one gate to the inputs of another.
Each gate has two inputs and one output wire, and the circuit has one final output wire. Figure (1) is an example of an arithmetic circuit.

Задача доказательства использования LLM включает в себя неинтерактивный протокол доказательства с нулевым разглашением (zk-SNARK) и правила арифметизации тензорных операций в графе вычислений.

Арифметизация R1CS (Rank-1 Constraint System) — это процесс преобразования вычислений, обычно выраженных в виде программы или арифметического выражения, в систему линейных ограничений над полем конечных характеристик. R1CS используется в системах доказательств с нулевым разглашением (т.к. zk-SNARKs или zk-STARKs), для проверки правильности вычислений.

R1CS представляет вычисление, как набор ограничений, где каждое ограничение имеет вид:
$\langle a, w \rangle \circ \langle b, w \rangle = \langle c, w \rangle,$
где:

w — вектор переменных (свидетельство),
a, b, c — векторы коэффициентов (строки матриц A, B, C),
$\langle \cdot, \cdot \rangle$ — скалярное произведение.

Каждое ограничение проверяет, что произведение двух линейных комбинаций переменных равно третьей линейной комбинации.

Этапы арифметизации

- Преобразование программы в плиномиальную форму: Исходная программа (например, на языке высокого уровня) преобразуется в арифметическую цепочку (Arithmetic Circuit), где все операции выражаются через сложение и умножение в конечном поле.

- Построение R1CS:
  Определяются входные данные, промежуточные переменные и выходные данные как элементы вектора w. Для каждой операции создаются соответствующие ограничения. Например, для умножения $z = x \cdot y$, добавляется ограничение $\langle a, w \rangle \cdot \langle b, w \rangle = \langle c, w \rangle$, где a, b, c выбираются так, чтобы отразить это равенство.

Матричная форма: Ограничения компактно записываются в виде матриц A, B, C, где каждая строка соответствует одному ограничению, а столбцы — переменным w.

- [2022/1048] Practical Sublinear Proofs for R1CS from Lattices

Пусть d — степень двойки, $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^d + 1)$.
Жирные заглавные буквы $\mathbf{p}$ обозначают элементы в R или R_q, а жирные строчные буквы с стрелками $\vec{b}$ представляют столбцы векторов с компонентами в R или R_q. Мы также используем жирные заглавные буквы для матриц B над R или R_q.

Кольцо R_q является $\mathbb{Z}_q$-модулем, порожденным базисом степеней $\{1, X, \ldots, X^{d-1}\}$.
Умножение гомоморфизма $x \mapsto ax$ для $a = a_0 + \cdots + a_{d-1}X^{d-1} \in R_q$ представлено негациклической матрицей:

$$\mathrm{Rot}(a) = \begin{pmatrix} a_0 & -a_{d-1} & \cdots & -a_1 \\ a_1 & a_0 & \cdots & -a_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_{d-1} & a_{d-2} & \cdots & a_0 \end{pmatrix} \in \mathbb{Z}_q^{d \times d}.$$

Произведение полиномов $a, b \in \mathcal{R}_q$ с редуцированием по полиному $(X^d + 1)$ можно записать в матричной форме как $\mathbf{c} = Rot(\mathbf{a}) \cdot \mathbf{b}$ или по элементам вектора:

$$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j} - \sum_{j=i+1}^{d-1} a_j b_{d+i-j} \pmod{q}$$

Это распространяется на R_q-модульные гомоморфизмы, заданные $A \in R^{m \times n}$ в блочном виде. Они представлены как $\mathrm{Rot}(A) \in \mathbb{Z}_q^{md \times nd}$.

В данной статье выбирается простое q, такое что $\mathbb{Z}_q$ содержит примитивный $2d$-й корень единицы $\zeta \in \mathbb{Z}_q$, но не содержит элементы, чей порядок является более высокой степенью двойки, т.е. $q - 1 \equiv 2d \pmod{4d}$. Следовательно, у нас есть:

$$X^d + 1 \equiv \prod_{j=0}^{d-1} (X - \zeta^{2j+1}) \pmod{q},$$

где $\zeta^{2j+1} (j \in \mathbb{Z}_d)$ пробегает все d примитивных $2d$-х корней единицы. Мы определяем Численное Теоретическое Преобразование (NTT) многочлена $p \in R_q$ следующим образом:

$$\mathrm{NTT}(p) := \begin{bmatrix} \hat{p}_0 \\ \vdots \\ \hat{p}_{d-1} \end{bmatrix} \in \mathbb{Z}_q^d \quad \text{где} \quad \hat{p}_j = p \mod (X - \zeta^{2j+1}).$$

Для любых полиномомв $f, g \in R_q$ выполняется $\mathrm{NTT}(f) \odot \mathrm{NTT}(g) = \mathrm{NTT}(fg)$, где $\odot$ -- поэлементное умножение векторов.

Мы также определяем обратную операцию NTT. А именно, для вектора $\vec{v} \in \mathbb{Z}_q^d$ $\mathrm{NTT}^{-1}(\vec{v})$ — это многочлен $p \in R_q$, такой что $\mathrm{NTT}(p) = \vec{v}$.

Норма и длина вектора
Для элемента $w \in \mathbb{Z}_q$ под $|w|$ мы понимаем $|w \mod q|$. Определяем нормы ℓ_∞ и ℓ_2 для $w \in \mathcal{R}_q$ следующим образом:

$$\|w\|_\infty = \max_i |w_i| \quad \text{и} \quad \|w\|_2 = \sqrt{|w_0|^2 + \cdots + |w_{d-1}|^2}.$$

Аналогично, для $\vec{w} = (w_1, \ldots, w_k) \in \mathcal{R}_q^k$ определяем:

$$\|\vec{w}\|_\infty = \max_i \|w_i\|_\infty \quad \text{и} \quad \|\vec{w}\|_2 = \sqrt{\|w_1\|_2^2 + \cdots + \|w_k\|_2^2}.$$

Тензорный ассемблер

У нас есть свое видение того как должна выглядеть система доказательства. Не могу сказать что нравится, но это то что сидит в голове и требует осмысления. Если рассматривать систему как процессор. Вероятно в этом случае надо говорить про некоторую виртуальную машину zkVM которая умеет обрабатывать тензорные операции в вычислительном графе, но выполняет это поверх модульной арфиметики содержащей только умножения и сложения. Мы хотим расширить понятие полиномиальной арифметики на рациональные функции, рассматривая их как отношение полиномов, т.е. одна из арифметических операций - обратное значение. Мы также хотим описывать операции в терминах дифференциальных операторов над векторами, выраженных через базисные полиномы на интервале [0,1] или [-1,1]. Операторы также могут быть записаны, как рациональные - отношение передаточных функций.

Дополнительным требованием является возможность использования *компрессии* векторов, которая позволяет сократить объем данных на входе и выходе тензорной операции. Компрессия

определяется для строк матрицы и для векторов. Также мы определяем правила компрессии для тензоров использующих пермутации, транспонирование и броадкастинг данных.

Функция компрессии определяется для вектора v. Функция криптографического хэша может быть использована для получения компрессии.

Cryptographic Compression Functions. Пусть $\mathcal{P}$ выбранная функция смешивания (permutation) на поле $\mathbb{F}^t$. Операцию можно определить как редуцирование вектора t элементов в вектор n -элементов $\mathbb{F}^t \mapsto \mathbb{F}^n$, где $n < t \leq 2n$.:

$$\mathcal{C} : \mathbb{F}^t \mapsto \mathbb{F}^n, \quad \mathcal{C}(v) = \mathrm{Tr}_n(\mathcal{P}(v) + M \cdot v) \in \mathbb{F}^n$$

В операции могут использоваться матрицы перестановок основанные на циркулянтах и нелинейные полиномиальные преобразования, наравне с обратными значениями - арифметические операции свойственные рациональным функциям. Операция truncate - младшая часть вектора. Операцию truncate, в частном случае, можно рассматривать как редуцирование по полиному $\mathbb{F}_q/\langle x^n - 1\rangle$. Умножение на матрицу M - линейная операция, не влияет на сложность. Заметим, что функция редуцирования без смешивания $\mathcal{P}(v)$ не является криптографической. Наоборот на базе критографической функции XOF (eXtended Output Function) можно определить функцию редуцирования с заданными параметрами. Так можно рассматривать функции типа SHAKE-256 и SHAKE-128, а так же блочные хэши SHA256. Однако, перспективным направлением представляется использование функции редуцирования с заданными параметрами поля, определенную через модульную арифметику на кольце. Функцией компрессии может быть функция с ключом или параметром кастомизации, такие как KMAC-XOF (KECCAK Message Authentication Code with Arbitrary-Length Output, заменяющая собой CMAC и HMAC) и cSHAKE (кастомизированная версия алгоритма SHAKE) определенные в [NIST SP 800-185].

> In cryptography, the Merkle–Damgård construction is a method of building collision-resistant cryptographic hash functions from collision-resistant one-way compression functions.

Альтернативно можно рассматривать $\mathcal{P}(v)$ как поток псевдослучайных чисел (PRF) с гарантированным большим периодом повтора и зависимый от ключа и IV (параметров инициализации). Для реализации на GPU нам важно чтобы функция редуцирования могла выполняться параллельно в варп-группе кратно 32 и содержала полиномиальную арифметику с использованием 32-х битных операций или тензорных операций пониженной размерности, сдвиговые операции MDS также должны выполняться параллельно в группе. Матрицы могут строится, как сумма диагональных матриц и матриц-циркулянтов.

Пример на языке OpenCL C, умножение на матрицу MDS циклические перестановки в варп-группе выполняются на базе функций `sub_group_rotate()`, `sub_group_reduce_add()` и

`sub_group_broadcast()` . Выигрыш по производительности получается при использовании матричного умножения на тензорных ядрах с накоплением результата в регистрах i32 бит при вычислениях на тензорных ядрах. Наибольшую скорость линейной записи и чтения данных можно получить на векторах 1024 бит с загрузкой данных в режиме interleaving между ядрами в варп-группе. Это справедливо для всех архитектур GPU: NVIDIA, AMD и Intel. Чтобы задействовать тензорные ядра необходимо ориентироваться на целочисленную операцию *dot product* с накоплением результата в регистре i32.

{рассмотреть функцию нормализации слоя}

{рассмотреть функцию softmax}

{рассмотреть поэлементные нелинейные функции}

Рассмотрим пример преобразования тензорной операции с использованием циклической или негациклической свертки в полиномиальную форму для FHE:

$$\mathrm{NWC}(a,b) = ... = \sum_{i=0}^{N-1} w_i x^i \mod (x^N+1),$$

где x, w — входные данные и веса, представленные, как полиномы в $\mathbb{Z}_q[x]/\langle x^N+1\rangle$.

Определение негациклической свёртки (NWC, Negacyclic Convolution)

Негациклическая свёртка (Negacyclic Convolution, NWC Negative Wrap Convolution) — это операция умножения двух полиномов в кольце полиномов $\mathbb{Z}_q[x]/\langle x^N+1\rangle$, где q — модуль (обычно простое число), а N — степень полинома, выбираемая как степень двойки. Эта операция является ключевой в криптографических системах, таких как Ring-LWE, используемых в пост-квантовой криптографии (PQC) и полностью гомоморфном шифровании (FHE).

Формальное определение.
Пусть даны два полинома $a(x) = \sum_{i=0}^{N-1} a_i x^i$ и $b(x) = \sum_{i=0}^{N-1} b_i x^i$ с коэффициентами в $\mathbb{Z}_q$. Негациклическая счётка $c(x) = a(x) * b(x) \mod (x^N+1)$ определяется как полином $c(x) = \sum_{i=0}^{N-1} c_i x^i$, где коэффициенты c_i вычисляются по формуле:

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{N-1} (-a_j) b_{N+i-j} \mod q,$$

где:

- Первая сумма $\sum_{j=0}^{i} a_j b_{i-j}$ соответствует стандартной свёртке для индексов, не превышающих i.
- Вторая сумма $\sum_{j=i+1}^{N-1} (-a_j) b_{N+i-j}$ учитывает отрицательный вклад после редуцирования по модулю $x^N + 1$.

Отличие от циклической свёртки

В отличие от циклической свёртки, где редукция выполняется по модулю $x^N - 1$, негациклическая свёртка использует редуцирование по полиному $x^N + 1$. Это приводит к появлению отрицательных коэффициентов во второй части выражения, что делает её подходящей для криптографических приложений, таких как Ring-LWE, где полиномиальное кольцо $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^N + 1 \rangle$ обеспечивает устойчивость к квантовым атакам.

Реализация с использованием NTT

Для эффективного вычисления негациклической свёртки используется *быстрое теоретико-числовое преобразование (NTT, Number Theoretic Transform)*. NTT позволяет преобразовать полиномы $a(x)$ и $b(x)$ в частотную область, где их произведение становится поэлементным умножением:

$$c(x) = \mathrm{NTT}^{-1}(\mathrm{NTT}(a) \odot \mathrm{NTT}(b)) \mod (x^N + 1),$$

где $\odot$ обозначает поэлементное умножение векторов коэффициентов.

Использование матриц циклических перестановок позволяет заменить матричную операцию на поэлементное умножение векторов. Например,

$$c(x) = \mathrm{NTT}^{-1}(\mathrm{NTT}(MDS(:,0)) \odot \mathrm{NTT}(b)),$$

Алгебраические свойства негациклической свёртки

1. *Коммутативность*
   $a(x) \cdot b(x) = b(x) \cdot a(x) \mod (x^N + 1)$.
2. *Ассоциативность*
   Для любых трёх полиномов $a(x), b(x), c(x) \in \mathcal{R}_q$:
   $(a(x) \cdot b(x)) \cdot c(x) = a(x) \cdot (b(x) \cdot c(x)) \mod (x^N + 1)$.
3. *Дистрибутивность по сложению*
   Для любых полиномов $a(x), b(x), c(x) \in \mathcal{R}_q$:
   $a(x) \cdot (b(x) + c(x)) = a(x) \cdot b(x) + a(x) \cdot c(x) \mod (x^N + 1)$.
4. *Свойство редукции по полиному $x^N + 1$*:
5. *Симметрия и структура кольца*:
   $x^N + 1 = \prod_{i=0}^{N/2-1} (x^2 - \zeta^{2\mathrm{BitRev}_N(i)+1})$,

где ζ — примитивный корень степени N из единицы.

6. *Обратимость, существование обратного элемента*
   В кольце $\mathbb{Z}_q[x]/\langle x^N+1\rangle$,где q — простое число и полином $a(x)$ обратим. Существует $b(x)$, такой что $a(x)\cdot b(x)\equiv 1 \mod (x^N+1)$, негациклическая свёртка с $a(x)$ обратима. Это свойство важно для криптографических приложений, где требуется вычисление обратных элементов для шифрования. Это свойство мы применяем для рациональных функций.

Условие обратимости: Полином $a(x)$ обратим, если он взаимно прост с x^N+1, не имеет общих корней.

- [2025/166] Polynomial Inversion Algorithms in Constant Time for Post-Quantum Cryptography
- Seo, EY., Kim, YS. & No, JS. Fast polynomial inversion algorithms for the post-quantum cryptography. J Cryptogr Eng 15, 16 (2025). (https://doi.org/10.1007/s13389-025-00380-w)
- Bernstein, D.J., Yang, B.: Fast constant-time gcd computation and modular inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems 2019(3), 340–398 (2019) (https://doi.org/10.13154/tches.v2019.i3.340-398)

Свойство гомоморфизма:
Негациклическая свёртка поддерживает гомоморфные свойства в контексте FHE. Если $a(x), b(x)$ представляют зашифрованные данные, то их негациклическая свёртка соответствует умножению исходных сообщений в зашифрованном виде:
$c(x) = a(x)\cdot b(x) \mod (x^N+1) \implies D_k(c(x)) = D_k(a(x))\cdot D_k(b(x)),$
где D_k — функция расшифровки.

Входные данные имеют приведенный формат $x=\{x_0,x_1,\cdots,x_{N-1}\}$ и $w=\{w_0,w_1,\cdots,w_{N-1}\}$. Все операции с тензорами должны выражаться через полиномиальное умножение и сложение. Приведение к полиномиальной форме выполняется в два этапа.

Приведение к целочисленной арифметике через квантизацию, напрмер MXINT8, MXFP8, FP16, BF16.

Приведение к полиномиальной форме путем редуцирования полинома до заданной степени.

Квантизация порождает остатки - квантовый шум. Остатки накапливаются и компенсируются в процессе расчета, согласно правилам PTQ, и используются в качестве параметров алгоритма шифрования Ring-LWE.

Построение схемы MLP. Необходимо выразить функцию MLP в виде полиномов заданной степени, выражения можно строить с использованием базисных полиномов Бернштейна, Лежандра, Чебышева и Якоби. Построение полиномов должно выполнятья реккурентно с увеличением степени полинома.
В качестве базовых полиномов на интервале [-1,1] можно использовать обобщенные полиномы Бернштейна.

Базисные полиномы Бернштейна могут быть обобщены для покрытия произвольного интервала $[a, b]$ путём нормализации t над этим интервалом, то есть $t = (x - a)/(b - a)$, что приводит к следующему выражению:

$$B_{j,n}(x) = \binom{n}{j} \frac{(x-a)^j (b-x)^{n-j}}{(b-a)^n}, \quad j = 0, 1, \ldots, n\ .$$

Полиномы Бернштейна удовлетворяют симметрии $B_{j,n}(x) = B_{n-j,n}(1-x)$, положительности $B_{j,n}(x) \geq 0$, и $\sum_{j=0}^{n} B_{j,n}(x) = 1$ на определяющем интервале $[a, b]$. Для разложения рациональных функций полезным является интервал $[-1, 1]$. Тогда система полиномов Бернштейна примет вид

$$B_{j,n}(x) = \binom{n}{j} \frac{(x+1)^j (1-x)^{n-j}}{2^n}, \quad j = 0, 1, \ldots, n.$$

Макрооперациями являются softmax, Attention, FFN с ипользованием GELU, SiLU. Базовый метод - умножение матрицы на вектор и сложение векторов. Все нелинейные функции можно выразить через поэлементную экспоненту от вектора, входит в *Softmax*, и функцию *Sigmoid*, входит в GELU, SiLU. Дополнительно следует рассмотреть функцию нормализации слоя LayerNorm на базе 2-Norm. Для этих трех функций надо определить полиномиальные приближения.

Согласно теореме KAT, на единичном интервале любую фукнцию $f : [0, 1]^n \to \mathbb{R}$ можно разложить в виде композиции(суперпозиции) из непрерывных функций одного переменного ...

Every multivariate continuous function
$f : [0, 1]^n \to \mathbb{R}$ can be represented as a superposition of continuous single-variable functions.

$$f(\mathbf{x}) = f(x_1, \ldots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right).$$

where
$\phi_{q,p} \colon [0, 1] \to \mathbb{R}$ and $\Phi_q \colon \mathbb{R} \to \mathbb{R}$.

В формулировке KAN мы можем искать аппроксимацию для функций Φ_q и $\phi_{q,p}$ в форме B-сплайнов (базисные полиномы Бернштейна), заданной степени (кубических сплайнов). Аналогией может быть бикубические фильтры построенные на сплайнах, их обобщение на большее число координат - мультивариативные кубические сплайны.

Эта формула слишком абстрактаная, не дает способа разложения. Не адаптирована для каскадного применения, т.е. не содержит функции нормализации на входе и выходе. Нужен алгоритм разложения MLP в форму KAN. В форме KAN функции представляются полиномами 3-й степени.

В нашей формулировке разложение может быть представлено в виде свертки по системе полиномов:

- $\{x^k(1-x)^{n-k}\}_{k=0}^n$ - разложение функции по базисным полиномам Бернштейна на интервале $[0,1]$;
- $\{(x+1)^k(1-x)^{n-k}\}_{k=0}^n$ - разложение по обобщенным полиномам Бернштейна на интервале $[-1,1]$.

Заметим, что все разложения данной степени изоморфны, т.е. любое разложение можно привести в разложение по степеням $\sum_{j=0}^n a_k x^k$ используя матрицы. Например интерполяционный полином в форме Лагранжа можно привести к разложению по степеням и можно привести к разложению по системе полиномов Бернштейна.

Приложение C. Референсные алгоритмы

Проблема отладки сводится к сравнению результатов референсной реализации и векторной реализации для CPU или GPU.

Алгоритм C.1 Редуцирование полиномов в кольце Rq

Два действия используются при редуцировании полиномов

1. Folding (схлопывание) $\ell = nL$
2. редуцирование $L < \ell < 2L$

```
if (len%L){
    for (int k=0;k<len%L;k++) {// по маске
        d = m[k] - m[k+N];
        m[k] = d<0? d+q : d;// mod q
    }
}
for (int i=len/L; i>0; --i){// folding
    for (int k=0;k<L;k++) {
        d = m[k] - m[k+N];
        m[k] = d<0? d+q : d;// mod q
    }
}
```

Операция может выполняться параллельно на длине nL.
Основано на векторной операции коррекции после вычитания $d = (s - s_1) \mod q$, которая может быть представлена в системе команд Intel AVX512 как `_mm512_min_epu32(d, d+q)` .
Операция коррекции после сложения по модулю $a = (s + s_1 \mod q)$ может быть выражена в векторном виде как `_mm512_min_epu32(a, a-q)` .

Алгоритм C.2 Ротация полиномов на кольце Rq

Алгоритм C.3 Умножение полиномов на кольце Rq

Алгоритм C.4 MWC - Генерация псевдо-случайных чисел

Алгоритм может быть эффективно реализован на GPU в варп-группе по 32. Модули мы выбираем из ряда совместимых для операции MWC32 (генерация последовательности PRNG).

- MWC128 - содержит список модулей для MWC32.

Рассмотрим способ получения последующего случайного числа методом MWC. Число представлено двумя частями, старшая - перенос(c) и младшая (x), $\beta = 2^N$

$$\{x_{i+1}, c_{i+1}\} = (x_i + c_i\beta) \cdot a \mod (a\beta - 1) \equiv x_i a + c_i$$

генерация последовательности это последовательное умножение на a взятое по модулю, которое вычисляется через умножение с переносом благодаря выбору модуля.
Приведу доказательство через преобразование

$$(x_i + c_i\beta) \cdot a = x_i a + c_i(a\beta - 1) + c_i \mod (a\beta - 1)$$

второе слагаемое в выражении равно нулю, поскольку кратно модулю.
Пропуск сегмента в - это умножение на a^N по модулю.

{рассмотреть генерацию последовательности по модулю} $q = a\beta - 1$.

```
uint32_t mwc32_next(uint32_t h, const uint32_t A){
    h = (h&0xFFFF)*A + (h>>16);
    return h;
}


uint64_t mwc64_next(uint64_t h, const uint64_t A1){
    h = (h&0xFFFFFFFFu)*A1 + (h>>32);
    return h;
}


uint64_t mwc64_seed(uint64_t h, const uint64_t A1){
    const uint64_t q = (A1<<32)-1;
    if (h>=q) h-=q;
    return h;
}
```

{рассмотреть генерацию последовательности по модулю} $q = a\beta + 1$.

$$\{x_{i+1}, c_{i+1}\} = (x_i + c_i\beta) \cdot a \mod (a\beta + 1) \equiv x_i a - c_i$$

{возможно использование редуцирования по модулю со знаком}

```
int32_t mwc32s_next(int32_t h, const int32_t A){
    h = (h>>16) - (h&0xFFFFu)*A;
    return h;
}


int64_t mwc64s_next(int64_t h, const int32_t A){
    h = (h>>32) - (h&0xFFFFFFFFu)*A;
    return h;
}
```

Алгоритм С.5 Кодирование вещественных чисел

Перевод вещественных чисел в операции по модулю. Вещественые числа floatN можно представить как сдвиг и редуцирование. По сути экспонента в модульной арифметике заменяется на умножение по модулю. Таким образом преобразование номализованного числа в целое по модулю будет представлять из себя умножение на констранту по таблице экспонент 2^N. Эта операция соответствует поэлементному модульному умножению векторов.

Числа с плавающей точкой представляются в форме нормализованного или денормализованного числа:
$\pm(1. + m) \cdot 2^e$ или $\pm(m) \cdot 2^{e-t}$, где t - разрядность мантиссы.

Мы хотим обратимым (deterministic) образом кодировать числа с плавающей точкой в целые числа по модулю. Для этого мы выполняем операцию $\pm m \cdot 2^{e-t} (\mod \pm q)$ для нормализованного числа $[0.5, 1)$. Если разрядность чисел превышает разрядность модуля, то выполняется редуцирование мантиссы по модулю. Подходящий способ редуцирования - это модульная редукция Монтгомери со знаком.

Пусть $K_s = 2^s \mod q$ -- сдвиговые константы $e_{min} \leq s \leq e_{max}$.

Преобразование типа float (псевдокод):

```
uint32_t convert_f32_to_mwc32(float f){
    const uint32_t M = (1u<<24);// число бит в матиссе для нормализованного числа 23+1
    f = frexpf(f,&ex);     // загрузка экспоненты, результат в интервале [0.5,1)
    int32_t m = rint(f*M);// округление до ближайшего целого RNE
    if (m<0) m = q+m;
    return ((uint32_t)m*(uint64_t)K[ex+E_OFF])%q;// умножение на сдвиговую константу по модулю
}
```

Вещественное число в формате FloatN {sign, exp, mant} харакеризуется разрядностью мантиссы и экспоненты. Для экспоненты используется сдвиговые константы определенные в диазоне от -E_OFF. E_OFF - минимальная степень возвращаемая функцией frexpf(). Алгоритм корректно работает со всеми нормализоваными и денормализованными числами.

Обратное преобразование типа float (псевдокод):

```
float convert_mwc32_to_f32(uint32_t m, int ex){
    m = (m*(uint64_t)Kr[ex+E_OFF])%q; // умножение на сдвиговую константу по модулю `q`
    int i = (m>>31)?(int)m-q:m; // восстановить знак
    return ldexpf(i, ex-24);
}
```

Второй вариант преобразования, соответствует операциям модульной арифметики со знаком $-q < m < q, q < 2^{31}$.

```
int32_t convert_f32_to_mwc32s(float f){
    f = frexpf(f,&ex);     // загрузка экспоненты, результат в интервале [0.5,1)
    int32_t m = rint(f*M);// округление до ближайшего целого RNE
    return (m*(int64_t)K[ex+E_OFF])%q;// умножение на сдвиговую константу по модулю `q`
}


float convert_mwc32s_to_f32(int32_t m, int ex){
    m = (m*(int64_t)Kr[ex+E_OFF])%q; // умножение на сдвиговую константу по модулю `q`
    return ldexpf(m, ex-24);
}
```

Расчет сдвиговых констант $K_s = 2^s \mod q$ и обратных констант $K_{-s}^{(r)} = 2^{-s} \mod q$ выполняется с использованием функции модульного возведения в степень. Параметром преобразования для разных типов данных является M- число бит в мантиссе.

Методы округления: Stochastic rounding (SR)

Пример округления RNE при переходе между типами F32 и E4M3 для нормализованных чисел:

```
        uint32_t fixup = (m>>20)&1;
        m = (m + 0x7ffffu+fixup)>>20;
```

Аналогично выглядит переход между типами F64 и FP16 для нормализованных чисел:.

Заслуживает внимания тема округления с распределением ошибки, т.н. stochastic rounding.

- [US10684824B2] Stochastic rounding of numerical values

{разобрать пример и алгоритм stochastic rounding.}.

Пример аппаратной реалищации Stochastic Rounding (SR) при переходе из F32 (E8M23) в TF32 (E8M10) или FP16 (E5M10), $D = 23 - 18$:

```
uint32_t prob = m&0xFF;// биты 16..23
m = ((m>>D) + prob)>>8;// биты 11..18
```

Такое округление имеет смысл, если мы договоримся хранить вероятности обратной ошибки в младшей части мантиссы, а начальное значение вероятности будет задано нормальным распределением.

Stochastic Rounding следует рассматривать в контексте операции *dot product* или FMA. Следует доказать, что ошибка округления не накапливается при выполнении векторных операций, когда один или оба аргумента содержат случайную величину ошибки с нормальным распределением.

- [doi:10.1137/20M1334796] Stochastic Rounding and Its Probabilistic Backward Error Analysis, 2021

В контексте методов округления **backward error** (обратная ошибка) — это мера того, насколько входные данные должны быть изменены, чтобы точное решение задачи с этими изменёнными данными совпадало с полученным приближённым решением.

Формально, если $f(x)$ — точная функция, а $\tilde{f}(\tilde{x})$ — приближённое решение, полученное с использованием метода округления, то обратная ошибка определяется как величина Δx, такая что:

$$f(x + \Delta x) = \tilde{f}(x).$$

В численных методах, связанных с округлением, обратная ошибка показывает, насколько нужно "подправить" исходные данные, чтобы результат вычислений с учётом ошибок округления был точным. Это полезно для оценки устойчивости алгоритма: меньшая обратная ошибка указывает на более устойчивый метод.

В работе метод SR (Stochastic Rounding) определяется через случайное распределение ошибки округления $\delta : \mathrm{fl}(x) = x(1 + \delta)$ с мат ожиданием $\mathsf{E}(\delta) = 0$. Ррассмотрен ряд теорем, которые показывают, что операции с плавающей точкой при использовании стахостического округления в операциях сложения, умножения, операции dot-prouct и matrix-multiplication сохраняют вероятностные свойства ошибок округления. Случайный характер ошибки определяется в рамках модели ошибок аргументов векторных операций, когда ошибка последующего аргумента не коррелирует (не связана) с ошибкой предыдущих аргументов. В той же модели доказана теорема о нормальном распределении ошибки для решения линейной системы уравнений.

Таким образом работа содержит достаточно полное обоснование выбора stochastic rounding для операций с плавающей точкой при использовании форматов пониженной разрядности. Ряд теорем обосновывает использование SR в задачах машинного обучения, вернее линейной алгебры и полиномиальной арифметики. Открытым остается вопрос использования SR при вычислении обратных значений.

Заметим, что для реализации SR требуется N-независимых генераторов случайных чисел, которые могут быть реализованы аппаратно со смещением начального значения. В приложении криптографических методов защиты и ZKP мы рассматриваем методы MWC для параллельной генерации случайных чисел со смещением или с разными модулями.

Определение (Stochastic Rounding).
Let $\mathcal{F} \subseteq \mathbb{R}$ denote a number system. For $x \in \mathbb{R}$, define the two rounding candidates

$$\lfloor x \rfloor = \min\{y \in \mathcal{F} : y \leq x\}, \quad \lceil x \rceil = \max\{y \in \mathcal{F} : y \geq x\}.$$

so that $\lfloor x \rfloor \leq x \leq \lceil x \rceil$, with equality throughout if $x \in \mathcal{F}$. Note that when $x \notin F$, the two numbers $\lfloor x \rfloor$ and $\lceil x \rceil$ are adjacent in $\mathcal{F}$. We denote by fl any rounding operator that maps numbers in $\mathbb{R}$ to either of the two nearest numbers in $\mathcal{F}$.
For $x \in \mathbb{R} \backslash \mathcal{F}$, SR is defined by

$$\mathrm{fl}(x) = \begin{cases} \lceil x \rceil, & \text{with probability } q(x), \\ \lfloor x \rfloor, & \text{with probability } 1 - q(x) \end{cases}$$

где $q(x)$ рассматривается как вероятность того что результат примет значение $\mathrm{fl}(x) = \lceil x \rceil$. Для чисел пониженной разрядности, явно проявляются нелинейности рапределения чисел. Для компенсации такой нелинейности предлагается использовать соотношение (mode 1):

$$q(x) = \frac{x - \lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor}$$

Лемма Если $y = \mathrm{fl}(x) = x(1 + \delta)$, где $x \in \mathbb{R}$, получена методом стохастического округлением, то δ является случайной величиной с $\mathsf{E}(\delta) = 0$.

Доказательство.
Очевидно, $\mathrm{fl}(x)$ и $\delta = (\mathrm{fl}(x) - x)/x$ являются случайными величинами. Имеем:

$$\mathsf{E}(\mathrm{fl}(x)) = p\lceil x \rceil + (1 - p)\lfloor x \rfloor = \frac{(x - \lfloor x \rfloor)\lceil x \rceil + (\lceil x \rceil - x)\lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor} = x.$$

Тогда $\mathsf{E}(\delta) = \mathsf{E}((\mathrm{fl}(x) - x)/x) = 0$.

□

- [2410.10517] Stochastic Rounding 2.0, with a View towards Complexity Analysis

Накопление и компенасция ошибки. Мы рассмотрим механизм компенсации ошибки квантизации в процессе вычислений. Допустим на каждом цикле остаток $q_i = x_i + q_{i-1} \pmod \beta$, где $q(x)$ - остаток при выполнении квантизации. Остатки мы будем исчислять в формате Bfloat16 или FP16 в то время как основное значение представлено в формате BFloat16. Пространственное накопление ошибки с усреднением по соседним значениям может быть выполнено по варп-группе. Такого рода компенсайия ошибки на потоке данных дает вероятностное представление ошибки (детерминированное). В тоже время может быть предложен вариант компенсации ошибки с использованием генератора случайных чисел. В патенте nVidia [US10684824B2] предлагается использовать метод аппаратного округления смахивающий на генератор случайных чисел для компенсации ошибки, при котором младшая часть мантиссы при округлении считается случайной величиной и добавляется к окргугляемому значению $\lfloor (m[16:23] + m[11:18])/2^8 \rfloor$.

Заметим что есть три разные трактовки stochastic rounding. В первом случае используется соотношение (mode 1), во втором используется статистический характер распределения квантовой ошибки на потоке данных, и в случае аппаратной реализации возникает что-то третье - метод снижения влияния ошибки округления основанный на вероятностном распределении ошибки округления при выполнении векторных операций таких как dot product. Метод RNE является наиболее распространенным и в стандарте IEEE 754-2019 упомянут как метод используемый по умолчанию. Stochastic rounding в настоящее время не стандартизован.

формат	M	обозн
Float32	23	E8M23
TF32	23	E8M10
BFloat16	7	E8M7
Float16	10	E5M10
BF8	2	E5M2
FP8	3	E4M3

- [2405.13938] eXmY: A Data Type and Technique for Arbitrary Bit Precision Quantization, 2024.

Правила преобразования для различных типов векторов можно выполнить через приведение к типу _Float32. Кроме того, необходимо определить правила округления при переходе между типами: RNE или RTZ. Заявлен режим округления: stocastic. По умолчанию используется RNE

(rounding to nearest even) для всех типов. В языке OpenCL C по-умолчанию используется RTZ (rounding to zero). Следует обратить внимание, что на некоторых платформах округление может работать как RN(x) = RTZ(x+0.5), round to nearest.

Для эмуляции stochastic rounding предлагается использовать алгоритмы повышения точности:

- FPQUANT Ранее мы рассматривали алгоритмы (Деккера и Кнута) умножения и сожения с отстаком, а также с использованием инструкции FMA.
- [15] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd ed., Society for In-dustrial and Applied Mathematics, Philadelphia, PA, 2002, https://doi.org/10.1137/1.9780898718027.

Важным для компенсированных алгоритмов суммирования является тот факт, что для чисел с плавающей точкой a и b, если $s = \mathrm{fl}(a + b)$ с округлением до ближайшего (RNE), то $t = a + b - s$ является числом с плавающей точкой, которое может быть вычислено следующим алгоритмом.

Алгоритм (FastTwoSum).
Даны числа с плавающей точкой a, b такие, что $|a| \geq |b|$, надо вычислить (с округлением до ближайшего) такие s и t, что $s = \mathrm{fl}(a + b)$ и $s + r = a + b$ точно. Все операции в числах сплавающей точкой:

1. $s \leftarrow a + b$
2. $t \leftarrow s - a$
3. $r \leftarrow b - t$

Величина r является ошибкой округления связанная с квантизацией. Ошибку можно рассматривать как квантовую, с нормальным распределением вероятности. На потоке данных вероятности суммируются. Например, при вычислении ембеддинг-векторов, ошибка (вектор ошибок) добавляется к вероятности на следующем цикле и компенсирует ошибку округления. Нейросети с архритекутурой трансформеры используют смешивание на выходе каждого слоя, которое может рассматриватся также в контексте управления ошибками.

- [EP4318229A1] Instructions to convert from fp16 to fp8

BF8 и HF8-based operations support round to nearest even (RNE) and stochastic rounding.

{Сослаться на схему CKKS}
{Разобрать вариант округления RNE для тернарных векторов}. Определить правило округления с компенсацией ошибки округления с использованием типа BFloat16 или Float16.

{Рассмотреть частный случай Float32} E8M23

{Рассмотреть частный случай BF16} E8M7

{Рассмотреть частный случай Float16} E5M10

{Рассмотреть частный случай BF8} E5M2

{Рассмотреть частный случай MXINT8}
Целый тип с общей экспонентой (E8M0) на 32 значения.

{Рассмотреть частный случай MXFP8}

{Алгоритм С.2 Поэлементное умножение полиномов}

Алгоритм С.6 Расширенное редуцирование

Разберем, как это соотносится с редукцией Барретта (см. Алг.А.1). В нашей версии алгоритма применются иные сдвиги выровненные на 32 бита, иначе считается константа обратной величины (к), что позволяет использовать операцию `mul_hi` .
$k = \lfloor (2^{64} - q)/q \rfloor$ подобрали выражение в таком виде.

1. function Barrett_Reduction(d, q, k, Q=32, L=64)
2. $c_1 \leftarrow d \gg Q$
3. $c_2 \leftarrow d + k \cdot c_1$
4. $c_3 \leftarrow c_2 \gg (L - Q)$
5. $c_4 \leftarrow d - q \cdot c_3$
6. if $c_4 \geq q$ then
7. $\quad c_4 \leftarrow c_4 - q$
8. end if
9. return c_4

Алгоритм использует не полное редуцирование и может потребоваться финальное редуцирование $c_4 < 2^Q$. L, Q выбираются кратно слову. При выполнении операции сложения переносы могут накапливаться, требуется завершающая операция $\mod q$.

/*! Редуцирование по модулю простого числа, с использованием Барретта
a - число для редуцирования

q - простое число модуль $q < \beta = 2^{32}$
U - константа Барретта $U = \lfloor (2^{64} - q\beta)/q \rfloor$
*/

Listing (Barrett reduction)

```
uint32_t MODB(uint64_t a, uint32_t q, uint32_t U) {
    uint64_t c2 = a + U*(a >>32);
    uint64_t c4 = a - q*(c2>>32);
    return  (c4>= q)? c4 - q: c4;
}
```

Алгоритм использует беззнаковые умножения 32 x 32 бит, с результатом 64 бит. Алгоритм можно обобщить на случай производльной разрядности (arbitrary precision). Интерес представляют простые числа с разрядностью 32 бита (вида $2^{32} - a_0 \cdot 2^{16} - 1$, используются в MWC32), вида $2^L - a_0 \cdot 2^s + 1$, $s \geq 9$ - подходят для Ring-LWE, и 23 бита ($q = 2^{23} - 2^{13} + 1$ - используется в стандарте FIPS.204).

Listing (Montgomery reduction). Алгоритм редуцирования Монтгомери после умножения 32 x 32 бит. Результат: $zR \mod q$, где $R = 2^{-32}$. Величина p вычисляется как $q^{-1} \mod 2^{32}$.

```
uint32_t  mont_MODM(uint64_t z, uint32_t q, uint64_t p){
    uint32_t m = z * p;         // low product z_0 (1/q)
    z = (z + m*(uint64_t)q)>>32;// high product
    if (z>=q) z-=q;
    return  z;
}
```

Listing (Signed Montgomery reduction). Алгоритм редуцирования Монтгомери после умножения со знаком 32 x 32 бит. $-q\beta/2 < z < q\beta/2$ Результат: $zR \mod q$, где $R = 2^{-32}$. Величина p вычисляется как $q^{-1} \mod 2^{32}$.

```
int32_t  signed_mont_MODM(int64_t z, int32_t q, int32_t p){
    int32_t m = z * p;          // low  product
    z = (z + (int64_t)m*q)>>32; // high product
    if (z<0) z+=q;
    return  (int32_t)z;
}
```

Алгоритм видится наиболее перспективным для реализации на GPU и для встроенных приложений на платформе ARM, где присутствуют операции low-product и high-product.

Далее мы рассмотрим операцию folding, как основой элемент для построения параллельного алгоритма вычислений хэша.

Алгоритм C.7 NTT-Friendly modular folding

Функция используется для параллельного редуцирования больших чисел.

1. function Folding(h, d, q, K)
2. $c_1 \leftarrow d + K \cdot h$ -- сдвиг на константу K и хеш-код h
3. return $c_1 \mod q$

Эту операцию можно эффективно векторизовать.
Расчет сдвиговой константы для операции folding выполняется с помощью алгоритма возведения в степень по модулю $K_\ell = 2^\ell \mod q$.

Максимальный период повтора последовательности при возведении в степень (порядок мультипликативной группы) должен составлять $a^{(q-1)/2} \equiv 1(\mod q)$. Проверка выполняется с помощью алгоритма возведения в степень по модулю $K_\ell = 2^\ell \mod q$ пока значение не обратится к единице. Именно такой критерий мы выбираем для генератора PRNG - максимальная длина последовательности.

Сдвиговая константа может рассматриваться как генератор мультипликативной группы.

Период мультипликативной группы. Если мы возводим число α в степень по модулю q, то существует число k (период) такое, что $\alpha^k \equiv 1(\mod q)$. Если $\alpha^{k/2} \equiv -1(\mod q)$ то k - период.

- Для простых чисел Ризеля можно получить период $q-1$, если α является квадратичным невычетом $\left(\frac{a}{q-1}\right) = -1$.
- Для модулей NTT-friendly $q = a\beta + 1$, 2 не является генератором, максимальный период повтора $(q-1)/w$ достигается, когда

$$a^{(q-1)/2w} = -1 \bmod q \quad \text{и} \quad a^{(q-1)/w} = 1 \bmod q$$

Это накладывает ограничение на максимальную длину неповторяющейся последовательности чисел при использовании в качестве генератора псведослучайных чисел.

Частный случай #1:
Пусть $K = a$, $q = a \cdot \beta - 1$, $\beta = 2^N$ - степень двойки, $a = (q+1)/\beta$.

Функция расчитывает $h_{i+1} \leftarrow d_i + h_i a \mod (a\beta - 1)$

1. $\{x_i, c_i\} \leftarrow h_i = x_i + c_i\beta$ -- выполнить разложение
2. $h_{i+1} \leftarrow x_i a + c_i + d_i$
3. if $h_{i+1} \geq q$ then
4. $\quad h_{i+1} \leftarrow h_{i+1} - q$
5. end if

Для вывода используется тождество $(x_i + c_i\beta)a \mod (a\beta - 1) \equiv x_i a + c_i$.
Если k - период мультиплекативной группы по модулю, то $a \equiv 2^{k-N} \mod (a\beta - 1)$. Таким образом функция является частным случаем Алг.С2 при $K_{-N} = 2^{k-N}$. Таким образом можно ввести отрицательные значения сдвига и считать сдвиг $K_{-N} = 2^{-N}$. Мы можем использовать функцию уполовинивания или удвоения по модулю, чтобы проверить этот вывод. Утверждение равносильно тождеству $a\beta \equiv 1 \mod (a\beta - 1)$, что означает a - обратное число для β по модулю $(a\beta - 1)$. Функция может работать как хэш-функция при $0 \leq d < \beta$ и как генератор псевдослучайных чисел, хэшем является младшая часть числа $x_i = h_i \mod \beta$.

```
/*! функция хеширования эквивалентна FOLD(K=A), q=(A<<16)-1 */
uint32_t mwc32_hash(uint32_t h, uint16_t d, uint32_t q, uint32_t a){
    h = (0xFFFF&h)*a + (h>>16) + d;
    if (h>= q) h-=q;
    return h;
}
```

{доказать что это работает без коррекции}

```
uint32_t mwc32_hash_u16(uint32_t h, uint16_t d, uint32_t a){
    h += d;
    h = (0xFFFF&h)*a + (h>>16);
    return h;
}
```

Для модулей $q = A2^8 - 1$:

```
uint32_t mwc32_hash_u8(uint32_t h, uint8_t d, uint32_t a){
    h += d;
    h = (0xFF&h)*a + (h>>8);
    return h;
}
```

Значения на выходе не превышают q.

Пример `s` -битного генератора случайных числел на модуле $q = A2^s - 1$, для `a = (q+1)>>s` , `mask =(1u<<s)-1` :

```
#define MASK ((1u<<s)-1)
uint32_t mwc32_next(uint32_t h, uint32_t a){
    h = (MASK & h)*a + (h>>s);
    return h;
}
```

Можно предложить хэш функцию, основанную на простых числах Прота $q = A2^{16} + 1$, где $q < 2^{31}$.

```
int32_t mwc32s_hash_16(int32_t h, int16_t d, int32_t a){
    h+= d;
    h = (h>>16) - (0xFFFF&h)*a;
    return h;
}
```

И генератор на простых числах $q = A2^s + 1$:

```
int32_t mwc32s_next(int32_t h, int32_t a){
    h =  (h>>s) - (MASK & h)*a;
    return h;
}
```

Частный случай #2:
Модуль вида $q = a \cdot \beta - 1$, где $\beta = 2^N$ - степень двойки.

для случая $a_1\beta_1 = a\beta$, $\beta_1 = 2^{N_1}$ - величина сдвига, $N_1 < N$ можно получить следующее тождество:

1. $k_1 \leftarrow \lfloor d/a_1 \rfloor$
2. $\{x_i, c_i\} \leftarrow \{d - k_1 \cdot a_1, k_1\}$
3. $(x_i + c_i a_1) \cdot \beta_1 \mod (a_1\beta_1 - 1) \equiv x_i\beta_1 + c_i$

Замена деления константу на умножение и сдвиг.
следует подобрать такое обратное число $\bar{a}$:
$k_1 = (\bar{a} \cdot d) \gg (w + n) \equiv \lfloor d/a_1 \rfloor$ для любых $d < 2^w$

Алгоритм вычисления сдвига с редуцированием основан на тождестве:
$(d - k_1 a_1)\beta_1 + k_1 \equiv d\beta_1 \mod (a\beta - 1)$

```
uint32_t shlm(uint32_t d, uint32_t a, uint32_t a_, int n, int n1){
    uint32_t k = ((uint64_t)a_*d)>>n;
    return k + (d-k*a)<<n1;
}
```

Аналогичный алгоритм вычисления сдвига с редуцированием для модулей $q = a\beta + 1$:
$(d - k_1 a_1)\beta_1 - k_1 \equiv d\beta_1 \mod (a\beta + 1)$

```
uint32_t shlm(uint32_t d, uint32_t a, uint32_t a_, int n, int n1){
    uint32_t k = ((uint64_t)a_*d)>>n;
    return (d-k*a)<<n1 - k;
}
```

Для вычисления сдвиговых констант требуется функция powm - возведение в степень по модулю, power-of-two degree.

Алгоритм возведения в степень по модулю $f(x) = x^a \mod q$, где $a = 2^r$ - степень, q - модуль.

```
uint64_t powm_64(const uint64_t x, uint64_t a, const uint64_t P)
{
        uint64_t r = x;
    while (a>>=1) {
                r = ((uint128_t)r*r)%P;
        }
        return r;
}
```

Для полной картины надо рассматривать изготовление генераторов PRNG на модулях вида:

- $q = A2^s + 1$ -- модули NTT-friendly, простые числа Прота

$$(x_i + c_i\beta)a \mod (a\beta + 1) \equiv ax_i - \mathrm{c}_i$$

- $q = A2^s - 1$ -- модули Mongomery-friendly, простые числа Ризеля для MWC

$$(x_i + c_i\beta)a \mod (a\beta - 1) \equiv ax_i + c_i$$

- $q = 2^n - K$ -- модули Pseudo-Mersenne, где $K^2 < 2^n$

$$(x_i + c_i\beta) \bmod (\beta - K) \equiv x_i + c_i K$$

Остальные тождества - частные случаи:

- $q = 2^n - 1$ -- модули Mersenne

$$(x_i + c_i\beta)a \bmod (a\beta - 1) \equiv c_i + x_i 2^{n-s}\ ,\ \text{где } \beta = 2^s, \alpha = 2^{n-s}$$

$$(x_i + c_i\beta) \bmod (\beta - 1) \equiv x_i + c_i$$

- $q = 2^n - 2^k - 1$ -- модули Generalized Mersenne, где $\beta = 2^w < 2^k$:

$$(x_i + c_i\beta)a \bmod (a\beta - 1) \equiv c_i + x_i 2^{n-w} + x_i 2^{k-w}$$

$$(x_i + c_i\beta) \bmod (\beta - 2^k - 1) \equiv x_i + c_i + c_i 2^k$$

- $q = 2^n - a_1 2^s + 1$ -- модули word-size NTT-friendly, где $\beta = 2^s$:

$$(x_i + c_i\beta)a \mod (a\beta + 1) \equiv c_i - x_i 2^{n-s} + a_1 x_i$$

$$(x_i + c_i\beta) \mod (\beta - a_1 2^s + 1) \equiv x_i + c_i a_1 2^s - c_i$$

- $q = 2^n - a_1 2^s - 1$ -- модули word-size MWC, $x = x_1\beta + x_0$, где $\beta = 2^s$:

$$(x_i + \mathrm{c}_i\beta)a \mod (a\beta - 1) \equiv c_i + x_i 2^{n-s} + a_1 x_i$$

Генераторы типа MWC для простых чисел Ризеля и Прота (NTT-friendly модули) для выходного потока с разрядностью s получены путем разложения $x = x_0 + x_1\beta$, причем

- $a = \beta^{-1} \mod q$ для модулей $q = a\beta - 1$, где $\beta = 2^s$;
- $a = -\beta^{-1} \mod q$ для модулей вида $q = a\beta + 1$, где $\beta = 2^s$.

Если длина данных превышает период повтора, следует использовать Redundancy RNS (Residue Number System), по нескольким взаимно простым модулям.

Алгоритм С.8 Алгоритм параллельного хэша

Пусть $x = x_0 + x_1\beta + x_2\beta^2 + ... + x_{n-1}\beta^{n-1}$, где $\beta = 2^s$, тогда вычисления могут производиться в группе используя операцию сжатия (folding) $r_k \leftarrow (x_{i,k} + r_k\beta^{-w}) \bmod (a\beta \pm 1)$, для $k = 1, ..., w$.

Алгоритм будет состоять из двух частей: параллельного использования операции свертки (folding) со смещением по w - ширине рабочей группы и финального редуцирования по рабочей

группе.

- $x\beta^{-n} \bmod (a\beta - 1) \equiv x_0 a^n + x_1 a^{n-1} + x_2 a^{n-2} + ... + x_{n-1}$; или
- $x\beta^{-n} \bmod (a\beta + 1) \equiv x_0 a^n - x_1 a^{n-1} + x_2 a^{n-2} - ... - x_{n-1}$.

При совместном использовании с NTT выполняется компрессия вектора коэффициентов Z_q^n по полиному $(X^N \pm 1)$. В этом случае входным вектором является вектор коэффициентов размера N. Функция компрессии может быть реализована с использованием быстрого преобразования Фурье (NTT).

Ниже мы предлагаем рассмотреть возможность построения кодов Рида-Соломона над полем $\mathbb{F}_q$, где q - простое число, и α — элемент поля $\mathbb{F}_q$, - примитивный корень степени n. Обобщенные коды Рида-Соломона используют преобразования подобные NTT, связанные с понятием матрицы MDS (максимальной различимой дистанцией) кодирования.

Коды Рида-Соломона

Пусть $\mathbb{F}_q$ — поле целых чисел по модулю простого числа q и α — примитивный элемент $\mathbb{F}_q$ - корень степени n.

Определение (Матрица Генераторов). Код Рида-Соломона над $\mathbb{F}_q$ размерности k, $\mathcal{RS}_q(\alpha; k)$, является линейным кодом $\mathbb{F}_q^n$ с матрицей генераторов

$$G = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{k-1} & \alpha^{(k-1)2} & \cdots & \alpha^{(k-1)(n-1)} \end{pmatrix}.$$

Линейный код $\mathcal{C}$ с параметрами $[n, k, d]$ над конечным полем $\mathbb{F}_q$ является k-мерным векторным подпространством $\mathbb{F}_q^n$ с минимальной дистанцией Хэмминга d.

Граница Синглтона (названная в честь Р. К. Синглтона) устанавливает предел мощности кода $\mathcal{C}$ с символами из поля $\mathbb{F}_q$ длины n и минимального расстояния Хэмминга $d \leq n - k + 1$. Если минимальное расстояние кода $\mathcal{C}$ достигает границы Синглетона, то $\mathcal{C}$ называется кодом с максимальным разделяемым расстоянием (MDS, сокр. Maximum Distance Separable). С момента зарождения теории кодирования MDS-коды привлекают большое внимание благодаря своей максимальной способности исправлять ошибки.

Определение (Проверочная матрица). Коды Рида-Соломона из матриц проверки чётности.

Рассмотрим матрицу

$$H = \begin{pmatrix} 1 & \alpha & \alpha^2 & \cdots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{n-k} & \alpha^{(n-k)2} & \cdots & \alpha^{(n-k)(n-1)} \end{pmatrix}.$$

Это матрица максимального ранга (а именно $n-k$) благодаря структуре Вандермонда. Кроме того, произведение матрицы G и транспонированной матрицы H равно нулевой матрице. Заметим, что произведение i-й строки матрицы G ($1 \leq i \leq k$) на j-ю строку матрицы H ($1 \leq j \leq n-k$) равно $\sum_{r=1}^{n} \alpha^{(i-1)(r-1)}\alpha^{j(r-1)} = \sum_{r=1}^{n} \alpha^{(i+j-1)(r-1)}$. Поскольку пределы i и j таковы, что $i+j-1 < q-1$ и, следовательно, $\alpha^{i+j-1} \neq 1$. Наконец, сумма равна $\frac{(\alpha^{i+j-1})^{n-1}-1}{\alpha^{i+j-1}-1} = 0$.

Это позволяет дать следующее эквивалентное определение.

Определение. Код Рида-Соломона над $\mathbb{F}_q$ размерности k, $\mathcal{RS}_q(\alpha, k)$, является линейным кодом $\mathbb{F}_q^n$ с матрицей проверки чётности, равной H.

Существует явная связь между кодами Рида-Соломона, MDS-матрицами, матрицей Вандермонда и интерполяционными полиномами Лагранжа. Коды Рида-Соломона являются MDS-кодами и допускают кодирование и декодирование с использованием быстрого преобразования Фурье (FFT) над конечным полем $\mathbb{F}_q$.

Теорема. Код Рида-Соломона $\mathcal{RS}_q(\alpha, k)$ имеет минимальное расстояние $d = n-k+1$ и, следовательно, является MDS-кодом.

Коды Рида-Соломона и интерполяционные полиномы

Рассмотрим множество $\mathbb{F}_q[x]_{<k}$ всех многочленов с коэффициентами в $\mathbb{F}_q$ и степенью строго меньше k. Любой полином $f(x) \in \mathbb{F}_q[x]_{<k}$ имеет вид $f(x) = a_0 + a_1 x + \cdots + a_{k-1}x^{k-1}$ с коэффициентами $a_i \in \mathbb{F}_q$. Заметим, что вычисление $f(\alpha^i) = a_0 + a_1\alpha^i + \cdots + a_{k-1}\alpha^{i(k-1)}$, что равно результату произведения вектора $(a_0, \ldots, a_{k-1})$ на $i+1$-й столбец матрицы G. Таким образом, произведение вектора $(a_0, \ldots, a_{k-1})$ на матрицу G равно вектору $(f(1), f(\alpha), f(\alpha^2), \ldots, f(\alpha^{n-1}))$.

Определение. Коды Рида-Соломона над $\mathbb{F}_q$ и размерности k, $RS_q(\alpha; k)$, есть множество $\{(f(1), f(\alpha), f(\alpha^2), \ldots, f(\alpha^{n-1})) : f \in \mathbb{F}_q[x]_{<k}\}$.

- (https://users.math.msu.edu/users/halljo/classes/codenotes/grs.pdf) -- связь с полиномами Лагранжа

Пусть $a_i \in A \subset \mathbb{F}_q$ -- множество корней полинорма, тогда

$$L(x) = \prod_{i=1}^{n}(x - \alpha_i)$$

и

$$L_i(x) = L(x)/(x - \alpha_i) = \prod_{j \neq i}(x - \alpha_j).$$

Многочлены $L(x)$ и $L_i(x)$ являются полиномами степеней n и $n-1$ соответственно. Вектор f имеет i-ю координату $v_i f(\alpha_i)$.
Используя формулу интерполяции Лагранжа, получим

$$f(x) = \sum_{i=1}^{n} \frac{L_i(x)}{L_i(\alpha_i)} f(\alpha_i).$$

Число-Теоретическое преобразование (NTT) на основе ω

Определение. Число-Теоретическое преобразование (NTT) для вектора коэффициентов полинома a определяется как $\tilde{a} = NTT_{\omega}^{n}(a)$, где:

$$\tilde{a}_j = \sum_{i=0}^{n-1} \omega^{ij} a_i \bmod q$$

для $j = 0, 1, \ldots, n-1$. NTT имеет структуру матрицы генераторов кода RS.

Обобщенные Коды Рида-Соломона

Пусть $\mathbb{F}_q$ — конечное поле, выберем ненулевые элементы $\{v_1, \ldots, v_n\} \in \mathbb{F}_q^*$ и различные элементы $\{\alpha_1, \ldots, \alpha_n\} \in \mathbb{F}_q$. Определим вектора $\mathbf{v} = (v_1, \ldots, v_n)$ и $\alpha = (\alpha_1, \ldots, \alpha_n)$. Для $0 \leq k \leq n$ определим *обобщенные коды Рида-Соломона*

$$GRS_{n,k}(\alpha, \mathbf{v}) = \{(v_1 f(\alpha_1), v_2 f(\alpha_2), \ldots, v_n f(\alpha_n)) \mid f(x) \in F[x]_k\}.$$

Здесь $F[x]_k$ обозначает множество полиномов в $F[x]$ степени менее k, векторное пространство размерности k над $\mathbb{F}$. Для фиксированных n, α и $\mathbf{v}$, различные GRS-коды обладают свойством вложенности $GRS_{n,k-1}(\alpha, \mathbf{v}) \subseteq GRS_{n,k}(\alpha, \mathbf{v})$.

Если $f(x)$ — полином, то мы будем писать f для его ассоциированного кодового слова. Это кодовое слово также зависит от α и $\mathbf{v}$; поэтому иногда мы предпочитаем записывать его однозначно

$$ev_{\alpha,\mathbf{v}}(f(x)) = (v_1 f(\alpha_1), v_2 f(\alpha_2), \ldots, v_n f(\alpha_n)),$$

указывая, что кодовое слово $f = ev_{\alpha,\mathbf{v}}(f(x))$ возникает при вычисления полинома $f(x)$ из векторов α и $\mathbf{v}$.

Найдем генераторную матрицу для $GRS_{n,k}(\alpha, v)$. Любая базисная система $f_1(x), \ldots, f_k(x)$ из $F[x]_k$ порождает базис кода $f_1, \ldots, f_k$. Канонической полиномиальной базисной системой является набор $\{1, x, \ldots, x^{k-1}\}$, степенной базис. Соответствующая генераторная матрица, чьи строки имеют вид

$$ev_{\alpha,v}(x^i) = (v_1\alpha_1^i, \ldots, v_j\alpha_j^i, \ldots, v_n\alpha_n^i), \text{ для } i = 0, \ldots, k-1;$$

является *канонической генераторной матрицей* для $GRS_{n,k}(\alpha, v)$:

$$\begin{bmatrix} v_1 & v_2 & \cdots & v_j & \cdots & v_n \\ v_1\alpha_1 & v_2\alpha_2 & \cdots & v_j\alpha_j & \cdots & v_n\alpha_n \\ \vdots & \ddots & \vdots & \ddots & \vdots & \\ v_1\alpha_1^i & v_2\alpha_2^i & \cdots & v_j\alpha_j^i & \cdots & v_n\alpha_n^i \\ \vdots & \ddots & \vdots & \ddots & \vdots & \\ v_1\alpha_1^{k-1} & v_2\alpha_2^{k-1} & \cdots & v_j\alpha_j^{k-1} & \cdots & v_n\alpha_n^{k-1} \end{bmatrix}$$

Пусть $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$, где α_i — различные элементы поля $\mathbb{F}_q$, $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})$ — ненулевые элементы $\mathbb{F}_q^*$. Тогда обобщённый код Рида-Соломона $\mathcal{GRS}_k(\alpha, y)$, состоит из всех кодовых векторов вида:

$$u = (v_0 f(\alpha_0), v_1 f(\alpha_1), \ldots, v_{n-1} f(\alpha_{n-1})),$$

где $f(x)$ — информационные многочлены над полем $\mathbb{F}_q$ степени не выше $k-1$. Кодовое расстояние кода $\mathcal{GRS}_k(\alpha, v)$ равно $n-k+1$. Если вектор $\mathbf{v}$ состоит из единиц и α — примитивный корень степени k поля $\mathbb{F}_q$, то в этом случае получаем код Рида — Соломона ($\mathcal{RS}_k(\alpha)$).

Теорема #1. Код, дуальный $GRS_k(\alpha, v)$-коду, является $GRS_{n-k}(\alpha, w)$-кодом для некоторого вектора w, причём в качестве вектора w можно взять $w = (w_0, w_1, \ldots, w_{n-1})$, где

$$w_i = \frac{1}{v_i \prod_{j \neq i}(\alpha_i - \alpha_j)}, \quad i = 0, 1, \ldots, n-1.$$

Из данной теоремы следует, что проверочная матрица H кода $GRS_k(\alpha, v)$ равна порождающей матрице кода $GRS_{n-k}(\alpha, w)$:

$$H = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{n-k-1} & \alpha_1^{n-k-1} & \cdots & \alpha_{n-1}^{n-k-1} \end{pmatrix} \begin{pmatrix} w_0 & 0 & \cdots & 0 \\ 0 & w_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w_{n-1} \end{pmatrix}.$$

Матрица H содержит $n-k = d-1$ строк и n столбцов.

Заметим, Генераторная матрица может быть определена на любом базисе полиномов, включая полиномы Лагранжа, Тейлора, Эрмита и Бернштейна. Между базисами полиномов существует изоморфное (обратимое) преобразование предаставленное матрицей $n \times n$. Это в теории. Интересно было бы посмотерть, как это утверждение реализуется на практике.

Алгебраические свойства обобщенных кодов Рида-Соломона

- $GRS_{n,k}(\alpha, \mathbf{v})^{\perp} = GRS_{n,n-k}(\alpha, \mathbf{u})$, при $u_i = v_i^{-1} L_i(\alpha_i)^{-1}$, где $L_i(\alpha_i) = \prod_{j \neq i}(\alpha_i - \alpha_j)$.
- Рассмотрим код $GRS_{n,k}(\alpha, \mathbf{v})$ над $\mathbb{F}_q$. Пусть a, c — ненулевые элементы $\mathbb{F}_q$, а $\mathbf{b}$ — вектор из $\mathbb{F}_q^n$, все элементы которого равны $b \in \mathbb{F}_q$.

$$GRS_{n,k}(\alpha, \mathbf{v}) = GRS_{n,k}(a\alpha + \mathbf{b}, c\mathbf{v}).$$

- Если $n < |\mathbb{F}_q|$, существует иной набор α' без элементов, равных 0, такой что

$$GRS_{n,k}(\alpha, \mathbf{v}) = GRS_{n,k}(\alpha', \mathbf{v}).$$

{правильно было бы сказать, что существует множество корней из единицы.}

Матрицы Вандермонда. Хотя матрицы Вандермонда можно определить над любым полем, для наших целей мы сужаем рассмотрение на конечных полях. Даны $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{F}_q$, матрица Вандермонда от $\alpha_1, \ldots, \alpha_n$ порядка r определяется как

$$V_r(\alpha_1, \alpha_2, \ldots, \alpha_n) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{r-1} & \alpha_2^{r-1} & \cdots & \alpha_n^{r-1} \end{pmatrix}.$$

Можно доказать, что определитель $V_r(\alpha_1, \alpha_2, \ldots, \alpha_n)$ удовлетворяет $|V(\alpha_1, \alpha_2, \ldots, \alpha_n)| = \prod_{i<j}(\alpha_j - \alpha_i)$. Следовательно, $V_n(\alpha_1, \alpha_2, \ldots, \alpha_n)$ имеет обратную матрицу тогда и только тогда $\alpha_i \neq \alpha_j$ для всех $1 \leq i < j \leq n$.

Определение(Дуальный код).
Скалярное произведение $\langle \cdot, \cdot \rangle$ в Евклидовом пространстве $\mathbb{F}_q^n$ определяется как

$$\langle u, v \rangle = \sum_{i=1}^{n} u_i v_i,$$

где $u = (u_1, \ldots, u_n) \in \mathbb{F}_q^n$ и $v = (v_1, \ldots, v_n) \in \mathbb{F}_q^n$. Для любого $[n, k, d]_q$-линейного кода $\mathcal{C}$ дуальный код $\mathcal{C}^\perp$ определяется через скалярное произведение, как

$$\mathcal{C}^\perp := \{u \in \mathbb{F}_q^n : \langle u, c \rangle = 0, \text{ для любого } c \in \mathcal{C}\}.$$

Класс самодуальных MDS кодов определяется как множество линейных кодов C над конечным полем $\mathbb{F}_q$, где код C равен своему дуальному коду $C^\perp$, то есть $C = C^\perp$. Это означает, что для любого вектора $u \in C$ и любого вектора $v \in C$ скалярное произведение $u \cdot v = 0$ в $\mathbb{F}_q$.

- [Ratseev] Ratseev S.M., Cherevatenko O.I. Decoding algorithms for generalized Reed-Solomon codes

-- В статье приводятся алгоритмы декодирования обобщенных кодов Рида — Соломона на случай
ошибок и стираний. Данные алгоритмы строятся на основе алгоритма Гао, алгоритма Сугиямы, алгоритма Берлекэмпа–Месси (алгоритма Питерсона — Горенстейна — Цирлера).

- [1706.03504] A Decoding Approach to Reed–Solomon Codes from Their Definition

Вывод по разделу
При использовании вектора $(1, \alpha, \alpha^2, ...\alpha^{n-1})$, где α - корень степени n в поле $\mathbb{F}_q$, можно построить матрицу Генераторов G кода Рида-Соломона. Матрица генераторов имеет структуру матрицы FFT и позволяет использовать NTT_α^n преобразование с параметром α. Структура кольца полиномов $\mathbb{F}_q[x]/\langle x^n + 1 \rangle$ соответствует обобщенным кодам Рида-Соломона. Существует изоморфизм меду кодами построенными на различных базисах, также должен существовать изоморфиз между кольцами полиномов одного порядка. В частности изоморфиз между $\langle x^n - 1 \rangle$ и $\langle x^n + 1 \rangle$ может быть представлен диагональной матрицей $diag(1, \psi, ..., \psi^{n-1})$ из степеней корня $2n$-ой степени из единицы. Для GRS-кодов, всегда существует дуальное представление состоящее из ортогональных дополнений, которое представлено проверочной матрицей со структурой Вандермонда.

Таким образом мы рассмотрели некоторые базовые принципы построения вектора некриптографического хэша на модульной арифметике с возможностью восстановления данных

с ошибками и стиранием. Класс MDS кодов, Обобщенные коды Рида-Соломона, которые могут применяться, как функция сжатия $\mathbb{F}_q^N \mapsto \mathbb{F}_q^k$.

Косо симметричные матрицы skew symmertic

Хотим рассматреть способ генерации MDS матриц с использованием матриц Вандермонда и определения Эрмитового скалярного произведения. Матрицы генерации обладают свойством симметрии ($M = M^T$). Другой класс матриц - кососимметричные матрицы, ($M = -M^T$). Вместе они встречаются в физике и математике комплексных чисел. Известен класс Гамильтоновых матриц, которые могут применяться при распознавании физики.

- [2508.09687]

Пусть $x = (x_1, x_2, \ldots, x_n), y = (y_1, y_2, \ldots, y_n) \in \mathbb{F}_q^n$, здесь мы напоминаем, что евклидово скалярное произведение векторов x и y равно

$$\langle x, y \rangle_E = \sum_{i=1}^{n} x_i y_i.$$

дуальный код для C из Евклидова скалярного произведения определяется как

$$C^{\perp_E} = \{x \mid x \in \mathbb{F}_q^n, \langle x, y \rangle_E = 0, \text{ для всех } y \in C\}.$$

Всегда полезно рассмотреть другое скалярное произведение, называемое Эрмитовым скалярным произведением.

$$\langle x, y \rangle_H = \sum_{i=1}^{n} x_i \bar{y}_i.$$

Аналогично, мы можем определить дуальный код для C из Эрмитова скалярного произведения следующим образом

$$C^{\perp_H} = \{x \mid x \in \mathbb{F}_{q^2}^n, \langle x, y \rangle_H = 0, \text{ для всех } y \in C\}.$$

Аналогично, вводится ермитова- само-ортогональность и ермитова- само-дуальность. Если $C \subseteq C^{\perp_H}$, то $C^{\perp_H}$ является ермитово- само-ортогональным. В частности, если $C^{\perp_H} = C$, то C является ермитово-само-дуальным.

Все звучит прекрасно, но в определении не используется конкретное определение сопряженных пар.
Определить сопряженные пары можно через мнимую единицу или через правило выполния

операции умножения сопряженных пар. Что является аналогом мнимой единицы в модульной арифметике?

В модульной арифметике аналог мнимой единицы j (где $j^2 = -1 \bmod q$) зависит от модуля. Если модуль q — простое число, то мнимая единица существует только в расширенных полях конечных элементов $\mathbb{F}_{p^k}$, где k — минимальная степень, при которой полином $x^2 + 1$ имеет корень.

В частности, в кольце полиномов $\mathbb{F}_q[x]/\langle x^N + 1\rangle$, можно представить разложение по модулю $x^N + 1 = \prod_{i=0}^{N/2}(X^2 + \xi_i)$. Таким образом вектор разбивается на пары и каждая пара имеет свою единицу ξ_i, для которой можно определить правило умножения сопряженных пар, без вычисления мнимой единицы, которого может и не существовать на множестве $\mathbb{F}_q$.

$$r_0 = x_0 y_0 + x_1 y_1 \xi_i$$
$$r_1 = x_0 y_1 + x_1 y_0$$

Еще можно не найти в конечном поле числа бесконечность. Хотя после того как мы определили споряженные пары напрашиваются конформные отображения в расширенной плоскости. Как расширить множество на бесконечность в модульной арифметике?

В модульной арифметике, которая работает в конечных множествах (например, $\mathbb{Z}/p\mathbb{Z}$), понятие бесконечности не имеет прямого смысла, так как все операции ограничены модулем p. Однако можно интерпретировать "бесконечность" в контексте расширенных структур, таких как проективные координаты или формальные дополнения. Так, например, вводится понятие *расширенного* обобщенного кода Рида-Соломона.

Пусть q — простое число, а $\mathbb{F}_q$ — конечное поле с q элементами. Пусть n и k — два целых числа такие, что $2 \le n \le q+1$ и $2 \le k \le n$. Запишем множество чисел $\{a_1, \ldots, a_q\} \in \mathbb{F}_q$ и пусть ∞ будет точкой на бесконечности. Для любого $\alpha \in \mathbb{F}_q \cup \{\infty\}$, пусть $c_k(\alpha)$ — вектор-столбец из $\mathbb{F}_q^k$, заданный как

$$c_k(\alpha) \triangleq \begin{cases} (1, \alpha, \alpha^2, \ldots, \alpha^{k-1})^T \in \mathbb{F}_q^k & \text{если } \alpha \in \mathbb{F}_q; \\ (0, 0, \ldots, 0, 1)^T \in \mathbb{F}_q^k & \text{если } \alpha = \infty. \end{cases}$$

Определение. Пусть $A = \{a_1, a_2, \ldots, a_n\}$ — подмножество из $\mathbb{F}_q \cup \infty$ и $v = (v_1, v_2, \ldots, v_n)$ — вектор в $(\mathbb{F}_q^*)^n$. Для любого $2 \le k \le n$, если $\infty \in A$, то линейный код с генераторной матрицей $[v_1 c_k(a_1), v_2 c_k(a_2), \ldots, v_n c_k(a_n)] = [c_k(a_1), c_k(a_2), \ldots, c_k(a_n)] \cdot \operatorname{diag}\{v_1, v_2, \ldots, v_n\}$ называется *расширенным* обобщённым кодом Рида-Соломона.

2-to-1 Compression function

Следует определить правила общие для криптографической функции сжатия и некриптографической

$$\mathbb{F}_q^N \mapsto \mathbb{F}_q^{N/2} : \mathrm{C}(v) = Tr(\mathcal{P}(v) + v) \in \mathbb{F}_q^{N/2}$$

где $\mathcal{P}(v)$ - функция пермутации, $Tr(v)$ - обрезает вектор до $N/2$ значений. Криптографической функция является если использовать функцию пермутации, устойчивую к коллизиям.
Умножение на MDS матрицу M - линейная (обратимая) операция не влияет на криптографические свойства.

Для некриптографического хэша рассматриваем следующее представление:

- $\mathcal{C}(X) := MX_{k-1} + (I - M)X_k$
- $\mathcal{C}(X) := X_k + MX_{k-1}$

Алгоритм NTT необходимо доработать, чтобы он на выходе давал вектор заданной длины k.

- $MDS_{\alpha}^{k}(M) : \{\alpha_0, \alpha_1, ..., \alpha_{n-1}\}$ - вектор коэффициентов.
- Результат расчитывается как произведение матрицы Вандермонда на диагональную матрицу

$$G = V_r(\alpha_0, \alpha_1, ..., \alpha_{n-1})^T \cdot diag(w_0, w_1, ..., w_{n-1})$$

причем элементы α_i составляют мультипликативную группу $\mathbb{F}_q^*$.

В работе предложен способ генерации MDS матриц с использованием матриц Вандермонда.

Для криптографической функции сжатия помимо линенйного преобразования следует выполнять нелинейные. Нелинейные преобразования представлены тремя вариантами: возведение в степень, перестановка бит или сдвиги и перестановки слов в векторе.

Пример нелинейной функции: $\mathcal{SB}ox_r(X) := (X + C_r)^e$, где $X \in \mathbb{F}_q^N$.

Мы рассматриваем применение функции смешивания $\mathcal{P}(v)$, такие как Poseidon2, Monolith, Rescue-Prime и конечно KECCAK в режиме XOF.

Monolith 2-to-1 Compression function:

- $\mathcal{C}oncrete(X) := \tilde{M} \odot \tilde{X}$ -- Linear Layer MDS.
- $\mathcal{B}ricks(X) := (x_0 - x_{n-1}^2, x_1 + x_0^2, ..., x_{n-1} + x_{n-2}^2)$ -- Diffusion Layer.
- $\mathcal{B}ars(x) := (x \oplus (\bar{x} \lll 1) \odot (x \lll 2) \odot (x \lll 3)) \lll 1$ -- Nonlinear Layer, $\mathbb{F}_2^8$.
- $\mathcal{R}_i(X) := \mathrm{C}_i + Concrete \circ Bricks \circ Bars(X)$ -- Round Function.
- $\mathcal{P}(X) := \mathcal{R}_n \circ ... \circ \mathcal{R}_1 \circ \mathcal{C}oncrete(X)$ -- Permutation Function.

Чего надо обеспечить:

- Уменьшить число раундов преобразований при сохранении уровня безопасности.
- Линейная операция $M \times (X + xX \odot X) + C_i$ может выражаться через умножение на матрицу MDS
- C_r -- константы выбираются $< 2^s$, чтобы снизить число операций редуцирования по модулю $q = a2^s \pm 1$.

Референсная реализация Алгоритмов NTT

Референсная реализация нужна для отладки параллельных алгоритмов. Отладка строится на сравнении результатов нескольких реализаций алгоритмов редуцирования и умножения полиномов. Для записи алгоритмов используется ряд операций модульной арифметики. В нашей реализации упор сделан на поддержку модулей степени 2^{32} и 2^{31} вида $q = a\beta \pm 1$.

Алгоритм поиска корня степени N из единицы по модулю простого числа $q = a\beta + 1$, где $\beta = 2^s$, N - степень двойки $N|(q-1)$.

```
uint32_t ntt_root(uint32_t N, uint32_t q){
    uint32_t gen =  3;// выбор генератора - квадратичный не-вычет
    while (jacobi(gen,q)!=-1 || POWM(gen, (q-1)/2, q)==1) gen++;
    return POWM(gen, (q-1)/N, q);
}
```

Реализация алгоритма для нахождения символа Якоби (Лежандра) представлена в проекте. Символ Якоби определяет возможность извлечения квадратного корня по модулю. В качестве генератора мультипликативной группы используется квадратичный не-вычет.

Listing (Polynomial multiplication). Референсный алгоритм для умножения полиномов без использования NTT

```
void poly_mul(uint32_t *r, const uint32_t *a, const uint32_t *b, unsigned int N, uint32_t q)
{
    poly_mulm_u(r, a, b[N-1], N, q);
    for (int i = N-2; i>=0; i--)
        poly_xtime_madd(r, a, b[i], N, q);
}
```

Функция `poly_xtime_madd()` выполняет операцию ротации полинома r и умножения на скаляр с накоплением, $r(x) = r(x) \cdot x + a(x) \cdot b_i \pmod{x^N + 1}$.

Listing (NTT). Алгоритм вычисления NTT (чисто-теоретического преобразования) на кольце полиномов
$\mathbb{Z}_q[x]/(x^N+1)$.

```
uint32_t* NTT(uint32_t *a, const uint32_t *gamma, unsigned int N, uint32_t q){
    unsigned int i, j, k, m, n;
    n = N/2;
    for (m = 1; m < N; m = 2*m, n = n/2) {
        for (i=0, k=0; i<m; i++, k+=2*n) {
            uint32_t w = gamma[m+i];
            NTT_CT_butterfly(a+k, a+k+n, w, n, q);
        }
    }
    return a;
}
```

Алгоритм получился уникально простым, функция NTT_CT_butterfly выполняет операцию "бабочка" над вектором длины n.

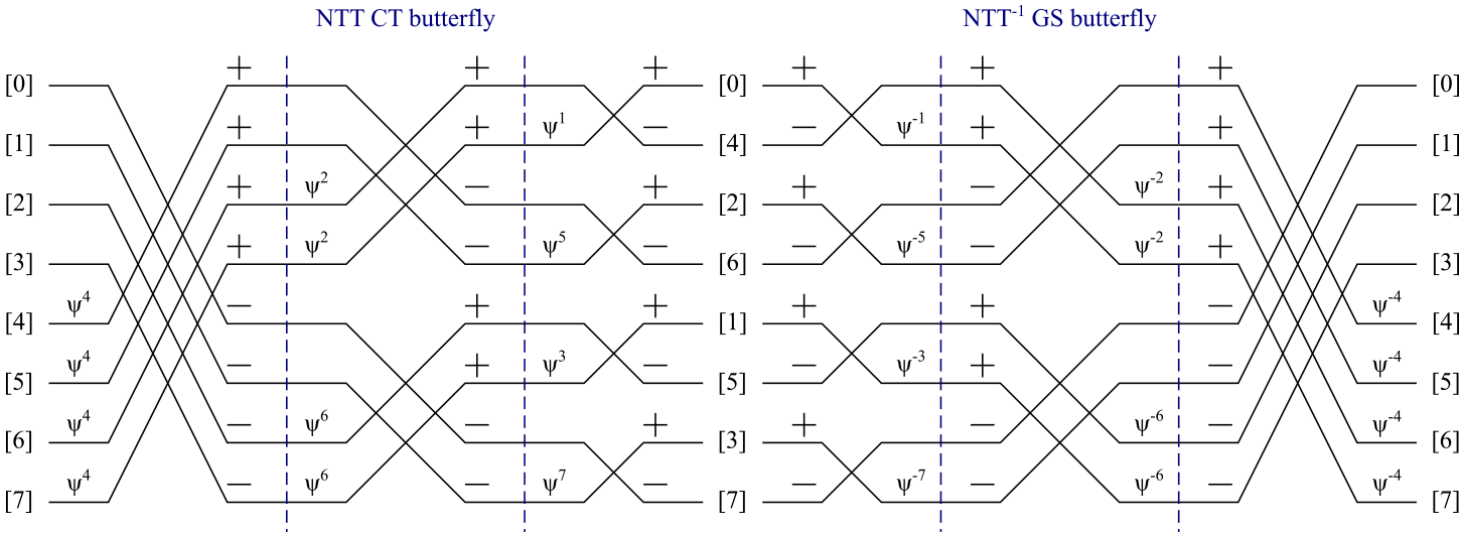

Схема расчета бабочек NTT CT butterfly размером 8 элементов представлена графчиески. Видно, что при понижении степени разложения $(m = 2^s)$ меняется длина вектора и удваиается число бабочек в цикле. Для векторной реализации и реализации на GPU можно предложить масштабировать бабочки до достижения заданного размера N=16, после чего применять оптимизированный вариант расчета на регистрах AVX512. Фактически это разбивает алгоритм на две части, первая считает векторные бабочки, блоками по 16, а вторая считает NTT фиксированного размера (завершающие 4-е стадии разложения) на регистрах AVX512 или в варп-группе на GPU.

Еще одно важное свойство - приодичность степени корней из единицы $2N$: $\psi^{-k} = -\psi^{N-k}$, может быть использовано при вычислении обратных степеней в алгоритме NTT GS butterfly.

- [2024/585] A Complete Beginner Guide to the Number Theoretic Transform (NTT)

-- Идея графического представления вычислительной сети из бабочек представлена в работе, но с некоторыми неточностями в знаках. В своей работе мы использовали графическое представление для отладки векторных бабочек. Нумерация индексов элементов на выходе алгоритма имеет обратный отраженный порядок бит (bit-reverse order). Нумерация индексов в таблице корней из единицы также дается в отраженном битовом порядке. Номер индекса в таблице (степеней корня) на стадии разложения s, $m = 2^s$, высчитывается как $\mathrm{RevBits}_N(2^s + i)$.

Listing (NTT CT butterfly). Алгоритм вычисления бабочки для прямого преобразования NTT. Алгоритм использует векторные операции модульнй арифметики: умножение на скаляр по модулю, сложение по модулю, вычитание по модулю.

```
void NTT_CT_butterfly(uint32_t *a, uint32_t *b, uint32_t g, unsigned int n, uint32_t q) {
    uint32_t w[n];
    vec_mulm_u(w, b, g, n, q);
    vec_subm(b, a, w, n, q);
    vec_addm(a, a, w, n, q);
}
```

Listing (NTT GS butterfly). Алгоритм вычисления бабочки для обратного преобразования NTT.

```
void NTT_GS_butterfly(uint32_t *a, uint32_t *b, uint32_t g, unsigned int n, uint32_t q) {
    uint32_t w[n];
    vec_subm(w, a, b, n, q);
    vec_addm(a, a, b, n, q);
    vec_mulm_u(b, w, g, n, q);
}
```

Listing (NTT polynomial multiplicaton). Алгоритма умножения полиномов в кольце $\mathbb{Z}_q[x]/(x^N + 1)$ с использованием NTT

```
void NTT_mul(uint32_t *r, uint32_t *a, uint32_t *b,
        uint32_t *gamma, uint32_t *r_gamma, unsigned int N, uint32_t q)
{
    NTT(a, gamma, N, q);
    NTT(b, gamma, N, q);
    vec_mulm(r, a, b, N, q);
    invNTT(r, r_gamma, N, q);
}
```

Для реализации умножения на предварительно рассчитанные контанты корня из единицы оптимально использвовать метод Shoup Modular Multiplication.

Listing (Shoup Modular Multiplication). Предварительное вычисление констант выполняется с использованием округления результата деления к ближайшему целому методом RNE.

```
//    uint64_t w = (double)(b<<32)/p;
uint32_t shoup_MULM(uint32_t a, uint32_t b, uint64_t w,  uint32_t p){
    uint64_t q = (a*(uint64_t)w)>>32;
    uint64_t r = a*(uint64_t)b - q*p;
    return  (r-p< r)? r - p: r;// min(r, r-p)
}
```

Listing (Barrett Modular Reduction P=3329). Используется в реализации ML-KEM. Для модуля $P = 13 \cdot 2^8 + 1 = 3329$, выбраны параметры Q=11, L=27, U=40317
$Ur = \lfloor 2^L / P \rfloor$ (L=Q+16, Q=11, L=27, Ur=40317)

```
uint32_t MODB(uint32_t a, uint32_t q, uint16_t Ur) {
const int Q = 11;
const int L = Q+16;
    uint32_t c2 = Ur*(a>>Q);// mul hi
     int16_t c3 = q*(c2>>(L-Q));
     int16_t c4 = a - c3;
    if (c4>=q) c4 -= q;
    return c4;
}
```

Коэффициенты L,Q подобраны таким образом, чтобы редуцирование работало для входных значений чисел до 2^32, или точнее до 0xFFFF0000 -- максимальное число при вычислениях, и L-Q = 16 (чтобы использовать операцию mulhi) и полученное значение константы U должно укладываться в 16 бит. Ориентировочно выбор Q может быть равен числу бит простого числа

минус один. Для чисел 32 бит, доказательство выполняется методом перебора всех возможных значений.

Результаты работы (в процессе)

Можно сказать, что мы живем в золотой век, когда основные алгоритмы уже написаны и реализованы. Может возникнуть критика, зачем вообще нужна собственная реализаия, почему не воспользоваться готовой библиотекой. На самом деле я пишу свою реализацию, чтобы разобраться в предмете и получить инструменты для отладки оптимизированных версий алгоритмов. Помимо этого в современных реалиях, только библиотеки интегрированные в дистрибутив GNU/Linux и переносимые между платформами могут использоваться для разработки собственных продуктов. Наша цель- переносимость в серверные и во встроенные приложения, достижение высокой производительности на аппаратной платформе. Современная тенденция связанная с развитием нейросетей - достижение независимости в коде от сторонних библиотек.

Ответ на основные вопросы: возможно ли разработать на пост-квантовой криптографии блокчейн с доказательсвтвом нулевого разглашения (ZKP), доказательством использования сложной функции, и гомоморфным шифрованием (FHE)?
-- Да, возможно.
Возможно ли совместить добычу криптовалюты и подписть результата работы сети на основе XMSS (цифровой подсписи на основе хеш-цепочек и дерева Меркла).
-- Да, возможно.
Возможно ли использовать принципы ZKP, FHE и PQC в системах машинного обучения?
-- Да, возможно.
Возможно ли использовать протокол удаленного запуска процедур для распределенных вычислений (MPC) онованный на принципах ZKP, и блокчейне с доказательством нулевого разглашения (ZKP).
-- Да, возможно.

Алгоритмы и численные методы:

- Разработаны методы модульной арифметики для модулей вида $p = 2^n - a_1 \cdot 2^s \pm 1$.
- Разработаны методы редуцирования для NTT-frienly модулей и MWC.
- Разработаны методы модульной арифметики со знаком
- Разработаны методы MWC-хэширования для быстрого поиска с использованием хэш таблиц.
- Разработаны векторные реализации прямого и обратного NTT
- Разработаны методы для RNS представления больших чисел и переключения модулей

- Разработаны методы поиска простых чисел NTT-Friendy, методы вычисления корней n-й степени.

Строительными блоками являются: Ring-LWE, PQC, XMSS, FHE, ZKP, RPC, protobuf. Важным показателем применения является возможность расчета криптографических функций на GPU и в векторных инструкциях, а также возможность арифметизации операций и представления в полиномиальной форме.

- ml_kem.c -- выработка общего ключа шифрования, реализация стандарта PQC криптографии NIST:FIPS.203.
  - векторная реализация NTT для кольца полиномов с модулем 3329 с использованием наборов инструкций AVX512 и AVX2, AVX10.1. Алгоритмы NTT требуют использования многоступенчатых бабочек, в векторной реализации используется 32 элементные пермутации и сложный подсчет степеней корней из единицы при перестановках элементов вектора.
  - векторная реализация для Aarch64 с использованием набора инструкций NEON.
- (shake256.c) -- функция хеширования SHAKE-256 и SHA3 - реализация с использованием векторных инструкций AVX512, с использованием транспонирования и тернарных операций (логических функций трех переменных). Алгоритм описан средствами векторного расширения языка C и переносим на другие аппаратные платформы.
  - Реализован метод синтеза алгоритма и констант для версии KECCAK-800 (32 бита).
  - Реализован метод XOF (Extendable Output Function), cSHAKE (Кастомизированная версия)
  - KMAC (Keyed Message Authentication Code).
- (poseidon2.c) -- функция хеширования Poseidon-2π - реализация пермутации (умножение на матрицы MDS и Sbox) с использованием векторных инструкций AVX512, параметризация под выбранный модуль и циклический полином (N,q), $q < 2^{32}$. Предполагается исоплъзование в качестве функции крптографической компрессии данных.
- (transpose.c) -- функция транспонирования матрицы 4x4, 5x5, 8x8, 16x16, до 32x32, прямоугольные матриц, с использованием векторных инструкций AVX512. Редуцирование тензорных операций с использованием транспонирования матриц.
- (mwc32.c)
  -- функция генерации PRNG MWC-32, с использованием векторных инструкций AVX512, для параллельного генератора. Методы MWC могут быть эффективно использованы на GPU в качестве источника квантовых шумов и в методах Stochastic Rounding.
  - функция хеширования для RPC протокола на базе векторов MWC-32
- (prime.c)
  -- генерация и проверка на простоту простых чисел вида $A \cdot 2^s \pm 1$. Простые числа Ризеля

и Прота. Реализован ряд критекриев и показателей применимости для Ring-LWE и генерации PRNG.

- qnn_protobuf.c
  -- бинарное кодирование тензорных данных в формате protobuf
  - реализация сериализации данных с кодированием protobuf,
  - загрузка и экспорт вычислительного графа в формате ONNX
- qnn_hexl.c Homogenous Encryption aXellerated Library
  -- алгоритмы для схемы FHE Ring-LWE с оптимизацией под AVX512 и AVX2. Оптимизация под 32-битные простые числа вида $A \cdot 2^s \pm 1$ и модули NIST PQC.
  - Редуцирование векторов по модулю простого числа q.
  - Прямое и обратное преобразование вещественных чисел в целые по модулю на кольце $\mathbb{Z}_q$. Алгоритм реализован для float32, поддерживаются типы MXINT8, MXFP8 и MXFP4. Используется в схеме кодирования LWE.
  - Полиномиальные операции сложения, вычитания, умножения, вращения.
  - Алгоритмы редуцирования полиномов $\mathbb{Z}_q/\langle x^N+1\rangle$: Барретта, Монтгомери, Шоупа.
  - Разработан алгоритм 4.1 для NTT с простыми числами вида $2^{32} - a_1 \cdot 2^{16} + 1$
  - Умножение полиномов с использованием прямого и обратного преобразования NTT (Number Theoretic Transform)
  - функция хеширования тензоров на кольце полиномов $\mathbb{Z}_q/\langle x^N+1\rangle$.
  - Алгоритм параллельного вычисления вектора хещей с разными $\{q_i\}$ для фильтра Блума.
  - Алгоритм параллельного вычисления Sbox (возведения в степень) по модулю.
- qnn_rns.c
  -- Residue Number System (RNS), непозиционная система остаточных классов
  - кодирование в RNS
  - реализация рашсширения базиса RNS через MRC
- qnn_rag.c
  -- реализация поиска по базе данных (тензоров, изображений и текстов) с использованием векторов семантических признаков

Направления практического применения:

- Разработка методов сравнения Embedding векторов в схеме Ring-LWE/LWR с использованием квантовой ошибки.
- Реализация схемы RAG (поиск по базе данных без раскрытия данных).
- шифрование и подпись тензоров в протоколе RPC для MPC (Multy-Party Computation)
- внедрение принципов кэширования тензоров и результатов тензорных операций с использованием векторов хэшей в RPC протокол.

Перенос алгоритмов на GPU

Цель - повышение производительности операций параллельной генерации RRNG и хеширования тензоров за счет тензорных вычислений на GPU.

- Модульная арифметика на кольце полиномов $\mathbb{Z}_q/\langle x^N+1\rangle$ может быть представлена, как тензорные операции (базовые операции линейной алгебры): ADDM, SUBM, MULM, DOTM, AXPYM, MVM (умножение матрицы на вектор).
- Преобразование тензоров весов и смещений в формате MXFP8, MXFP4 и MXINT8 может быть выполнено на GPU в группе (варп) из 32 ядер.
- Функция хеширования тензоров с редуцированием в кольце полиномов для N=256, N=32.

Спиок работ:

- [2507.04775] FIDESlib: A Fully-Fledged Open-Source FHE Library for Efficient CKKS on GPUs

{перенести к KEM}

- (https://pq-crystals.org/dilithium/data/dilithium-specification-round3.pdf)

Модуль q выбирается так, чтобы существовал N-й корень из единицы ζ по модулю q. Это означает, что циклический полином X^N+1 раскладывается на линейные множители $X-\zeta^i$ по модулю q при $i=1,3,5,\ldots,N-1$. По теореме об остатках циклическое кольцо $\mathcal{R}_q$ изоморфно произведению колец $\mathbb{Z}_q[X]/(X-\zeta^i)\cong\mathbb{Z}_q$. В этом произведении колец умножение элементов происходит поэлементно. Изоморфизм

$$a\mapsto(a(\zeta),a(\zeta^3),\ldots,a(\zeta^{N-1})):\mathcal{R}_q\to\prod_i\mathbb{Z}_q[X]/(X-\zeta^i)$$

может быть вычислен с помощью аналога Быстрого преобразования Фурье (NTT).

Поскольку $X^N+1=X^N-\zeta^N=(X^{N/2}-\zeta^{N/2})(X^{N/2}+\zeta^{N/2})$, можно сначала вычислить отображение

$$\mathbb{Z}_q[X]/(X^N+1)\to\mathbb{Z}_q[X]/(X^{N/2}-\zeta^{N/2})\times\mathbb{Z}_q[X]/(X^{N/2}+\zeta^{N/2})$$

и затем продолжить отдельно с двумя приведёнными полиномами степени менее $N/2$, заметив, что $\zeta^k=-\zeta^{N+k}$.

Почему циркулянты диагонализируемы?
Все циркулянтные матрицы являются нормальными матрицами, а нормальные матрицы всегда диагонализируемы.
В основе диагонализации лежит дискретное преобразование Фурье, которое выступает в роли унитарной матрицы, приводящей циркулянт к диагональному виду.

Можно ли применить принцип сжимающих отображений?

1. Существует неподвижная точка x^*, такая что $f(x^*) = x^*$, то есть f имеет фиксированную точку.
2. Пусть $x_0 \in X$ и $x_{n+1} = f(x_n)$, тогда $x_n \to x^*$ при $n \to \infty$.
3.