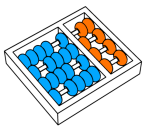
Universidade Estadual de Campinas
Instituto de Computação



Pedro Geraldo Morelli Rodrigues Alves

Computação sobre dados cifrados em *GPGPUs*

CAMPINAS
2016

Pedro Geraldo Morelli Rodrigues Alves

Computação sobre dados cifrados em *GPGPUs*

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Diego de Freitas Aranha

Este exemplar corresponde à versão final da Dissertação defendida por Pedro Geraldo Morelli Rodrigues Alves e orientada pelo Prof. Dr. Diego de Freitas Aranha.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): CNPq, 134218/2015-9; CAPES, 01-P-4554/2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

AL87c

Alves, Pedro Geraldo Morelli Rodrigues, 1988-
Computação sobre dados cifrados em GPGPUs / Pedro Geraldo Morelli Rodrigues Alves. – Campinas, SP : [s.n.], 2016.

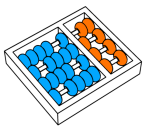
Orientador: Diego de Freitas Aranha.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia. 2. Criptografia homomórfica. 3. Programação paralela (Computação). 4. Computação de alto desempenho. I. Aranha, Diego de Freitas,1982-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Efficient GPGPU implementation of leveled fully homomorphic encryption scheme YASHE
Palavras-chave em inglês:
Cryptography
Homomorphic encryption
Parallel programming (Computer science)
High performance computing
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Diego de Freitas Aranha [Orientador]
Julio César López Hernández
Rafael Misoczki
Data de defesa: 09-06-2016
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação



Pedro Geraldo Morelli Rodrigues Alves

Computação sobre dados cifrados em *GPGPUs*

Banca Examinadora:

- Prof. Dr. Diego de Freitas Aranha
  Instituto de Computação (IC) - UNICAMP
- Prof. Dr. Julio César López Hernández
  Instituto de Computação (IC) - UNICAMP
- Prof. Dr. Rafael Misoczki
  Intel Labs

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 09 de junho de 2016

Aos meus pais, Ester e Geraldo Majela.

Agradecimentos

Ao meu orientador, Diego, pela paciência e dedicação em compartilhar todo o conhecimento e experiência que me faltaram na realização deste trabalho.

À minha mãe, Ester, por ter assumido sozinha e cumprido de forma exemplar a difícil tarefa de criar um adolescente e guia-lo em sua vida adulta. Ao meu pai, Geraldo Majela (*in memoriam*), pelas aulas de matemática e pela inspiração, que foram vitais para meu desenvolvimento acadêmico e profissional.

À Joyce, minha esposa e eterna namorada, por me prover chão firme, pela compreensão das ausências e pelas renúncias necessárias para compartilhar comigo este sonho.

Aos amigos, pela ajuda com os momentos de lazer e distração, tão necessários quanto aqueles de trabalho. Em especial ao meu amigo Anderson Coresma pela importantíssima ajuda com a gramática e à Jheyne pela companhia nos incontáveis cafés.

Ao LMCAD, pela infraestrutura e ambiente de trabalho, assim como aos seus membros pelos conhecimentos compartilhados e as diversas discussões pertinentes.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo financiamento.

Aos demais que de alguma maneira me ajudaram na conclusão deste trabalho: docentes, funcionários e colegas do Instituto de Computação da UNICAMP.

If you want to keep a secret,
you must also hide it from yourself.
(George Orwell)

Resumo

Há forte interesse da indústria de tecnologia em reduzir custos de instalação, manutenção e escalabilidade de servidores por meio da adoção do paradigma de computação em nuvem. Apesar disso, ainda faltam métodos que garantam o sigilo dos dados nesse ambiente. Ainda que seja possível garantir a privacidade durante o transporte e armazenamento, os dados precisam ser revelados ao serviço de nuvem para ocorrer processamento. A criptografia homomórfica é um ramo de criptossistemas que permitem computação sobre dados cifrados. Essa propriedade torna esquemas homomórficos candidatos promissores para a aplicação nesse contexto, fornecendo o transporte, armazenamento e processamento seguro dos dados. Este trabalho investiga estratégias para implementação eficiente do criptossistema completamente homomórfico em nível YASHE. Utilizou-se a plataforma CUDA para o processamento paralelo dos dados, além do teorema chinês do resto na substituição da aritmética de inteiros grandes por operações mais simples e implementadas com instruções nativas do processador. Também é realizada a comparação das transformadas *FFT* e *NTT* com o intuito de reduzir a complexidade computacional de uma operação de multiplicação polinomial. A implementação da primeira foi fornecida pela biblioteca cuFFT, enquanto a segunda foi implementada com código próprio baseado na formulação de Stockham. Como fruto das conclusões obtidas durante o desenvolvimento deste trabalho, a biblioteca cuYASHE foi desenvolvida e disponibilizada à comunidade. Sua implementação apresenta ganhos expressivos de velocidade em todas a operações do criptossistema em comparação com o estado da arte, o que sugere que as estratégias propostas são adequadas para a otimização. Foram feitas comparações entre a cuYASHE e quatro trabalhos com implementações baseadas em *CPU*, *GPU* e *FPGA*. Em particular, destaca-se o ganho de velocidade na multiplicação homomórfica, apresentando tempos de execução de 6 até 35 vezes menor. Como essa operação é essencial para avaliação de funções sobre criptogramas, foi possível concluir que o processamento em *GPGPU* é adequado para a implementação de serviços de computação em nuvem que preservam a privacidade.

Abstract

Security in cloud computing is a relevant topic for research. Multiple branches of industry are embracing this paradigm to reduce operational costs, improve scalability and availability. Surprisingly, several techniques are still missing to properly preserve privacy on the cloud. Employing encryption for data storage and transport is not enough once the data owner has no real control over the processing hardware. This way, security requirements must also be extended to data processing tasks. Homomorphic encryption schemes are natural candidates for computation over encrypted data, since they are able to satisfy the requirements imposed by the cloud environment. This work investigates strategies to efficiently implement the leveled fully homomorphic scheme YASHE. It employs the CUDA platform to provide parallel processing capabilities and the chinese remainder theorem to replace expensive big integer arithmetic by simpler instructions natively supported in hardware. Moreover, this work offers a comparison between the Fast Fourier transform and the Number-Theoretic transform for reducing the complexity of polynomial multiplication. The former is provided by the cuFFT library, while the latter is implemented through the Stockham formulation. As result of this research, the cuYASHE library was developed and made available to the community. When compared with the state-of-the-art implementation in *CPU*, *GPU* and *FPGA*, it shows speed-ups for all operations. In particular, there was an improvement between 6 and 35 times for polynomial multiplication. This operation is performance-critical for evaluating any function over encrypted data, demonstrating that GPUs are an appropriate technology for bootstrapping privacy-preserving cloud computing environments.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A computação em nuvem tem sido responsável por uma profunda mudança na comunidade de processamento distribuído. A possibilidade de terceirizar a instalação, manutenção e a escalabilidade de servidores, somada a preços competitivos, faz com que esses serviços se tornem altamente atraentes [11].

Diferentes serviços de nuvem surgiram na última década por conta de investimentos pesados de gigantes da indústria de tecnologia. Em pouco tempo, esse paradigma se popularizou e se tornou vital para a resolução de problemas computacionais, sendo abraçado por diferentes áreas da computação, de hospedagem de *sites* até computação científica. A Figura 1 apresenta as receitas para os principais serviços de nuvem em 2013 e 2015. Ela se limita aos serviços do tipo *IaaS* [1], ou "Infraestrutura como Serviço", que proveem infraestrutura e são usados, inclusive, por outros tipos de serviços de nuvem. É evidente o crescimento expressivo desse mercado nos últimos anos.

Durkee [24] discute em seu trabalho sobre a pressão que serviços de nuvem tem sofrido por conta da imensa demanda e grande concorrência. Ele afirma que há uma expectativa por parte dos clientes em resolver antigos problemas relacionados com a instalação e manutenção de um parque de *clusters*, como por exemplo a eficiência no consumo de recursos e a garantia de disponibilidade.

Dispositivos móveis, como *smartphone*, *tablets* ou *notebooks*, naturalmente sofrem da escassez de certos recursos, como capacidade de processamento, consumo energético e consequentemente autonomia de bateria. Dessa forma, a computação em nuvem também serviu como parte da solução para o desenvolvimento da indústria de dispositivos móveis. A nuvem, voltada para esses dispositivos, se tornou fundamental para suprir a demanda desses aparelhos, compensando sua limitação.

Dinh et al. [22] apresenta uma visão do mercado de computação móvel em nuvem. O trabalho discute como a nuvem, nesse contexto, frequentemente não assume apenas o armazenamento dos dados mas também seu processamento. Dessa forma pode-se aumentar a autonomia da bateria, compensar a possível falta de poder de processamento, aumentar a confiabilidade do ponto de vista de *backups* e a segurança contra progra-

[1] Acrônimo de *Infrastructure as a Service*.

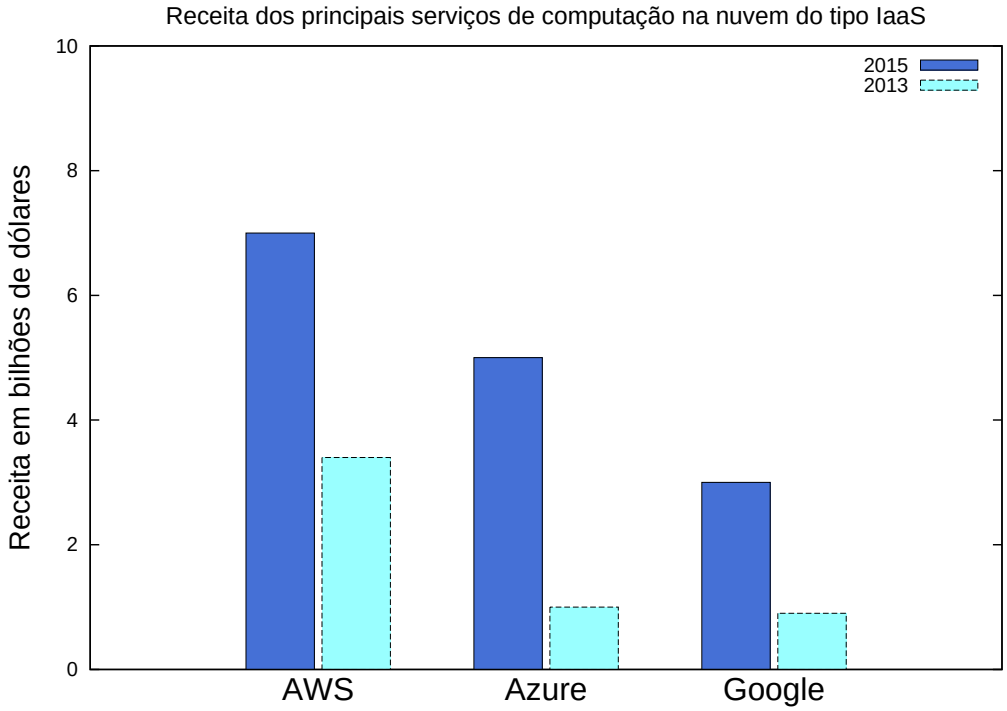


Figura 1.1: Comparação da receita dos principais serviços de computação em nuvem do tipo IaaS em 2013 e 2015. Valores para AWS [12] obtidos de seus relatórios anuais. As receitas para Azure e para os serviços de nuvem da Google são estimativas [38].

mas mal-intencionados. Essas questões permitem a redução dos requisitos de *hardware* no desenvolvimento desses dispositivos, e consequentemente o custo de produção desses dispositivos.

A computação científica, da mesma forma, encontrou maneiras de tirar proveito da computação em nuvem. Hoffa et al. [37] investigam a aplicabilidade desse paradigma na resolução de uma aplicação argumentada como comum na astronomia. Eles afirmam que o uso de recursos computacionais locais pode ser suficiente para pequenas instâncias do problema, mas essa não é uma solução escalável. Para instâncias maiores a computação em nuvem se torna mais vantajosa. Completando essa análise, Deelman [18] avalia a vantagem financeira no uso de nuvens nesse contexto e percebe que, com a devida análise do modelo de cobrança e o ajuste da configuração da nuvem que otimize tanto o poder computacional contratado quanto o custo, a computação científica em nuvem se mostra como uma solução com bom custo-benefício para problemas baseados em dados. Essa conclusão é reforçada pelo trabalho de Vecchiola, Pandey e Buyya [62], que também afirma que o uso da nuvem nesses problemas pode ser bastante vantajoso.

A adoção da computação em nuvem, entretanto, levanta importantes questões de segurança. A possibilidade do vazamento de informações acompanha o crescimento do número de entidades que manipulam os dados. Xiao [65] discute algumas das principais

questões de segurança envolvendo computação em nuvem, entre elas: confidencialidade e integridade dos dados.

O sigilo é apontado como uma das maiores preocupações. Isso se dá, principalmente, pela obrigatoriedade dos clientes em confiarem que os administradores da nuvem são honestos e verdadeiramente engajados em tornar o ambiente tão seguro quanto possível, reduzindo o risco de que entidades internas ou externas explorem brechas de segurança. Englobando, inclusive, os próprios administradores do serviço de nuvem.

A integridade dos dados implica que esses precisam ser armazenados de forma honesta e qualquer violação dessa integridade (como perda, alteração ou vazamento, por exemplo) será detectada. Da mesma forma, é esperado que a computação sobre esses dados também seja feita de forma integra, sem influência ou alterações por *softwares* maliciosos, outros usuários ou, de novo, o administrador da nuvem.

Diversos esquemas criptográficos são utilizados como padrão no armazenamento e troca de dados. Protocolos como o *TLS* [19] utilizam diferentes métodos para garantir a troca de mensagens de forma segura entre duas entidades. Criptossistemas simétricos são usados para proteger a mensagem, como por exemplo o *AES* [16], enquanto criptossistemas assimétricos, como o *RSA* [54] ou outros baseados em curvas elípticas [34], assumem o papel de garantir a troca de chaves, a integridade e a garantia de não-repúdio dos dados.

No caso da computação em nuvem, como discutido, existe a possibilidade de se lidar com um atacante que não apenas pode ter acesso ao dado armazenado ou mesmo interceptá-lo, como também pode ter acesso diretamente ao *hardware* que realiza o processamento. Desse modo, existe a necessidade que o processamento também seja protegido, garantindo uma trajetória completamente sigilosa para os dados: transporte, processamento e armazenamento.

Em 1978, Rivest et al. tentaram resolver esse tipo de problema através da criptografia homomórfica [53]. Essa classe de criptossistemas engloba esquemas que possibilitam a manipulação de criptogramas sem o conhecimento da chave de decifração. Operações de adição e multiplicação, assim como outras construídas sobre essas, podem ser aplicadas sem o conhecimento dos operandos. O resultado dessas operações é nativamente cifrado com a chave original, sem revelá-la em nenhum momento. Desde então, diversas abordagens foram tentadas para construir criptossistemas que suportassem essa propriedade. Alguns deles são bastante conhecidos, como o *RSA*, *ElGamal* ou *Paillier*.

A criptografia homomórfica se mostra promissora para aumentar a segurança na nuvem. Através desse tipo de cifração, não há razão para que dados sejam revelados para a nuvem ou para qualquer outra entidade que os processe. Dessa forma, a segurança na nuvem passa a não depender da confiança em seus administradores ou na garantia do funcionamento correto de seus métodos de segurança.

1.1 Motivação

A computação em nuvem é um tipo de serviço com intensa adoção pela indústria de tecnologia. Apesar disso, ainda há trabalho a ser feito na atualização de requisitos de segurança, quando nesse novo contexto. A perda de controle do *hardware*, da sua manutenção e localização física, implicam em novos cuidados a serem tomados para garantir o sigilo dos dados.

Em 2013 foi revelado [33] que a *NSA* [2] operava um programa de espionagem doméstico conhecido como *PRISM*. Seu objetivo era de interceptar informações que trafegassem pela *internet* através da infraestrutura dos EUA.

A metodologia usada pela *PRISM* era obrigar provedores de serviços, como a Microsoft, Google ou Yahoo, a entregar dados sobre indivíduos escolhidos arbitrariamente pela agência de inteligência. Além disso, foi revelado que o programa também era capaz de acessar dados sem precisar de autorização ou cooperação das empresas, através da exploração de falhas de segurança em *software* ou em *hardware* [63]. Em ambos os casos, os resultados eram armazenados em um banco de dados e podiam ser consultados por analistas em baixa posição hierarquica, sem a necessidade de aprovação jurídica ou supervisão.

Esse exemplo demonstra um caso onde administradores de serviços de nuvem foram incapazes de proteger dados armazenados em seus servidores. Uma vez que o *software* ou o *hardware* são intencionalmente comprometidos, diligências na conservação de sigilo de dados tornam-se inúteis. A única solução definitiva para esse problema é que os serviços de nuvem se resignem à sua capacidade de acessar dados de seus usuários. Dessa forma, mesmo que esses dados venham a ser interceptados, com ou sem conhecimento dos administradores da nuvem, o sigilo é mantido enquanto os criptossistemas (e suas implementações) forem seguros.

Em 2015, o *Facebook* se recusou a cumprir uma ordem judicial emitida por um tribunal brasileiro ordenando a quebra de sigilo de mensagens trocadas através de seu serviço de bate papo, *WhatsApp* [28]. Isso gerou uma disputa que provocou o pedido de bloqueio do serviço durante 48 horas por todo o território nacional.

Após o *PRISM* ter sido revelado, houve uma movimentação da indústria no sentido de tentar se proteger de programas desse tipo. O *WhatsApp* tomou como estratégia a proteção das mensagens trocadas pelos seus usuários através de um sistema de cifração ponta-a-ponta. Dessa forma, as mensagens passaram a ser visíveis apenas entre os atores em conversação. Nenhum outro ator, mesmo o *Facebook*, se tornou capaz de quebrar o sigilo dessas mensagens.

Esse tipo de estratégia resultou na imposição de sigilo absoluto de dados providos por usuários à um serviço de nuvem. Apesar dos possíveis dilemas morais e legais (como a impossibilidade do cumprimento de uma ordem judicial), essa é uma demonstração da tendência de serviços em nuvem em se armarem contra intrusos indesejados em seus

[2] Acrônimo de *National Security Agency*. Refere-se à agência de inteligência dos EUA.

sistemas e evitarem, inclusive, repercusões negativas ao seu modelo comercial.

Essa crise de confiança nos provedores de computação em nuvem ameaça todo o segmento. Usuários domésticos e corporativos começaram a repensar e considerar novos cuidados na adoção desses serviços. Há estimativas de que as revelações sobre o programa de vigilância doméstica implantado pela *NSA* podem causar reduções de receita de U$35 até U$180 bilhões em 2016 por conta da perda de negócios com o mercado estrangeiro. De fato, já se observa migração de negócios da indústria norte-americana para outros países. A China, por exemplo, chegou a declarar a intenção de abandonar tecnologia estrangeira por conta das revelações de Snowden [8].

1.2 Objetivos

Este trabalho se propôs a contribuir com o aumento da segurança na computação em nuvem ao estudar maneiras seguras de realizar o processamento de dados.

Em especial, desejou-se estudar maneiras de implementar um criptossistema homomórfico, o YASHE [9], e assim obter ganho de velocidade nas operações homomórficas em relação ao estado da arte. A abordagem foi usar paralelismo através de *GPGPUs* sobre a plataforma CUDA [3].

1.3 Contribuições

As contribuições deste trabalho se voltaram ao estudo de como implementar a aritmética polinomial necessária ao YASHE, em *GPGPUs*. Entre as contribuições obtidas, se tem:

- Formulação da aritmética que simplifique o cálculo das operações utilizadas pelo esquema YASHE. Em particular, a escolha eficiente de módulos com formato especial (primo de Mersenne e polinômio ciclotômico) para parametrizar o anel algébrico acelera operações de resto na implementação. Até onde foi possível determinar, a utilização de um primo de Mersenne na instanciação do esquema é contribuição inédita deste trabalho.

- O desenvolvimento de métodos para a implementação das operações de adição e multiplicação polinomial em CUDA. Foi realizado um estudo do teorema chinês do resto no contexto de paralelismo em *GPGPUs* [4], com o objetivo de simplificar a aritmética de inteiros grandes. Além disso, realizou-se a análise de como as transformadas *FFT* e *NTT* podem ser aplicadas na redução da complexidade de operações polinomiais.

[3] Acrônimo de *Compute Unified Device Architecture*.
[4] Acrônimo de *General Purpose Graphics Processing Unit*.

- O desenvolvimento da biblioteca cuYASHE [3]. Tal biblioteca aplica todo o conhecimento adquirido com este trabalho e provê uma implementação em CUDA do criptossistema em questão, com ganhos de velocidade de até 35 vezes na adição homomórfica e 6 vezes na multiplicação homomórfica, em relação ao estado da arte.

Resultados preliminares foram apresentados no XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais [1] e no X Workshop de Teses, Dissertações e Trabalhos de Iniciação Científica em Andamento do IC-Unicamp [2].

1.4 Estrutura do documento

Este documento é organizado como se segue. O Capítulo 2 apresenta as bases teóricas para este trabalho. É oferecida uma definição formal para a propriedade homomórfica em criptossistemas, assim como é apresentado o criptossistema YASHE. Além disso, o teorema chinês do resto e as transformadas utilizadas para simplificar a multiplicação polinomial recebem uma discussão formal. Nos Capítulos 3 e 4 são apresentados, respectivamente, a implementação cuYASHE com seus detalhes de construção e uma análise do ganho de desempenho sobre o estado da arte com a justificativa para algumas decisões de projeto. Por fim, O Capítulo 5 conclui apresentando um resumo do trabalho.

Capítulo 2

Fundamentação teórica

A implementação eficiente da aritmética de um criptossistema é fator fundamental para o ganho de velocidade em suas operações. Neste trabalho foi necessário que se estudasse estratégias para a implementação eficiente da aritmética polinomial sobre inteiros grandes. Essas estratégias envolveram a simplificação da implementação da aritmética de corpos finitos por meio do teorema chinês do resto, além do uso de transformadas para a redução da complexidade computacional da multiplicação polinomial.

Este Capítulo apresenta as ferramentas utilizadas para a implementação de um criptossistema que permite a manipulação de criptogramas.

2.1 Corpos finitos

A aritmética de corpos finitos é bastante comum em criptossistemas de chave pública. Seu funcionamento depende de algumas estruturas algébricas básicas como grupos, anéis e corpos, que são apresentadas nas Definições 1, 2 e 3. A Definição 4 apresenta o conceito de corpo finito.

Definição 1 (Grupo)**.** *Um grupo é um conjunto de elementos G equipado com uma operação $\circ$. Um grupo tem as seguintes propriedades:*

1. *A operação é fechada. Ou seja, para $\forall a, b \in G$, temos que $(a \circ b) \in G$.*
2. *A operação é associativa. Ou seja, $a \circ (b \circ c) = (a \circ b) \circ c$, para todo e qualquer $a, b, c \in G$.*
3. *Há um elemento $I \in G$ tal que $a \circ I = I \circ a = a$, para todo e qualquer $a \in G$. Ele é chamado de elemento neutro.*
4. *Para todo e qualquer $a \in G$ há um elemento $a^{-1} \in G$ tal que $a \circ a^{-1} = a^{-1} \circ a = I$. Esse elemento é chamado inverso.*
5. *Um grupo G é chamado abeliano (ou comutativo) se, além das propriedades citadas, $a \circ b = b \circ a$ para todo e qualquer $a, b \in G$.*

Definição 2 (Anel)**.** *Um conjunto R de elementos é chamado anel se satisfizer as seguintes propriedades:*

1. *R é um grupo abeliano com respeito a $+$.*
2. *Além da adição, R também é equipado com a operação multiplicação, $\times$.*
3. *A operação $\times$ é associativa e possui elemento neutro em R.*
4. *Quando as duas operações são combinadas, a propriedade distributiva da multiplicação sobre a adição se mantém. Ou seja, para todo e qualquer a, b e $c \in R$: $a \times (b + c) = (a \times b) + (a \times c)$.*

Definição 3 (Corpo)**.** *Um corpo $\mathcal{F}$ é um conjunto de elementos com as seguintes propriedades:*

1. *Todos os elementos de $\mathcal{F}$ formam um grupo aditivo com a operação $+$ e o elemento neutro 0.*
2. *Todos os elementos de $\mathcal{F}$, com exceção de 0, formam um grupo multiplicativo com a operação $\times$ e o elemento neutro 1.*
3. *Quando as duas operações do grupo são combinadas, a propriedade distributiva da multiplicação sobre a adição se mantém. Ou seja, para todo e qualquer a, b e $c \in \mathcal{F} : a \times (b + c) = (a \times b) + (a \times c)$.*

Pode-se perceber que, de acordo com essas Definições, um anel é dado como um grupo aditivo com uma operação adicional e a propriedade distributiva. Um corpo, por sua vez, completa as propriedades para esta operação, de forma que ele pode ser visto como um grupo ao mesmo tempo aditivo e multiplicativo.

Definição 4 (Corpo finito)**.** *Um corpo com um número finito de elementos é chamado corpo finito.*

Um exemplo importante de um corpo finito é aquele formado por inteiros módulo p, onde p é um primo. Este corpo finito é denotado por $\mathbb{Z}_p$ e suas operações de adição e multiplicação são realizadas módulo p.

Definição 5 (Polinômios irredutíveis)**.** *Sejam F um corpo finito e R um anel de polinômios com coeficientes em F. Um polinômio P é dito irredutível sobre F se $P \in R$ e não existirem polinômios $A, B \in R$ tal que $A \cdot B = P$, ou seja, P não pode ser fatorado pelo produto de dois polinômios de R.*

Também é possível definir um corpo finito sobre polinômios. Sejam $\mathbb{Z}_q[X]$ um conjunto de polinômios com coeficientes em $\mathbb{Z}_q$ e $\Phi_n(X) \in \mathbb{Z}_q[X]$ um polinômio irredutível. Então $R_q = \mathbb{Z}_q[X]/\Phi_n(X)$ é um corpo finito composto por polinômios com operações de adição e multiplicação módulo $\Phi(X)$.

Definição 6 (Polinômios ciclotômicos)**.** *Seja P um polinômio irredutível em um corpo F. P é dito o n-ésimo polinômio ciclotômico se for um divisor de $x^n - 1$ e não for um divisor de $x^k - 1$ para $k < n$, onde $n, k \in \mathbb{Z}$.*

Em especial, se n for uma potência de 2, o n-ésimo polinômio ciclotômico é dado por $\Phi_n(x) = x^{n/2} + 1$.

Um tipo especial de polinômios irredutíveis são os ciclotômicos. Este conjunto é formado por polinômios irredutíveis com a propriedade especial apresentada na Definição 6.

Teorema 1 (Pequeno Teorema de Fermat)**.** *Seja a um inteiro e p um primo. Então,*

$$a^p \equiv a \mod p.$$

Por fim, o Pequeno Teorema de Fermat provê um resultado muito útil para a aritmética de corpos finitos que é o cálculo do inverso multiplicativo modular. Se $a^p \equiv a \mod p$, então o inverso é dada por $a^{p-2} \mod p$.

Demonstração. Seja a um elemento de um corpo finito $\mathbb{Z}_p$. Então pelo Teorema 1,

$$\begin{aligned} a^p &\equiv a \mod p, \\ a^p a^{-2} &\equiv a a^{-2} \mod p, \\ a^{p-2} &\equiv \left(a a^{-1}\right) a^1 \mod p, \\ a^{p-2} &\equiv a^{-1} \mod p. \end{aligned}$$

Dessa forma, em um corpo finito $\mathbb{Z}_p$ a inversa multiplicativa modular de a é dada por $a^{p-2} \mod p$. □

2.2 Segurança de criptossistemas

A construção de criptossistemas sobre problemas matemáticos permite que se desenvolva uma demonstração quanto à sua segurança. Essa é classificada de acordo com o tipo de ataque que um criptograma suporta sem que ocorra quebra do sigilo da mensagem. Dois problemas bastante conhecidos são o problema de fatoração de inteiros e o problema do logaritmo discreto [48], que servem como base para a construção do RSA e do ElGamal.

2.2.1 Problemas subjacentes

As Definições 8 e 9 apresentam dois dos principais problemas utilizados na construção de criptossistemas resistentes a ataques quânticos [41]. Computadores quânticos são vistos como uma das principais ameaças para o futuro da criptografia, uma vez que estes serão capazes de resolver em tempo polinomial os problemas subjacentes dos principais criptossistemas em uso atualmente.

Ring learning-with-errors - RLWE Para as Definições 7 e 8, sejam $R_q = \mathbb{Z}_q[x] / \langle f(x) \rangle$ o anel de polinômios com coeficientes em $\mathbb{Z}_q$ módulo $f(x)$, e χ_{err} uma distribuição de erro sobre R concentrada em elementos pequenos.

Definição 7 (Distribuição *RLWE*)**.** *Seja $s \in R_q$, conhecido como segredo. A distribuição $A_{s,\chi_{err}}$, sobre $R_q \times R_q$ é amostrada escolhendo $a \in R_q$ uniformemente aleatório e $e \leftarrow \chi_{err}$, e retornando $(a, b = s \cdot a + e \mod q)$.*

Definição 8 (Problema de decisão *RLWE*)**.** *Sejam m pares (a, b) amostrados da distribuição $A_{s,\chi_{err}}$ – com s escolhido uniformemente e fixo para todos os pares – ou da distribuição uniforme.*

O problema de decisão RLWE consiste em distinguir em qual caso cada par se enquadra.

A Definição 8 apresenta o problema de decisão *RLWE* de maneira informal. Esse problema é considerado difícil, como discutido por Peikert [51] e Lyubashevsky [44]. Não existe algoritmo conhecido que resolva o *RLWE* em tempo polinomial.

Decisional Small Polynomial Ratio - DSPR O problema *DSPR* foi introduzido por López-Alt, Tromer e Vaikuntanathan [43]. Seu uso neste trabalho trouxe ganho de velocidade para as operações do criptossistema escolhido.

Definição 9 (Problema *DSPR*)**.** *Sejam d e q inteiros, $R_q = \mathbb{Z}_q/(x^n + 1)$ um anel de polinômios com coeficientes em $\mathbb{Z}_q$ e χ uma distribuição sobre R_q. Sejam também $t \in R_q^X$ inversível em R_q, $y_i \in R_q$ e $z_i = -y_i t^{-1} \mod q$, para $i \in \{1, 2\}$.*

O problema DSPR consiste em distinguir elementos da forma $h = a/b$, onde $a = y_1 + t \cdot \chi_{z_1}$, $b = y_2 + t \cdot \chi_{z_2}$ de elementos amostrados uniformemente de R_q.

2.2.2 Noções de segurança

Em criptografia, a comprovação do nível de segurança de um criptossistema é fator fundamental para a definição do seu contexto de uso. Em especial, a indistinguibilidade dos criptogramas é uma propriedade corriqueiramente buscada nessa avaliação de segurança. Dois cenários são considerados, quando o adversário é capaz de cifrar mas não decifrar mensagens e quando o adversário é capaz também de decifra-las. Usualmente, esses dois testes são avaliados através de um jogo, onde um adversário tenta adquirir conhecimentos sobre mensagens cifradas por um desafiante.

Indistinguibilidade de criptogramas contra ataque de texto claro escolhido - *IND-CPA*.

1. O desafiante gera um par de chaves PK e SK. A primeira é disponibilizada ao adversário enquanto a segunda é mantida sob sigilo.

2. O adversário pode realizar uma quantidade limitada de cifrações ou outras operações.

3. Em dado instante, o adversário envia dois textos claros M_0 e M_1 ao desafiante.

4. O desafiante toma um *bit* b aleatório e retorna a cifra de M_b, calculada a partir da chave PK, ao adversário.

5. O adversário pode realizar operações livremente e deve, ao final, adivinhar se o criptograma retornado é fruto da cifração de M_0 ou M_1.

Um criptossistema é dito indistinguível contra um ataque com texto claro escolhido, ou possui a segurança *IND-CPA*, se o adversário for incapaz de acertar (com vantagem não desprezível a uma adivinhação aleatória) qual das mensagens foi cifrada.

Indistinguibilidade de criptogramas contra ataques (adaptativos) de criptograma escolhido - *IND-CCA* ou *IND-CCA2* A definição desse tipo de indistinguibilidade se difere do *IND-CPA*, uma vez que o adversário tem acesso, além da chave PK, a um oráculo de decifração. Esse oráculo é capaz de decifrar criptogramas de interesse do adversário. Nesse contexto, o jogo é divido em uma versão adaptativa e outra não-adaptativa.

Na definição não-adaptativa, *IND-CCA*, o adversário pode utilizar o oráculo de decifração apenas até receber o criptograma-resposta do desafiante. Já na versão adaptativa, *IND-CCA2*, ele fica disponível inclusive após esse evento. Obviamente, o oráculo não pode decifrar o criptograma-resposta. Isso tornaria o desafio trivial.

1. O desafiante gera um par de chaves PK e SK. A primeira é disponibilizada ao adversário enquanto a segunda é mantida sob sigilo.

2. O adversário pode realizar uma quantidade limitada de cifrações ou outras operações, inclusive decifrações por meio do oráculo.

3. Em dado instante, o adversário envia dois textos claros M_0 e M_1 ao desafiante.

4. O desafiante toma um *bit* b aleatório e retorna a cifra de M_b (chamado criptograma-resposta), calculada a partir da chave PK, ao adversário.

5. O adversário pode realizar operações livremente de acordo com a versão do jogo em questão. Ao final, ele deve adivinhar se o criptograma retornado é fruto da cifração de M_0 ou M_1.

Um criptossistema é dito indistinguível contra um ataque (adaptativo) de criptograma escolhido, ou possui a segurança *IND-CCA*/*IND-CCA2*, se o adversário for incapaz de acertar (com vantagem não desprezível a uma adivinhação aleatória) qual das mensagens foi cifrada.

2.3 Criptografia homomórfica

Um esquema criptográfico homomórfico pode ser definido como na Definição 10.

Definição 10. *Seja E uma função de cifração e D a função de decifração correspondente. Sejam m_1 e m_2 dados em claro. A dupla (E, D) forma uma cifra dita homomórfica com respeito a um operador $\diamond$ se a seguinte propriedade for satisfeita para um par fixo de chaves de cifração e decifração:*

$$E(m_1) \circ E(m_2) \equiv E(m_1 \diamond m_2).$$

Ou seja, a operação $\circ$ é equivalente à operação $\diamond$ no espaço de criptogramas.

Criptossistemas homomórficos são classificados de acordo com a quantidade de operações homomórficas suportadas e suas limitações. A seguir, as principais classes serão apresentadas.

Criptossistemas parcialmente homomórficos são criptossistemas que satisfazem a Definição 10 para operações de adição ou multiplicação, sem restrições. Existem diversas propostas de esquemas desse tipo, dentre elas os esquemas de Paillier [49] e ElGamal [25], apresentados nas Seções 2.3.2 e 2.3.3. Esses são bastante conhecidos e se caracterizam pelo bom desempenho.

Criptossistemas completamente homomórficos são criptossistemas que satisfazem a Definição 10 para ambas as operações de adição e multiplicação, sem restrições. Esquemas desse tipo são construídos a partir de esquemas *SHE* e *LHE*.

Em uma posição intermediária aos criptossistemas anteriormente citados, existem criptossistemas ditos ligeiramente homomórficos[1] e completamente homomórficos em nível[2].

Criptossistemas ligeiramente homomórficos, ou *SHE* [3], são aqueles que satisfazem a Definição 10 para uma quantidade limitada de operações de adição e multiplicação. No trabalho de Gentry [29], é demonstrado como construir um criptossistema completamente homomórfico a partir de um criptossistema ligeiramente homomórfico por meio de uma técnica chamada *bootstrap*. Essa técnica reduz o ruído acumulado após uma série de operações homomórficas, o que viabiliza a aplicação da decifração. Esse trabalho é bastante citado na literatura e se tornou referência para essa classe de criptossistemas.

Criptossistemas completamente homomórficos em nível, ou *LHE* [4], são derivados da proposta anterior de Gentry. Braverski, Gentry e Vaikuntanathan [10] definem esquemas dessa classe como aqueles capazes de avaliar circuitos de tamanho arbitrário sem a necessidade de uma técnica de *boostrap* mas apenas variando os parâmetros de cifração.

[1] Do Inglês, *Somewhat Homomorphic Encryption*.
[2] Do Inglês, *Leveled Fully Homomorphic Encryption*.

Criptossistemas parcialmente homomórficos são conhecidos há décadas. Contudo, as aplicações mais comuns de processamento de dados, como aquelas de estatística ou financeiras por exemplo, frequentemente requerem as duas operações simultaneamente. Dessa forma, esses esquemas frequentemente não são suficientes para aplicações reais sobre dados cifrados.

Em 2009, Craig Gentry propôs em sua tese de doutorado [29] o primeiro esquema funcional completamente homomórfico, além de um guia de como converter um esquema *SHE* em um esquema *FHE*. Esse evento foi bastante notável na comunidade de criptografia e reacendeu a pesquisa em criptossistemas homórficos.

2.3.1 Segurança

O maior nível de segurança que se pode obter para um criptossistema homomórfico é *IND-CCA*. A explicação para essa afirmação vem do fato de que no jogo *IND-CCA2*, o adversário é capaz de utilizar o oráculo de decifração mesmo depois de receber o criptograma-resposta.

Seja um jogo em andamento onde um criptossistema homomórfico é avaliado para o nível de segurança *IND-CCA2*. Como esperado, o adversário envia dois textos claros, M_0 e M_1, para o desafiante e recebe um criptograma-resposta. Por conta das regras do jogo, o adversário não pode submeter esse criptograma ao oráculo de decifração. Contudo, ele pode cifrar um novo texto claro arbitrário M_2 e aplicar alguma das operações homomórficas sobre o novo criptograma e o criptograma-resposta. O resultado dessa operação, ao contrário do criptograma-resposta, pode ser decifrado pelo oráculo (a única restrição é quanto a decifração do criptograma-resposta). Assim, como o adversário conhece M_0, M_1 e M_2, ele adquire conhecimento sobre qual dos textos claros foi cifrado e ganha o desafio. Se ele não fosse capaz de utilizar o oráculo após o recebimento do criptograma-resposta, que é o caso do jogo *IND-CCA*, essa estratégia não seria possível.

Criptossistemas homomórficos, por construção, permitem a qualquer entidade manipular criptogramas. Apesar do resultado ser mantido sob sigilo, perde-se a garantia da integridade da mensagem.

2.3.2 Proposta de Paillier

O esquema criptográfico de Goldwasser-Micali [30] foi proposto em 1982 e é a origem da arvore de propostas de onde o esquema de Paillier surgiu. Seu principio básico é o particionamento de um subconjunto de inteiros módulo $n = pq$ em duas partes secretas M_0 e M_1, onde p e q são primos. Assim o processo de cifração seleciona um elemento aleatório do conjunto M_b para cifrar o *bit* b, enquanto que o processo de decifração envolve descobrir de qual das partes o elemento selecionado pertence. Esse esquema tem uso prático improvável principalmente devido ao baixo desempenho. Ele faz cifração e decifração *bit*-a-*bit*, e por depender de mapeamentos de cada *bit* em um inteiro módulo n seu fator de expansão é $\log_2 n$.

O esquema de Paillier é uma proposta consideravelmente mais adequada para uso prático. Baseado no problema da fatoração de inteiros, ele gera cifras em $Z^*_{n^2}$. Seu funcionamento é descrito na Definição 11.

Definição 11 (Esquema de Paillier)**.** *As operações do criptossistema são definidas a seguir.*

***Geração de chaves**:*

1. *Escolher dois primos p,q de mesma ordem e definir* $n = pq$ *e* $\alpha = MMC(p-1, q-1)$.
2. *Escolher aleatoriamente* $g \in \mathbb{Z}^*_{n^2}$.
3. *Define-se* (n, g) *como a chave pública e* (p, q) *como a chave privada.*

***Cifração**: Dado a chave pública* (n, g) *e uma mensagem* $m \in \mathbb{Z}_n$,

1. *Escolher aleatoriamente* $r \in \mathbb{Z}^*_n$.
2. *Definir o criptograma* $c = g^{m+nr} \mod n^2$.

***Decifração**: Dado a chave privada* (p, q) *e uma cifra c,*

Calcular $m \equiv \frac{L\left(c^\alpha \mod n^2\right)}{L(g^\alpha \mod n^2)} \mod n$, *com* $L(u) = \frac{u-1}{n}$ *a função de Carmichael [26].*

O custo computacional da função de cifração no esquema de Paillier depende de uma exponenciação e uma multiplicação módulo n^2. Em seu trabalho, Paillier demonstrou como realizar a decifração de forma eficiente através do teorema chinês do resto [21] e de como a aplicação do algoritmo de exponenciação binária [48] reduz significativamente o custo das exponenciações. Contudo, as operações homomórficas sobre criptogramas, assim como a cifração e decifração, tem custo computacional alto por serem feitas em módulo n^2.

O criptossistema de Paillier é probabilístico, tem segurança contra ataques "passivos" (recuperar uma mensagem cifrada através da chave pública e da cifra) baseada no problema da fatoração de inteiros e possui segurança contra ataques do tipo *IND-CPA*.

2.3.3 Proposta de ElGamal

O esquema de ElGamal foi proposto por Taher Elgamal [25] em 1985 e é baseado no problema de Diffie-Hellman [20]. Ele possui diversas variações para cifração e assinatura digital e é um esquema probabilístico com homomorfismo multiplicativo. Ele é facilmente adaptável para adquirir o homomorfismo aditivo [15] e inclusive para se tornar determinístico. Sua proposta original é descrita na Definição 12.

Definição 12 (Esquema de ElGamal)**.** *As operações do criptossistema são definidas a seguir.*

***Geração de chaves**:*

1. *Escolher um primo p grande e um elemento gerado α do grupo cíclico $\mathbb{Z}_p^*$.*
2. *Escolher d tal que $1 < d < p-1$ e definir $\beta = \alpha^d \mod p$.*
3. *Definir (p, α, β) como a chave pública e (d) como a chave privada.*

***Cifração**: Dado a chave pública (p, α, β) e uma mensagem m,*

1. *Escolher aleatoriamente i tal que $1 < i < p-1$.*
2. *Calcular $(k_E, c) = (\alpha^i, m\beta^i) \in Z_p^*$. k_E é a chave efêmera.*

***Decifração**: Dado a chave privada (d), uma cifra c e sua chave efêmera k_E,*

Calcular $m = c.k_M^{-1} \in \mathbb{Z}_p^*$, com k_M, a chave de mascaramento, tal que $k_M \equiv k_E^d \mod p$.

A segurança do esquema de ElGamal contra ataques "passivos" (recuperar uma mensagem cifrada através da chave pública e da cifra) se baseia na dificuldade do problema de Diffie-Hellman[20]. Atualmente não há estratégia conhecida para resolver esse problema a não ser por meio do problema do logaritmo discreto[48]. Assim como o esquema de Paillier, ele também possui segurança *IND-CPA*.

Por fim, caso uma mesma chave efêmera seja utilizada para cifrar duas mensagens diferentes, o vazamento da mensagem relativa a um dos criptogramas permite a um atacante recuperar a chave de mascaramento k_M e utiliza-la para decifrar o outro criptograma.

2.4 YASHE - *Yet Another Somewhat Homomorphic Encryption*

YASHE [9] é um criptossistema completamente homomórfico em nível baseado em uma proposta de Stehlé e Steinfield [61] e no problema *RLWE*. Ele opera sobre elementos de um anel gerado por um polinômio ciclotômico em $\mathbb{Z}_q$.

2.4.1 Descrição

Seja χ_{err} a distribuição Gaussiana discreta e χ_{key} a distribuição estreita, que gera valores aleatoriamente dentre $\{-1, 0, 1\}$. O criptossistema YASHE é composto pelas seguintes operações:

Escolha de parâmetros: Dado um parâmetro λ de segurança,

1. Seja q um inteiro que define o espaço dos coeficientes dos criptogramas.

2. Seja t um inteiro que define o espaço dos coeficientes dos textos claros tal que $1 < t < q$.

3. Seja $w > 1$ um inteiro que define o tamanho da palavra usada na etapa de decomposição da operação *KeySwitch*.

4. Seja o anel $R_q = \mathbb{Z}_q[X]/\phi_n(X)$, onde $\phi_n(X)$ é o n-ésimo polinômio ciclotômico.

PowersOf$_{w,q}$: Dado o polinômio a.

Retornar $\left(\left[a \cdot w^i\right]_q\right)_{i=0}^{\log_w q - 1} \in R^{\log_w q}$.

WordDecomp$_{w,q}$: Dado o polinômio a.

Retornar $\left(\left[a_i\right]_w\right)_{i=0}^{\log_w q - 1} \in R^{\log_w q}$.

Geração de chaves:

1. Amostrar dois polinômios f' e g da distribuição χ_{key} e definir $f = \left[tf' + 1\right]_q$.

2. Se f não for inversível em R_q, repetir o passo 1 para um novo f'.

3. Seja $h = \left[tgf^{-1}\right]_q$.

4. Seja $\gamma = \left[PowersOf_{w,q}(f) + e + h \cdot s\right]_q \in R_q^{\log_w q}$.

5. Retornar as chaves pública, secreta e de avaliação: $(pk, sk, evk) = (h, f, \gamma)$.

Cifração: Dadas a chave (pk) e uma mensagem $m \in R_t$,

1. Amostrar dois polinômios s e e da distribuição χ_{err}.

2. Retornar o criptograma $c = \left[\frac{q}{t} \cdot [m]_t + e + h \cdot s\right] \in R_q$.

Decifração: Dadas a chave (sk) e um criptograma $c \in R_q$,

Retornar o texto claro $m = \left[\left\lfloor \frac{t}{q} \cdot [fc]_q \right\rceil\right]_t \in R$.

Adição: Dados os criptogramas c_1 e c_2,

Retornar $c_{add} = \left[c_1 + c_2\right]_q$.

Multiplicação: Dados os criptogramas c_1 e c_2 e a chave evk,

1. Seja $\hat{c}_{mult} = \left[\left\lfloor \frac{t}{q} \cdot c_1 \cdot c_2 \right\rceil\right]_q$.

2. Retornar $c_{mult} = KeySwitch(\hat{c}_{mult}, evk)$.

KeySwitch: Dado o criptograma c e a chave evk,

Retornar $\left[\langle WordDecomp_{w,q}(c), evk\rangle\right]_q$.

Funções de decomposição O YASHE possui duas operações auxiliares que manipulam os operandos de acordo com o tamanho de palavra w. A primeira, $PowersOf_{w,q}$, recebe um polinômio a e o mapeia em um conjunto formado pelo produto de a com potências de w módulo q. $WordDecomp_{w,q}$, por sua vez, decompõe o operando em $\log_w q$ polinômios com coeficientes em $\mathbb{Z}_w$.

Sobre a multiplicação A multiplicação homomórfica depende da avaliação pela função *KeySwitch* de um resultado intermediário $\hat{c}_{mult}$, calculado a partir da multiplicação polinomial normalizada dos criptogramas. O Teorema 2 e o Lema 1 são propostos pelos autores do YASHE [9] e analisam o ruído da multiplicação.

Teorema 2 (Ruído da multiplicação)**.** *Sejam $c_1, c_2 \in R$ os respectivos criptogramas resultantes da cifração das mensagens $m_1, m_2 \in R$, ambos decifráveis com a chave secreta f. Além disso, sejam $v_1, v_2 \in R$ seus ruídos inerentes tal que $||v_i||_\infty \leq \Delta/2$, para $i \in \{1, 2\}$ e $\Delta = \lfloor q/t \rfloor$. Seja $\hat{c}_{mult}$ o resultado intermediário da operação de multiplicação do* YASHE*.*

Então $f^2\hat{c}_{mult} = \Delta\left[m_1 m_2\right]_t + \hat{v}_{mult} \mod q$, onde

$$||\hat{v}_{mult}||_\infty < \delta t\left(4 + \delta t B_{key}\right) V + \delta^2 t^2 B_{key}\left(B_{key} + t\right). \tag{2.1}$$

KeySwitch A função *KeySwitch* se apóia nos resultados das operações *PowersOf* e *WordDecomp* para eliminar parte do ruído do resultado intermediário $\hat{c}_{mult}$, transformando-o em um criptograma com ruído reduzido e que pode ser decifrado com a chave privada.

Esse algoritmo faz uso da chave de avaliação $evk = \left[PowersOf_{w,q}\left(f\right) + e + h \cdot s\right]_q$, onde $e, s \leftarrow \chi_{err}^{\log_w q}$ são vetores de polinômios amostrados da distribuição de erros χ_{err}. Como a multiplicação homomórfica depende dela, a chave evk precisa ser pública. Isso leva a uma suposição circular de segurança, assumindo que o YASHE continua seguro mesmo com a divulgação da chave evk.

O limitante superior para o ruído do criptograma resultante da função *KeySwitch* é dado pelo Lema 1.

Lema 1 (Ruído da *KeySwitch*)**.** *Seja $\hat{c}_{mult}$ o resultado intermediário da operação de multiplicação do* YASHE *e $\hat{v}_{mult}$ seu ruído. Seja evk a chave de avaliação e $c_{mult} = KeySwitch(\hat{c}_{mult}, evk)$. Então $fc_{mult} = \Delta m_1 m_2 t + v_{mult} \mod q$, tal que $\Delta = \lfloor q/t \rfloor$ e*

$$||v_{mult}||_\infty < ||\hat{v}_{mult}||_\infty + \delta^2 t\left(\log_w q\right) w B_{err} B_{key}. \tag{2.2}$$

2.4.2 Segurança

O criptossistema YASHE é um esquema probabilístico e tem sua segurança baseada no problema *RLWE*. Seus autores [9] oferecem uma variante que afirmam ter um uso mais prático, mas que possui dependência de um problema adicional, o *DSPR*. Este trabalho utiliza essa variante do esquema.

Além disso, o YASHE atinge o padrão de segurança *IND-CPA* [9] , assim como os esquemas de Paillier e ElGamal, sobre as premissas de que os problemas *RLWE* e *DSPR* são realmente difíceis e de que o YASHE se mantém *IND-CPA* mesmo se a chave *evk* for pública.

2.5 Teorema chinês do resto

O teorema chinês do resto, ou *CRT* [5], foi utilizado como ferramenta para simplificar a manipulação dos coeficientes polinomiais. Ele mapeia um polinômio com coeficientes inteiros arbitrariamente grandes em diversos polinômios com coeficientes inteiros arbitrariamente pequenos chamados polinômios residuais, ou simplesmente resíduos. Essa substituição permite que os coeficientes sejam manipulados utilizando uma aritmética mais simples e suportada nativamente pela arquitetura.

O funcionamento das funções direta e inversa do *CRT* pode ser visto nas Definições 13 e 14.

Definição 13 (Função direta do *CRT*)**.** *Sejam x um polinômio em R_q e $\{p_0, p_1, \ldots, p_{l-1}\}$ um conjunto de l coprimos. A representação residual de x pode ser calculada como:*

$CRT(x) = \{x \mod p_0, x \mod p_1, \ldots, x \mod p_{l-1}\}$.

Definição 14 (Função inversa do *CRT*, ou *ICRT*)**.** *Dado uma representação residual de x, $X = \{x_0, \ldots, x_{l-1}\}$; $\{p_0, p_1, \ldots, p_{l-1}\}$ o respectivo conjunto de primos; e $M = \Pi_{i=0}^{l-1} p_i$. Então $x \mod M$ pode ser calculado como:*

$ICRT(X) = \sum_{i=0}^{l-1} \left(\frac{M}{p_i}\right) \cdot \left(\left(\frac{M}{p_i}\right)^{-1} x_i \mod p_i\right) \mod M$.

Para o funcionamento correto do *ICRT* é necessário que o produto dos primos seja maior do que o maior possível coeficiente de um polinômio representado no domínio de resíduos, inclusive após uma operação de adição ou multiplicação. Ou seja, para a representação correta dos criptogramas do YASHE é preciso que $M = \prod_{i=0}^{l-1} p_i > n \cdot q^2$, onde n e q são os parâmetros que definem o anel de criptogramas.

No domínio de resíduos do *CRT* as operações de adição e multiplicação são definidas como na Definição 15.

Definição 15 (Adição e multiplicação no domínio do *CRT*)**.** *Sejam $x, y \in \mathbb{Z}_q$, $X = CRT(x)$ e $Y = CRT(y)$ seus respectivos conjuntos de resíduos representados com l primos e uma operação $\diamond$, adição ou multiplicação.*

Define-se: $X \diamond Y = \{(X_0 \diamond Y_0), (X_1 \diamond Y_1), \ldots, (X_{l-1} \diamond Y_{l-1})\}$.

[5] Acrônimo de *chinese remainder theorem*.

2.6 Multiplicação polinômial

Sejam p e q dois polinômios de grau N,

$$p(x) = a_0 + a_1 x + \ldots + a_{N-1} x^{N-1},$$
$$q(x) = b_0 + b_1 x + \ldots + b_{N-1} x^{N-1}.$$

Seu produto é definido como

$$\begin{aligned} p(x) \cdot q(x) &= c(x), \\ &= c_0 + c_1 x + \ldots + c_{2N-2} x^{2N-2}, \end{aligned} \tag{2.3}$$

onde

$$c_i = \sum_{\max\{0,i-(N-1)\} \leq k \leq \min\{i,N-1\}} a_k b_{i-k}. \tag{2.4}$$

Dessa forma, expandindo a somatória para cada c_i, a Equação 2.3 passa a ser escrita como

$$p(x) \cdot q(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)\, x + (a_0 b_2 + a_1 b_1 + a_2 b_0)\, x^2 + \cdots + a_{N-1} b_{N-1} x^{N-1}.$$

A quantidade de adições e multiplicações necessárias para o cálculo de c_i é $\Theta(N^2)$, o que compromete o desempenho com o aumento do tamanho dos operandos.

No contexto dos criptossistemas baseados no problema *RLWE*, como observado por Lindner e Peikert [42], a segurança é fortemente relacionada com o grau do anel de polinômios. Especificamente para o YASHE, Lepoint-Naehrig [40] utilizam parâmetros com N variando de 2^{11} até 2^{16} para atingir um padrão de segurança equivalente a $\lambda = 80$ *bits*. Dessa forma, a multiplicação de polinômios de grau alto é uma operação vital para esses esquemas, o que implica que melhorias de velocidade geram impacto considerável na velocidade da implementação.

O algoritmo da transformada rápida de Fourier [6], ou *FFT*, pode ser empregado na redução da complexidade computacional dessa operação [13]. Por meio dele é possível reduzi-la para $\Theta(N \log N)$, assintoticamente menor do que o algoritmo convencional. Uma versão alternativa desse algoritmo é a *NTT*, uma variação especializada para polinômios com coeficientes inteiros e que oferece a mesma redução de complexidade.

A adição polinomial se mantém correta quando executada no domínio de ambas transformadas. Sua aplicação se mantém a mesma, uma adição coeficiente-a-coeficiente, com a adição da conversão dos operandos ao seu custo.

2.6.1 Transformada discreta de Fourier

A transformada discreta de Fourier utiliza propriedades da raiz primitiva e N-ésima da unidade para construir seu domínio de operação.

[6] Do Inglês, *Fast Fourier transform*.

Definição 16 (Raízes primitivas e N-ésimas da unidade)**.** *Seja p um número primo e $N \geq 2$ um número inteiro. Um inteiro ω_N é uma raiz primitiva e N-ésima da unidade se e somente se:*

- $\omega_N^N = 1$,
- *Para x variando de 1 até $p-1$, $\omega_N^x \neq 1p$.*

A *DFT* é uma função bijetora de $\mathbb{R}^N$ em $\mathbb{C}^N$ e sua formulação é dada pela multiplicação do operando a pela matriz de Vandermonde V_N gerada por $\omega_N = e^{i\frac{2\pi}{N}}$, uma raiz primitiva e N-ésima da unidade baseada no corpo complexo, onde $i \in \mathbb{C}$ tal que $i^2 = -1$, como visto na Equação 2.5.

$$V_N \cdot a = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_N & \omega_N^2 & \dots & \omega_N^{N-1} \\ 1 & \omega_N^2 & \omega_N^4 & \dots & \omega_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_N^{(N-1)} & \omega_N^{2(N-1)} & \dots & \omega_N^{(N-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \hat{a}_2 \\ \vdots \\ \hat{a}_{N-1} \end{pmatrix} = \hat{a} \quad (2.5)$$

Quando o vetor a é o vetor de coeficientes de um polinômio $p(x) = a_0 + a_1 x + \cdots + a_{N-1}x^{N-1}$, é fácil ver que essa transformação é basicamente a avaliação de ω_N^i em $p(x)$, com $i \in \mathbb{Z}_{N-1}$.

Transformada inversa

A recuperação de a a partir de $\hat{a}$ é feita por meio da multiplicação do resultado da Equação 2.5 pela inversa V_N^{-1} da matriz de Vandermonde, definida na Equação 2.6.

$$V_N^{-1} = \frac{1}{N} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_N^{-1} & \omega_N^{-2} & \dots & \omega_N^{-(N-1)} \\ 1 & \omega_N^{-2} & \omega_N^{-4} & \dots & \omega_N^{-2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_N^{-(N-1)} & \omega_N^{-2(N-1)} & \dots & \omega_N^{-(N-1)^2} \end{pmatrix} \quad (2.6)$$

Preparação dos operandos para a aplicação da *DFT*

Os algoritmos propostos neste trabalho supõe, sem perda de generalidade, a aplicação da *DFT* em operandos de grau N, onde $N+1$ é uma potência de 2. Para a aplicação da transformada em polinômios com grau que não respeite essa condição, completa-se o vetor de coeficientes com zeros até que seu tamanho seja igual a menor potência de 2 maior do que $N+1$.

Além disso, para a recuperação do resultado de operações no domínio do *DFT* é preciso que o tamanho dos operandos seja adaptado de acordo com a expectativa do grau

do resultado. Enquanto a adição de dois polinômios de grau N resulta em um polinômio de grau N, a multiplicação de dois polinômios de grau N resulta em um polinômio de grau $2N$. Dessa forma, os vetores de coeficientes dos operandos precisam ser concatenados a um vetor complementar formado por zeros de tamanho N, simbolizando os coeficientes de x^N até x^{2N-1}, antes da aplicação da transformada direta.

Operações no domínio da *DFT*

No domínio da *DFT* a aplicação das operações de adição e multiplicação polinomial é realizada coeficiente-a-coeficiente. A descrição dessas operações é feita na Definição 17.

Definição 17 (Operações polinomiais no domínio da *DFT*)**.** *Sejam x e y polinômios de grau N, $X = DFT(x)$, $Y = DFT(y)$ e uma operação $\diamond$, adição ou multiplicação.*

Define-se: $x \diamond y = IDFT([X_0 \diamond Y_0, X_1 \diamond Y_1, \ldots, X_{N-1} \diamond Y_{N-1}])$.

Como entrada da transformada, serão considerados apenas vetores de coeficientes com o mesmo tamanho. Quando diferentes, toma-se o maior e completa-se o menor com zeros até que os tamanhos sejam iguais.

Feitos esses ajustes e respeitando a Definição 17, a multiplicação polinomial é apresentada no Algoritmo 1. Ele apresenta a rotina genérica para uma multiplicação polinomial realizada no domínio da *DFT* ou alguma variante. Ele recebe os operandos A e B e a quantidade n de coeficientes dos operandos.

Algoritmo 1: Algoritmo de multiplicação polinomial para transformadas baseadas na *DFT*.

Entrada: $n \in \mathbb{Z}$, $A \in \mathbb{Z}_{p_i}^n$, $B \in \mathbb{Z}_{p_i}^n$.
Saída : $C = A \cdot B$.

```
1 begin
2     A_Transformada ← TRANSFORMADA(A, 2n);
3     B_Transformada ← TRANSFORMADA(B, 2n);
4     C_Transformada ← MUL(A_Transformada, B_Transformada, 2n);
5     C' ← TRANSFORMADA INVERSA(C, 2n);
6     C ← SCALE (C', 2n);
7 end
```

As rotinas apresentadas nos Algoritmos 2 e 3 realizam, respectivamente, a multiplicação de dois vetores elemento-a-elemento e uma normalização por um inteiro n. Ambas

são utilizadas pelo Algoritmo 1 e recebem dois vetores A e B com n elementos.

Algoritmo 2: Rotina MUL, utilizada pelos Algoritmos 1 e 5.

Entrada: $A \in \mathbb{Z}_{p_i}^n$, $B \in \mathbb{Z}_{p_i}^n$,$n \in \mathbb{Z}$.
Saída : $C = \{A_i \cdot B_i\}_{i \in [0,n)}$

```
1 begin
2 |  C ← VETOR_VAZIO;
3 |  for i ∈ [0, n) do
4 |  |  C_i ← A_i · B_i;
5 |  end
6 end
```

Algoritmo 3: Rotina SCALE, utilizada pelos Algoritmos 1 e 5.

Entrada: $A \in \mathbb{Z}_{p_i}^n$, $s \in \mathbb{Z}$,$n \in \mathbb{Z}$.
Saída : $C = A/s$

```
1 begin
2 |  C ← VETOR_VAZIO;
3 |  for i ∈ [0, n) do
4 |  |  C_i ← ⌊A_i / s⌋;
5 |  end
6 end
```

Análise de complexidade

Cada elemento $\hat{a}_i = \mathrm{DFT}(a_i)$, para $a \in \mathbb{Z}^N$ e $i \in \{0, \cdots, N\}$, requer N multiplicações e $N-1$ adições para ser calculado. Dessa forma, a quantidade de operações aritméticas necessárias para o cálculo dos N elementos de $DFT(a)$, assim como para a aplicação da transformada inversa, é $\Theta(N^2)$.

A multiplicação polinomial, assim como a adição, é feita coeficiente-a-coeficiente. Dessa forma, sua complexidade computacional é $\Theta(N)$. Contudo, por conta do custo de aplicação da transformada, não há ganho assintótico para a multiplicação polinomial por meio da DFT.

2.6.2 Transformada rápida de Fourier

A implementação simples da DFT não apresenta vantagens para a multiplicação polinomial por conta do alto custo de aplicação da transformada. Para obter a redução de complexidade dessa operação utiliza-se a transformada rápida de Fourier, ou FFT [7]. Esta utiliza o mesmo princípio da DFT mas aplica um método de dividir-para-conquistar na redução da complexidade da aplicação da transformada para $\Theta(N \log N)$. Esse resultado

[7] Do inglês, *Fast Fourier transform*.

contribui para a redução de erros oriundos de operações de ponto-flutuante, já que há redução na quantidade dessas operações.

A diferença entre a *DFT* e a *FFT* mora na avaliação de cada uma das i-ésimas potências de ω_N pelo polinômio de entrada. Essa etapa utiliza uma estratégia do tipo dividir-para-conquistar proposta por Cooley- Tukey [14] para reduzir seu custo para $\Theta(\log N)$. Dessa forma, o custo total de aplicação da *FFT* para N pontos é $\Theta(N \log N)$, menor do que a complexidade quadrática da *DFT*.

O Anexo A.1 apresenta uma proposta de implementação da *FFT* por meio do algoritmo de Stockham de raio-R para a *FFT*.

2.6.3 *NTT*

A *DFT*, e por consequência a *FFT*, faz uso de um elemento do corpo complexo como raiz primitiva e N-ésima da unidade. A *Number-Theoretic Transform*, ou *NTT*, é uma variação sutil dessa transformada que utiliza corpos finitos para a obtenção dessa raiz.

A *NTT* precisa de um elemento ω_N de um corpo finito $\mathbb{Z}_p$ que satisfaça as condições da Definição 16 para operações módulo p. Com essa motivação, seja $p = k \cdot N + 1$, onde k é um inteiro positivo. Dessa forma, $p - 1 = k \cdot N$. Além disso, seja r uma raiz primitiva de p. Assim, define-se $\omega_N \equiv r^k \mod p$ como a raiz primitiva e n-ésima da unidade no corpo finito $\mathbb{Z}_p$.

Demonstração. Sejam N uma potência de 2, p um primo tal que $p = k \cdot (N+1)$, onde k é um inteiro positivo e r uma raiz primitiva de p. Além disso, seja $\omega_N \equiv r^k \mod p$. Assim,

$$\omega_N^N \equiv r^{kN} \equiv r^{p-1} \equiv 1 \mod p.$$

Além disso, $\omega_N^m \neq 1 \mod p$ se $m < N$ (por conta de ω_N ter sido construído em cima de uma raiz primitiva de p). Logo, ω_N é uma raiz primitiva e N-ésima da unidade de $\mathbb{Z}_p$. □

Dessa forma, substituindo ω_N por $r^k \mod p$ e as operações complexas por operações modulares sobre $\mathbb{Z}_p$, define-se a transformada *NTT*. O Anexo A.2 apresenta o código com o algoritmo de Stockham de raio-R para a *NTT*, convertido do algoritmo para a *FFT*.

2.6.4 Primos de Solinas

Para a construção da raiz ω_N utilizada pela *NTT*, foi proposto a adoção de um primo de Solinas [58]. Essa classe de primos é uma especialização dos primos de Mersenne [59], possuindo o formato $2^a \pm 2^b \pm 1$, onde $0 < b < a$ e $a, b \in \mathbb{Z}$.

Este trabalho decidiu usar o primo $P = 2^{64} - 2^{32} + 1$. Sobre ele, observa-se duas propriedades importantes:

1. $2^{64} \equiv 2^{32} - 1 \mod P$,

2. $2^{96} \equiv -1 \mod P$.

Por meio delas pode-se simplificar a redução modular em $\mathbb{Z}_P$, como definido no Lema 2.

Lema 2 (Resto modular em $\mathbb{Z}_{2^{64}-2^{32}+1}$)**.** *Seja um inteiro* $0 < a < 2^{128}$ *escrito como*

$$a = x_3 2^{96} + x_2 2^{64} + x_1 2^{32} + x_0, \text{ com } x_i \in \mathbb{Z}_{\geq 0} \text{ para } i \in \{0, 1, 2, 3\}.$$

Então, $a \equiv (x_1 + x_2)\, 2^{32} + x_0 - x_3 - x_2 \mod P$.

A prova de correção desse Lema é bastante simples e deriva das propriedades apresentadas.

Demonstração. Seja um inteiro $a < 2^{128}$ escrito como

$$a = x_3 2^{96} + x_2 2^{64} + x_1 2^{32} + x_0.$$

Sabendo que $2^{96} \equiv -1 \mod P$, então

$$\begin{aligned} a &\equiv x_3(-1) + x_2 2^{64} + x_1 2^{32} + x_0 \mod P, \\ &\equiv x_2 2^{64} + x_1 2^{32} + x_0 - x_3 \mod P. \end{aligned}$$

Por fim, uma vez que $2^{64} \equiv 2^{32} - 1 \mod P$, logo

$$\begin{aligned} a &\equiv x_2 \left(2^{32} - 1\right) + x_1 2^{32} + x_0 - x_3 \mod P, \\ a &\equiv x_2 2^{32} + x_1 2^{32} + x_0 - x_3 - x_2 \mod P, \\ &\equiv (x_1 + x_2)\, 2^{32} + x_0 - x_3 - x_2 \mod P. \end{aligned}$$

□

Diferente da *FFT*, o conjunto domínio da *NTT* é finito, sendo limitado pelo primo utilizado na geração de ω_N.

2.7 Redução modular de Barrett

O cálculo do resto é a operação mais importante ao se trabalhar com corpos finitos. Por conta do seu uso frequente, a velocidade é um fator importante.

Um gargalo dessa operação é a divisão, bastante utilizada em algoritmos como o de Euclides e expressivamente mais cara que a adição e multiplicação. Dessa forma, Barrett propôs [6] um algoritmo que otimiza o cálculo de $W \mod M$ quando $W < M^2$, $M \geq 3$ constante e não sendo uma potência de 2. Ele é apresentado no Algoritmo 4.

O resto R é definido como $R = W - M\frac{W}{M} \equiv W \mod M$, onde a divisão é uma operação sobre inteiros. Considerando que no contexto dos corpos finitos o cálculo do resto, ou redução modular, é feito com frequência e sempre pelo o mesmo módulo M e

supondo uma implementação eficiente da multiplicação de inteiros, pode-se pré-calcular o recíproco $N = 1/M$ e com isso obter R com duas multiplicações e uma subtração:

$$R = W - M\,(W \cdot N)\,. \tag{2.7}$$

O problema desse método é que N é um número real, possivelmente bem menor do que 1. Isso pode causar sérios problemas de implementação por conta da precisão da aritmética de ponto-flutuante. Como solução, Barrett propõe a representação de N como um inteiro, com a devida correção durante o cálculo do resto. Sua proposta é que isso seja feito por meio de uma multiplicação por alguma potência de 2. Usualmente, e como justificado no Anexo A.3, a aproximação N' de N é calculada como uma potência k de 4 tal que $4^k > M^2$.

Seja N' tal que $N \approx \frac{N'}{4^k}$. Re-escrevendo a Equação 2.7 como em 2.8 é fácil ver que se $N < \frac{N'}{4^k}$ então $M \cdot \left(\frac{W \cdot N'}{4^k}\right) > W$ e R assume um valor negativo, o que é conflitante com a definição de resto. Para evitar esse problema, costuma-se definir a aproximação N' como $N' = \lfloor N \cdot 4^k \rfloor$, para $k \in \mathbb{Z}$.

$$R = W - M \cdot \left(\frac{W \cdot N'}{4^k}\right). \tag{2.8}$$

A divisão por 4^k, assim como a multiplicação, pode ser aplicada utilizando uma operação de deslocamento binário para a direita. Assim o ganho obtido com essa estratégia é mantido, com o aumento de uma operação de deslocamento.

A escolha de k depende da precisão necessária para a aproximação de R. Barrett discute essa escolha em seu trabalho. Ele afirma que quanto mais precisa for a representação de N, mais cara será a multiplicação por W. Ao mesmo tempo, quanto menos preciso for a representação de N, maior será o erro de R.

A Equação 2.8 reduz $W \in \mathbb{Z}_{M^2}$ para $R \in \mathbb{Z}_{2M}$. Dessa forma, é necessário que seu resultado seja comparado com M. Se for maior, uma subtração por M é necessária.

A prova de correção do algoritmo de redução de Barrett é apresentada no Anexo A.3.

Algoritmo 4: Algoritmo de redução de Barrett.

Entrada: $M \in \mathbb{Z}$; $W \in \mathbb{Z}_{\mathbb{M}^2}$; $N' \in \mathbb{Z}$; $k \in \{x \in \mathbb{N} | 4^x > M^2\}$
Saída : $R = W \mod M$

```
1 begin
2 |  aux ← W · N';
3 |  aux ← aux >> 2k;
4 |  R ← W − aux;
5 |  if R > M then
6 |  |  R ← R − M
7 |  end
8 end
```

2.8 Resumo do capítulo

O Capítulo 2 apresentou conceitos fundamentais utilizados por este trabalho, como corpos finitos, polinômios irredutíveis e homomorfismo em criptossistemas. Além disso, apresentou-se o esquema YASHE assim como os problemas sobre os quais ele se baseia, *RLWE* e *DSPR*. Como estratégias para simplificar e otimizar a implementação da aritmética polinomial utilizada por esse esquema, foram apontados o teorema chinês do resto, ou *CRT*, além das transformadas *DFT*, *FFT* e *NTT*, que possibilitam a redução da complexidade computacional da multiplicação polinomial para $\Theta(N \log N)$. Por fim, também se ofereceu uma análise de segurança de criptossistemas homomórficos, com a definição dos níveis *IND-CPA*, *IND-CCA* e *IND-CCA2* e a justificativa da limitação desses esquemas ser *IND-CCA*.

Capítulo 3

Implementação da biblioteca cuYASHE

A biblioteca cuYASHE foi desenvolvida durante a execução deste trabalho. Ela foi escrita em C++, utiliza CUDA para acelerar a aritmética polinomial, *CRT* para simplificar a manipulação dos operandos pela *GPU* e compara as transformadas *FFT* e *NTT* para diminuir a complexidade de tempo de uma multiplicação polinomial. Seu código está disponível ao público [3] sob uma licença GNU GPLv3 [60].

As operações sobre inteiros grandes executadas na *CPU* são realizadas pela biblioteca *NTL*. Essa decisão possibilitou maior afinco na implementação para *GPU*.

Uma vez que CUDA não oferece suporte e não há bibliotecas de terceiros com esta finalidade, as operações sobre inteiros grandes executadas na *GPU* tiveram de ser implementadas de forma independente. Para isso, a biblioteca RELIC [4] foi utilizada como base e as rotinas necessárias (adição, subtração, multiplicação, divisão e resto modular) foram adaptadas para execução na *GPU*.

Como a YASHE trabalha com a aritmética de corpos finitos, a cuYASHE opera sobre inteiros do tipo *unsigned* de 64 *bits*. Dessa forma, mesmo que os polinômios residuais venham a ter seus coeficientes limitados a primos menores, por exemplo de 32 *bits*, essa decisão simplifica o tratamento de *overflows*. Além disso, esse tamanho de palavra aumenta a compatibilidade da biblioteca com a implementação da *NTT*, que utiliza um primo de Solinas de 64 *bits*, descrito na Seção 2.6.4.

Para maximizar os ganhos com o uso das transformadas descritas na Seção 2.6, a cuYASHE trabalha com resíduos dentro do domínio da transformada. Dessa forma, mapeia-se polinômios aos resíduos do *CRT* e, em seguida, aplica-se a transformada *FFT* ou *NTT* em cada um desses resíduos. Ao manter os operandos nesses domínios, pode-se executar adições e multiplicações com complexidade linear.

A cuYASHE foi construída sobre o paradigma de orientação a objetos. Ela representa polinômios-genéricos e polinômios-criptogramas com as classes *Polynomial* e *Ciphertext*, respectivamente, como visto no diagrama simplificado de classes apresentado na Figura 3.1.

Na primeira versão da biblioteca, havia uma relação de herança entre essas classes. A *Ciphertext* era herdeira da *Polynomial*. Contudo, durante a implementação, notou-se uma perda de velocidade causada pelo custo desse tipo de polimorfismo em C++. Essa perda foi da ordem de 1 ms por operação e era causada pelo custo associado a chamada de uma função da superclasse por uma instância da classe filha.

O estado da arte do YASHE apresenta tempos de execução da ordem de milisegundos. Dessa forma, a perda de velocidade observada se mostrou inaceitável. O modelo de classes foi refatorado e essa relação de herança foi desfeita.

As distribuições probabilísticas requeridas pelo criptossistema, estreita e Gaussiana discreta, foram concentradas na classe *Distribution*. Como desejou-se que a aritmética polinomial fosse executada exclusivamente na *GPU*, a implementação dessas distribuições foi feita utilizando a biblioteca cuRAND [45].

A cuRAND é uma biblioteca mantida pela NVIDIA com a proposta de oferecer uma *API* para a geração de números pseudo-aleatórios. A cuYASHE utiliza essa biblioteca para amostrar e armazenar esses números diretamente na *GPU*. Dessa forma, pode-se realizar a amostragem dos polinômios já no domínio do *CRT* e sem o custo de cópia de dados entre as memórias.

O amostrador da distribuição estreita foi implementado por meio da distribuição uniforme. Contudo, como não há suporte pela cuRAND para amostragem da distribuição Gaussiana discreta, utilizou-se o método de arredondamento proposto por Peikert [50] para converter as amostras da Gaussiana contínua, que é suportada pela biblioteca.

3.1 CUDA

Este trabalho teve como proposta a aplicação de paralelismo por meio de *GPGPUs*. Para isso, escolheu-se CUDA como a plataforma base.

Apesar de ter sofrido uma evolução considerável nas últimas revisões, CUDA ainda é conhecida por exigir do programador conhecimentos profundos de seu funcionamento, ele precisa entender suas capacidades e limitações para obter ganhos reais de velocidade. O mesmo não acontece com bibliotecas de paralelismo em *CPUs*. Nesse caso, o entendimento de como o *hardware* processará os dados não é obrigatório para a obtenção de ganho.

Assim, uma vez que o conhecimento da plataforma se mostrou de grande importância para a otimização da aritmética da cuYASHE, esta Seção discute detalhes importantes de seu funcionamento.

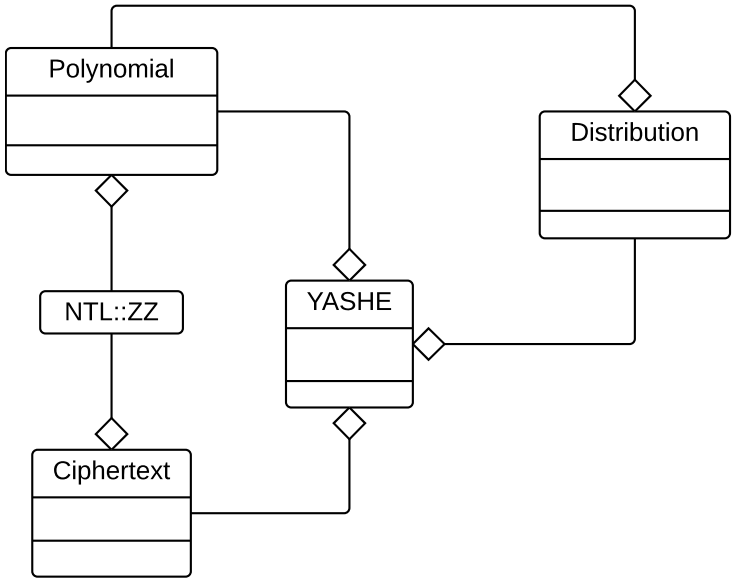


Figura 3.1: Diagrama de classes simplificado da CUYASHE. A classe *Polynomial* é utilizada para a representação de polinômios-genéricos. A *Ciphertext*, por sua vez, representa polinômios-criptogramas. A classe YASHE contém a implementação do criptossistema e agrega instâncias dessas duas classes, além da *Distribution*, responsável pelas distribuições probabilísticas requeridas.

3.1.1 Contexto

Durante as primeiras décadas da computação, aumento de velocidade de dispositivos computacionais era sinônimo de aumento da frequência do *clock*. Essa estratégia funcionou satisfatoriamente até o começo dos anos 2000.

Hennessy e Patterson [36] estimam que, entre 1978 e 1986, a frequência do *clock* aumentou a uma taxa anual de 15%, com aumento de desempenho da ordem de 25%. Entre 1986 e 2003, o desempenho das *CPUs* cresceu em 52% por ano, enquanto a frequência do *clock* acompanhou a 40%. A partir de 2003, entretanto, essa frequência estagnou, apresentando aumento de menos de 1% ao ano.

O aumento da frequência do *clock* de um processador implica no aumento do consumo energético e, consequentemente, no aumento da emissão de calor. Dessa forma, no começo da década de 2000 essa frequência precisou ser estabilizada em torno de 3 GHz. O consumo energético e a emissão de calor se tornaram proibitivos e precisou-se buscar alternativas. Como solução, a indústria se voltou para o processamento paralelo [56]. Assim, mesmo sem a elevação da frequência do *clock*, o desempenho pôde continuar crescendo a uma taxa anual de 25%.

No final de 2006 a NVIDIA apresentou CUDA [1], uma plataforma de computação

[1] Acrônimo de *Compute Unified Device Architecture*.

paralela de propósito geral que transforma *GPUs* [2] em *GPGPUs* [3]. Isso é obtido através de uma *API* que permite ao programador aplicar a *GPU* em propósitos não apenas gráficos.

Em seu princípio, o mercado de placas gráficas tinha como objetivo atender a demanda de consumidores de jogos. Os *softwares* consumidos por esse nicho são caracterizados por requererem processamento não-interdependente de grandes quantidades de dados. Nesse contexto, os dados consistiam de parâmetros de entrada que seriam combinados para calcular uma cor para cada *pixel* da tela. Com o tempo, percebeu-se que esse conjunto de entrada e saída poderia ser estendido para além do cálculo de cores.

O modelo de processamento da CUDA é similar ao de um processador vetorial. Seu funcionamento envolve a manipulação de conjuntos de dados em paralelo, ao contrário do que acontece em processadores escalares onde se opera sobre dados individualmente. Dessa forma, a melhor aplicação de *GPGPUs* é em problemas baseados em dados com pouca ou nenhuma interdependência.

Processamento gráfico é um exemplo de problema ideal para ser resolvido por *GPUs*. Nesse problema, os dados de entrada são naturalmente divididos em blocos de *pixels*, que podem ser mapeados em *threads*. Outros exemplos podem ser encontrados em simulações físicas, geofísicas, financeiras, biológicas e inclusive na criptografia.

3.1.2 Modelo de programação

"If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?",

Seymour Cray, Father of the Supercomputer [56].

A programação em CUDA depende de três conceitos fundamentais: *kernels*, hierarquia de *threads* e de memórias. Manipulando essas abstrações o programador é capaz de explorar o poder computacional de *GPGPUs*. Além disso, esse modelo de programação auxilia o desenvolvimento de forma a otimizar algoritmos para execução paralela.

Kernels

Em CUDA, a execução de um algoritmo é iniciado por meio de uma função chamada *kernel*. Essa função serve como uma espécie de trilho, guiando toda a execução paralela na *GPU*. Ela configura os recursos a serem requeridos do *hardware*, lança todos os *threads* e os descarta ao final de seu processamento.

[2] Acrônimo de *Graphic Processor Units*.
[3] Acrônimo de *General Purpose Graphic Processor Units*.

CUDA *Threads*

Threads são fluxos de processamento independente e passíveis de serem executados em paralelo. No caso de *GPUs*, sua execução é feita por multiprocessadores de fluxo [4], ou *SM* [64].

No contexto da plataforma CUDA, costuma-se utilizar o termo CUDA *Thread* ao invés de simplesmente *thread*. Ao empregar um nome exclusivo para um conceito que transcende *GPUs*, a plataforma lembra os programadores que, apesar de semelhantes, existem diferenças profundas no funcionamento de *threads* executados por *CPUs* e *GPUs*. Por simplicidade, neste trabalho não haverá distinção entre os termos, a menos que seja explicitado.

Não há comunicação direta entre CUDA *Threads* como acontece por exemplo em *MPI* [27]. Essa comunicação só pode ser feita por meio de memórias compartilhadas. Por isso, algoritmos executados em CUDA precisam ter baixa dependência de dados e de resultados entre *threads*. Caso contrário, os custos de sincronismo podem arruinar o desempenho.

O modelo de execução de CUDA *Threads* segue uma arquitetura do tipo *SIMD* [5], ou *SIMT* [6] como chamado pela NVIDIA. Nessa arquitetura *threads* são divididos em grupos chamados *warps*, de forma que cada *warp* executa uma única instrução por vez. A Seção 3.1.3 aprofunda a discussão sobre essa abstração.

Portanto, a comparação de poder de processamento entre CUDA *Threads* e *threads* executados pela *CPU* torna-se sem sentido. CUDA *Threads* são processados sempre em *warps*, enquanto a *CPU* processa *threads* individualmente. De fato, uma comparação de desempenho mais adequada seria entre *warps* e *threads* executados pela *CPU*.

Como visto, CUDA *Threads* e *threads* executada pela *CPU* tem características bastante distintas. Dessa forma, seguindo a citação de Seymour Cray, é importante que isso seja levado em consideração durante uma implementação. É necessário avaliar o problema a ser resolvido para só então decidir qual dos modelos o resolveriam de maneira mais eficiente.

Hierarquia de *threads* Cada CUDA *Thread* faz parte de um bloco de *threads*. Dentro desses blocos, são atribuídas três coordenadas como índice de cada *thread*, enunciadas como x, y e z. Da mesma forma, cada bloco faz parte de uma grade de blocos, identificados com até duas dimensões de coordenadas x e y. Uma representação dessa estrutura pode ser vista na Figura 3.2.

CUDA requer que todos os *threads* de um bloco sejam processados por um único *SM*. Dessa forma, os recursos de *hardware* disponíveis em cada um desses multiprocessadores limitam as dimensões do bloco. Em *GPUs* com suporte a revisões entre 2.0 e 5.2 da CUDA, um bloco pode ser constituído por no máximo 1024 *threads*.

[4] Do inglês, *Streaming Multiprocessors*.
[5] Acrônimo do inglês *Single Instruction Multiple Data*.
[6] Acrônimo do inglês *Single Instruction Multiple Threads*.

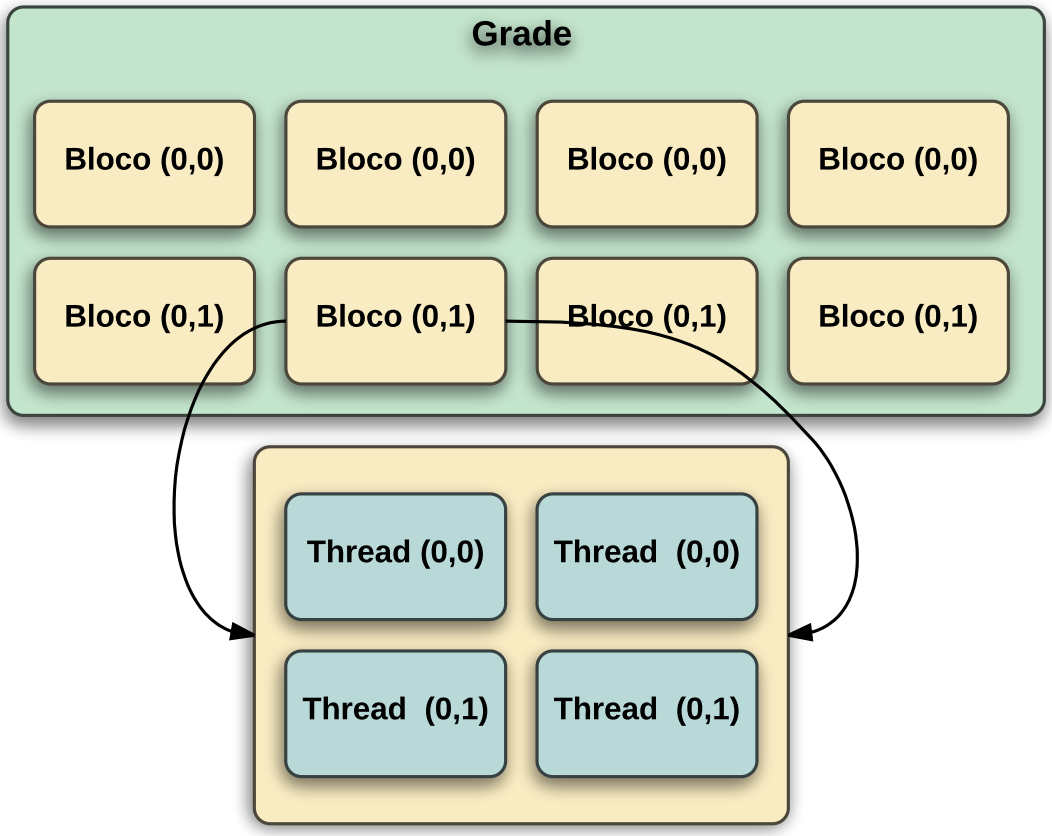


Figura 3.2: Esquema estrutural de *threads*. Neste exemplo, blocos são constituídos por 4 *threads*, enquanto a grade possui 8 blocos divididos em duas dimensões.

Quando um *kernel* é executado com múltiplos blocos, costuma-se definir um índice global para cada *thread*. Dessa forma, pode se simplificar a atribuição de trabalho para um *thread*, além de possibilitar otimizações no acesso à memória.

Não há regra quanto a maneira com que esse índice é calculado. Sugere-se que ele ordene os *threads* da forma mais adequada ao algoritmo implementado no *kernel*. Usualmente, costuma-se usar como parâmetros as dimensões da grade e do bloco, o índice do bloco na grade e o índice do *thread* no bloco.

As Equações 3.1 e 3.2 são opções simples e diretas para o cálculo desse índice em configurações de *kernels* com grades unidimensionais e blocos unidimensionais ou bidimensionais, respectivamente. Elas dependem da dimensão e índice do bloco, além do índice do *thread* no bloco.

$$\text{tid} = \text{blockIdx}.x \times blockDim.x + \text{threadIdx}.x \tag{3.1}$$

Bloco e grade unidimensionais

$$\text{tid} = \text{blockIdx}.x \times \text{blockDim}.x \times \text{blockDim}.y + \text{threadIdx}.y \times \text{blockDim}.x + \text{threadIdx}.x \tag{3.2}$$

Bloco bidimensional e grade unidimensional

Quando a grade de blocos deixa de ser unidimensional, torna-se necessário redefinir o índice do bloco para só então poder gerar um índice global para seus *threads*. As Equações 3.3 e 3.4 apresentam sugestões para esse cálculo.

$$\begin{aligned} \text{bid} &= \text{blockIdx}.y \times \text{gridDim}.x + \text{blockIdx}.x \\ \text{tid} &= \textit{bid} \times \text{blockDim}.x + \text{threadIdx}.x \end{aligned} \tag{3.3}$$

Bloco unidimensional e grade bidimensional

$$\begin{aligned} \text{bid} &= \text{blockIdx.y} \times \text{gridDim.x} + \text{blockIdx.x} \\ \text{tid} &= \text{bid} \times \text{blockDim.x} \times \text{blockDim.y} + \text{threadIdx.y} \times \text{blockDim.x} + \text{threadIdx.x} \end{aligned} \tag{3.4}$$

Bloco e grade bidimensionais

A implementação da cuYASHE por simplicidade padronizou o uso de blocos e grades unidimensionais.

Memórias

O modelo de memória proposto por CUDA propõe uma separação entre a memória da *GPU* e a memória principal da máquina. Mesmo que eventualmente não exista uma separação física (no caso de uma *GPU* integrada na placa mãe, por exemplo), ainda assim haverá a necessidade do programador explicitar a cópia dos dados entre as memórias.

Todo CUDA *Thread* possui acesso de leitura e escrita à três tipos principais de memória residentes na *GPU*:

Memória local Cada *thread* possui uma memória com escopo exclusivo ao *thread*. Ela possui a menor latência entre as memórias.

Memória compartilhada Todos os *threads* de um bloco compartilham um espaço de memória. A latência é um pouco maior do que a memória local.

Memória global A memória global pode ser lida e escrita por todos os *threads*. Essa também é a única das memórias citadas que possui privilégios de acesso para *threads* na *CPU*. Sua latência é muito maior que a memória local ou compartilhada.

Além dessas, também existem as memórias constante e de textura [7]. Ambas possuem permissão de apenas leitura para CUDA *Threads* e de leitura e escrita para *threads* na *CPU*.

A memória constante, como o nome sugere, é voltada para dados que serão lidos mas nunca alterados durante a execução de um *kernel*. Os dados dessa memória são mantidos na *cache*, o que implica que seu tamanho é bastante limitado. Atualmente, a memória constante é limitada a 64kB.

A memória de textura é fruto do foco original da *GPU*: processamento gráfico. Ela é, na verdade, apenas uma abstração da memória global que privilegia acessos com localidade espacial. Dessa forma, a *cache* é alimentada com posições não necessariamente consecutivas mas próximas de uma leitura anterior.

No caso das memórias locais e compartilhadas, o tempo de vida dos dados armazenados é igual ao tempo de vida do *thread*. Para todas as outras, o tempo de vida é igual ao tempo de execução do programa.

A hierarquia de memória descrita pode ser vista na Figura 3.3.

3.1.3 *Warps*

A arquitetura da CUDA é do tipo *SIMT* [8], onde carrega-se uma única instrução para operar sobre um conjunto de *threads*. Esse conjunto é chamado *warp* e é constituído por 32 *threads* [9].

Durante a execução de uma função *kernel*, cada bloco de *threads* é distribuído pelos *SMs* disponíveis. Dentro de um *SM* os blocos são segmentados em *warps* e organizados em uma fila. Com a finalização do processamento de um bloco os recursos são alocado para um novo bloco de *threads* e assim a execução segue até o final.

Cada *SM* possui um conjunto de *núcleos* CUDA [10]. Esses *cores* consomem elementos da fila de *warps*, de forma que caso todos os *threads* de um bloco já tenham sido mapeados a um *núcleo* CUDA e ainda haja *núcleos* CUDA disponíveis, outro bloco mapeado àquele *SM* é escolhido e o processo segue da mesma maneira. Não há garantias sobre a ordem de processamento.

No início do processamento de um *warp* é realizado o *fetch* de instruções. Esse é feito não para os *threads* individualmente mas para todo o *warp*. Em caso de desvio condicional dentro de um *warp* ele é segmentado em duas partes: uma com os *threads* que tomaram o desvio e outra com os que não tomaram. Quando o desvio é resolvido esses *threads* são sincronizados, o *warp* original é reconstruído e o processamento segue normalmente.

Um *núcleo* CUDA possui recursos para tratar um *warp* completo. Assim, o desvio descrito implica em desperdício de recursos. O *core* não poderá aproveitar os recursos oci-

[7] Do inglês *texture memory*.

[8] A *SIMT* é uma especialização da *SIMD*.

[9] Historicamente, *GPUs* com suporte a CUDA sempre utilizaram *warps* de 32 *threads*. Apesar disso, esse valor pode ser alterado em futuras revisões da plataforma.

[10] A quantidade de *núcleos* CUDA por *SM* pode variar de acordo com a *GPU*. Por exemplo, a GeForce GTX Titan Black possui 192 *núcleos* CUDA por *SM*, enquanto a GeForce GTX 980 possui 128.

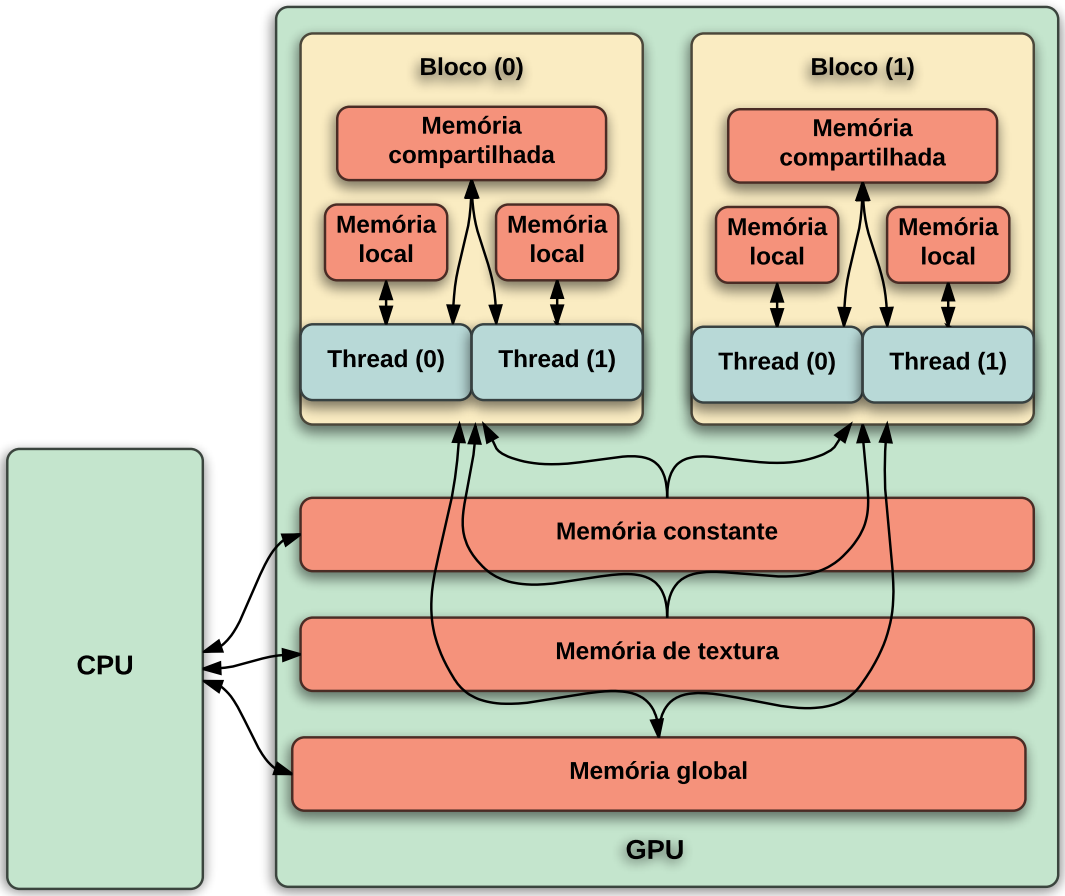


Figura 3.3: Esquema da hierarquia de memória, exposta ao programador pela plataforma CUDA. Toma-se como exemplo a visibilidade das memórias para uma grade unidimensional de dois blocos, cada um também unidimensional com dois threads.

osos com outro *warp* e ainda gastará o dobro do tempo previsto com o *warp* em execução. Isso implica no atraso do processamento dos *warps* daquele bloco e, consequentemente, de todos os blocos alocados para o *SM*.

Dessa forma, a ocorrência de desvio condicional dentro de um *warp* precisa ser tratado com bastante cuidado para não trazer danos ao desempenho. É importante que o *kernel* seja construído de forma a eliminar as divergências quando possível ou alinhar os *threads* que divergem em diferentes *warps*, quando a divergência for necessária. Essas duas estratégias reduzem ou eliminam prejuízos de desempenho.

3.1.4 Acessos coalescidos à memória

Uma das características mais importantes para se preocupar durante a programação em CUDA é a coalescência dos acessos à memória global [11]. Instruções de leitura e escrita realizadas por um *warp* são coalescidas pela *GPU* em uma única transação quando

[11] Do inglês, *coalescing of global memory accesses*.

respeitam certos requisitos. Dessa forma, pode-se reduzir consideravelmente o tráfego no barramento de memória, além de possíveis travamentos no *pipeline*. É necessário lembrar que acessos à memória global são consideravelmente mais lentos que todos os outros. De fato, em seu manual de melhores práticas [47], a NVIDIA argumenta que otimizações nesse tipo de acesso tem alta prioridade.

Quando um *warp* realiza um acesso de leitura ou escrita na memória global, as requisições de dados de cada *thread* são coalescidas em transações. Essas transações alimentam as linhas da *cache* com 32, 64 ou 128 *bytes* em posições consecutivas. Dessa forma, um padrão de acesso à memória global é considerado eficiente se maximizar o aproveitamento de dados em cada transação. Por exemplo, seja um padrão de acesso de forma que as requisições de um *warp* só possam ser coalescidos para 4 *bytes* consecutivos. Serão necessárias 8 transações de 32 *bytes* para satisfazer de todos os *threads* do *warp*, sendo que cada transação preencherá a *cache* com 28 *bytes* desnecessários. Deste modo, a taxa de transferência será 8 vezes menor do que um padrão de acesso onde as requisições pudessem ser coalescidas em uma única transação.

O caso mais simples de otimização de acessos coalescidos ocorre quanto o k-ésimo *thread* em um *warp* acessa a k-ésima palavra de uma linha de *cache*. A Figura 3.4 demonstra esse caso para *GPUs* que utilizem a *cache* L1 com linhas de 128 *bytes*.

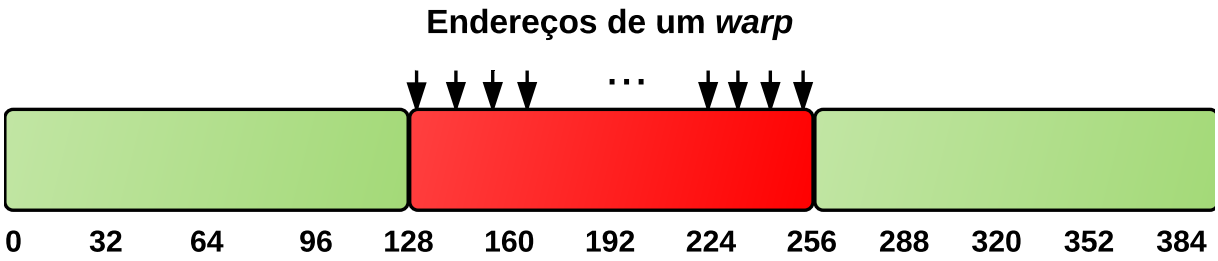


Figura 3.4: Ilustração de um padrão de acesso à memória global com coalescência ótima.

Outra questão importante a se chamar atenção é o alinhamento dos acessos. Como pode ser visto na Figura, o primeiro *thread* do *warp* acessa a primeira palavra da linha de *cache*. Caso esse tipo de alinhamento não seja satisfeito o acesso coalescido pode ser comprometido.

3.2 Manipulando inteiros grandes

Operações sobre inteiros com precisão maior do que 64 *bits* podem ser bastante caras e adicionar um grau de complexidade considerável ao trabalho. CUDA não possui suporte nativo para essas operações e nem mesmo se encontram disponíveis bibliotecas numéricas alto nível que implementem essa aritmética para *GPUs* como a *NTL* [57], FLINT [35] ou *GMP* [32].

Com o intuito de compensar essa lacuna, aplicou-se o *CRT* para a conversão dos operandos em polinômios com coeficientes pequenos. Assim, a CUYASHE tem a necessidade

de que sua aritmética polinomial seja implementada de forma a replicar todas as operações igualmente por esses resíduos, conforme visto na Definição 15. Essa estratégia implica na troca do custo computacional de aplicar uma operação sobre polinômios com coeficientes grandes, que requerem aritmética de inteiros grandes implementada em *software*, pelo custo de aplicar essa mesma operação diversas vezes sobre polinômios com coeficientes arbitrariamente pequenos e suportados por instruções nativas da *GPU*.

O uso da aritmética de inteiros grandes se limitou a aplicação do *CRT* e do *ICRT* feita na *GPU*.

3.3 Aritmética polinomial

A aritmética polinomial implementada na cuYASHE é executada exclusivamente na *GPU*. Para isso, foi definido um modelo de CUDA *Threads* que otimiza o acesso à memória e paraleliza as operações, tanto do ponto de vista dos coeficientes quanto dos polinômios residuais. Isso permitiu absorver o custo do aumento de operandos causado pela necessidade de se replicar a operação pelos polinômios residuais, além de também trazer ganhos no tempo de execução das operações do YASHE.

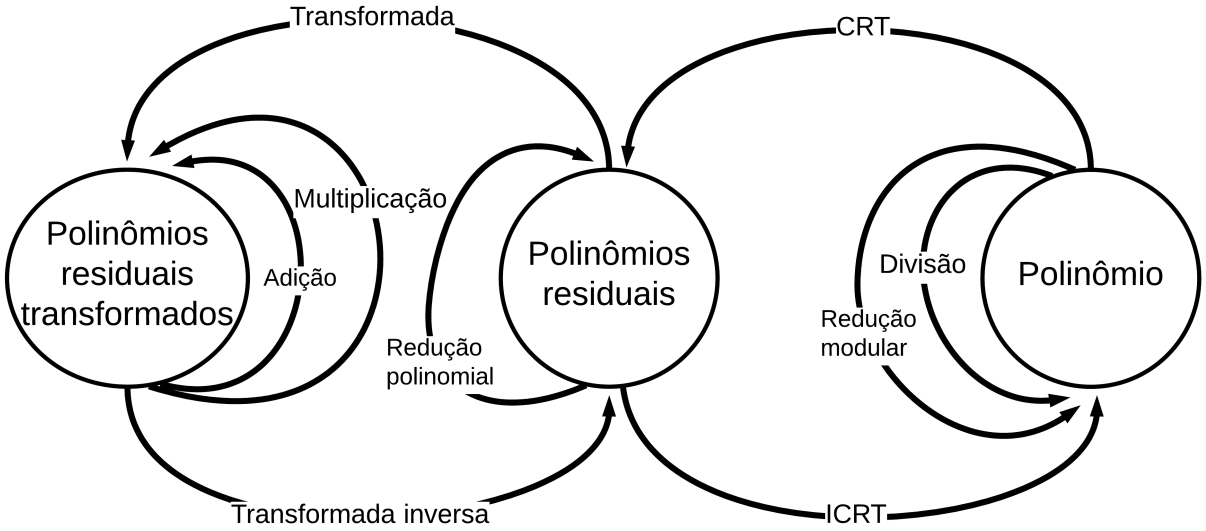


Figura 3.5: Diagrama da máquina de estados de objetos do tipo *Polynomial* ou *Ciphertext*. As operações de divisão polinomial e redução modular são aplicadas no estado original dos operandos. A redução polinomial, por sua vez, é aplicada apenas sobre os polinômios residuais. Para a adição e multiplicação polinomial há a necessidade de se aplicar previamente alguma das transformadas citadas, *FFT* ou *NTT*, sobre os resíduos.

A aritmética polinomial da cuYASHE faz uso de adições, multiplicações e deslocamentos binários para implementar versões eficientes das operações de subtração, divisão e resto.

Adição e subtração

Na adição, cada bloco de CUDA *Threads* é composto por uma única dimensão de 32 *threads*. Dessa forma, a grade de blocos é montada com $\left\lceil \frac{N \cdot M}{32} \right\rceil$ blocos para um polinômio de grau $N-1$ mapeado em M polinômios residuais.

O *kernel* de adição opera sobre dois vetores de inteiros, elemento-a-elemento, compostos pela concatenação dos coeficientes dos resíduos de cada operando. A subtração é realizada da mesma maneira, apenas substituindo a operação.

Por executarem operações muito simples, os *kernels* são resolvidos rapidamente e sem qualquer divergência condicional. Além disso, o modelo de armazenamento dos operandos provê naturalmente acessos coalescidos à memória.

Multiplicação

Foram abordadas duas estratégias para a multiplicação polinomial. A primeira faz uso da transformada NTT, cuja implementação é baseada na formulação de Stockham, apresentada no Anexo A.2. A segunda estratégia usa a FFT, fornecida pela biblioteca cuFFT. A Seção 3.6 apresenta uma discussão aprofundada sobre cada uma.

O uso de qualquer uma dessas transformadas faz com que a complexidade da multiplicação polinomial entre polinômios de grau $N-1$ seja $\Theta(N \log N)$, considerando o custo de aplicação da transformada. A cuYASHE mantém os operandos no domínio da transformada, dessa forma pode-se diluir esse custo por uma série de operações consecutivas.

A vantagem da NTT sobre a FFT, no contexto da YASHE, vem da maior compatibilidade com a aritmética modular. Pela FFT operar no corpo dos complexos, há a necessidade da conversão dos coeficientes a partir do conjunto dos inteiros. Essas conversões podem gerar erros de precisão, o que limita o tamanho dos primos utilizados pela CRT e, consequentemente, aumenta o custo das operações polinomiais. A NTT, por outro lado, opera sobre corpos finitos assim como a YASHE. Isso descarta a necessidade de qualquer conversão.

A aplicação das duas transformadas na multiplicação polinomial é descrita pelo Algoritmo 1.

Divisão e resto por polinômios ciclotômicos

O cálculo do quociente e do resto em uma divisão de polinômios pode ser consideravelmente caro. Dessa forma, é importante que se explore as propriedades dos operandos a fim de encontrar uma formulação que simplifique essas operações.

O YASHE constrói seu anel de criptogramas R_q por meio de polinômios ciclotômicos. Essa classe de polinômios tem a propriedade de que o n-ésimo polinômio ciclotômico é dado por $\Phi_n(x) = x^{n/2} + 1$, para todo n potência de 2. Dessa forma, a divisão por esses polinômios pode ser feita como no Lema 3, requerindo apenas um deslocamento de $\frac{n}{2}$ posições dos coeficientes de maior grau de $P(x)$ para o cálculo do quociente e de uma

subtração para o cálculo do resto.

Lema 3 (Divisão e resto pelo n-ésimo polinômio ciclotômico)**.** *Sejam $\Phi_n(x)$ o n-ésimo polinômio ciclotômico e $P(x) = \sum_{i=0}^{m-1} a_i x^i$, com $m > n$ e n potência de 2. Então, o quociente $Q(x)$ e o resto $R(x)$ da divisão de $P(x)$ por $\Phi_n(x)$, são dados por:*

$$Q(x) = \frac{P(x)}{x^{\frac{n}{2}}},$$

$$R(x) = P(x) - Q(x) \cdot \Phi_n(x).$$

Prova do Lema 3. Sejam o polinômio $P(x)$ de grau m e $\Phi_n(x)$ o n-ésimo polinômio ciclotômico, com $m > n$ e n potência de 2. Além disso, sejam $Q(x)$ e $R(x)$ respectivamente o quociente e o resto da divisão de $P(x)$ por $\Phi_n(x)$. Dessa forma,

$$\begin{aligned} P(x) &= Q(x) \cdot \Phi_n(x) + R(x), \\ &= Q(x) \cdot \left(x^{\frac{n}{2}} + 1\right) + R(x), \\ &= Q(x) \cdot x^{\frac{n}{2}} + Q(x) + R(x). \end{aligned} \tag{3.5}$$

Logo, lembrando que o quociente da divisão $\frac{Q(x)+R(x)}{x^{\frac{n}{2}}}$ é 0, uma vez que $Q(x)$ e $R(x)$ tem grau necessariamente menor que $\frac{n}{2}$, para $n > 0$, então

$$\frac{P(x)}{x^{\frac{n}{2}}} = \frac{Q(x) \cdot x^{\frac{n}{2}} + Q(x) + R(x)}{x^{\frac{n}{2}}} = Q(x).$$

Substituindo $Q(x)$ na Equação 3.5,

$$R(x) = P(x) - Q(x) \cdot \Phi_n(x),$$

o que encerra a demonstração do Lema 3. □

O cálculo de Q e R pode ser implementado com a simples manipulação dos coeficientes de potências menores e maiores que $n/2$, além de uma operação de subtração polinomial.

O cálculo do resto por $\Phi_n(x)$ é essencial para o YASHE, uma vez que a redução polinomial é uma operação frequente por conta do anel R_q.

Divisão e resto por q

No YASHE os coeficientes dos polinômios-criptogramas moram em um anel gerado por um primo q. Assim, a aritmética desse criptossistema requer a execução frequente de uma operação de redução modular. A decifração, em especial, também requer uma divisão por q.

Como otimização, este trabalho propõe que q seja um primo com um formato especial que simplifique essas operações. A cuYASHE foi implementada com q sendo um primo

de Mersenne [59]. Esses primos podem ser escritos como $2^n - 1$, para n inteiro maior ou igual a 2. Dessa forma, a divisão e a redução modular por q podem ser implementadas por meio de deslocamentos binários, como proposto no Lema 4.

Lema 4 (Divisão e resto por um primo de Mersenne)**.** *Sejam n um inteiro maior ou igual a 2, q um primo de Mersenne tal que $q = 2^n - 1$ e x um inteiro em $\mathbb{Z}_{q^2}$. Então, o quociente $Q(x)$ e o resto $R(x)$, da divisão de x por q, são dados por:*

$$Q(x,n) = \begin{cases} (x \cdot 2^{-n}), & se\ (x \cdot 2^{-n}) < q, \\ (x \cdot 2^{-n}) - q, & caso\ contrário. \end{cases}$$

$$R(x,q,n) = Q(x,n) + (x \text{ AND } q).$$

Operações no domínio do *CRT* se resumem a adições, subtrações e multiplicações. A redução modular não pode ser facilmente aplicada sobre os polinômios residuais. Como solução, encontrou-se trabalhos na literatura propondo o uso de uma variação do *CRT*, chamada *RNS* [12] [5]. Devido à complexidade de implementação e ao desempenho oferecido, essa solução não foi utilizada por este trabalho. Como alternativa, a CUYASHE reconstrói o polinômio com a *ICRT* e aplica um algoritmo convencional de redução.

3.4 Localidade de memória

Uma vez que seja feita a cópia de um polinômio para a memória global da *GPU*, todas as operações subsequentes mantém a localidade dos dados sem em momento algum requerer a cópia para a memória principal da máquina. A Figura 3.5 demonstra essa dinâmica apresentando o diagrama da máquina de estados de objetos do tipo *Polynomial* ou *Ciphertext*.

Os coeficientes de um polinômio são organizados em um vetor de inteiros grandes e manipulados com funções oriundas da biblioteca RELIC adaptadas para CUDA. Seguindo o mesmo padrão de armazenamento, os polinômios residuais também tem seus coeficientes armazenados na memória global da *GPU* durante toda a execução. Esse armazenamento é realizado na forma da concatenação dos coeficientes de cada resíduo em um grande vetor de inteiros *unsigned* de 64 *bits*. Essa estrutura de armazenamento contribui para que as operações polinomiais tirem proveito do acesso coalescido de memória.

Com o suporte de operações sobre inteiros grandes em CUDA, foi possível o cálculo da *ICRT* e da redução modular diretamente na *GPU*. As vantagens dessa estratégia são apresentadas na Seção 4.3.

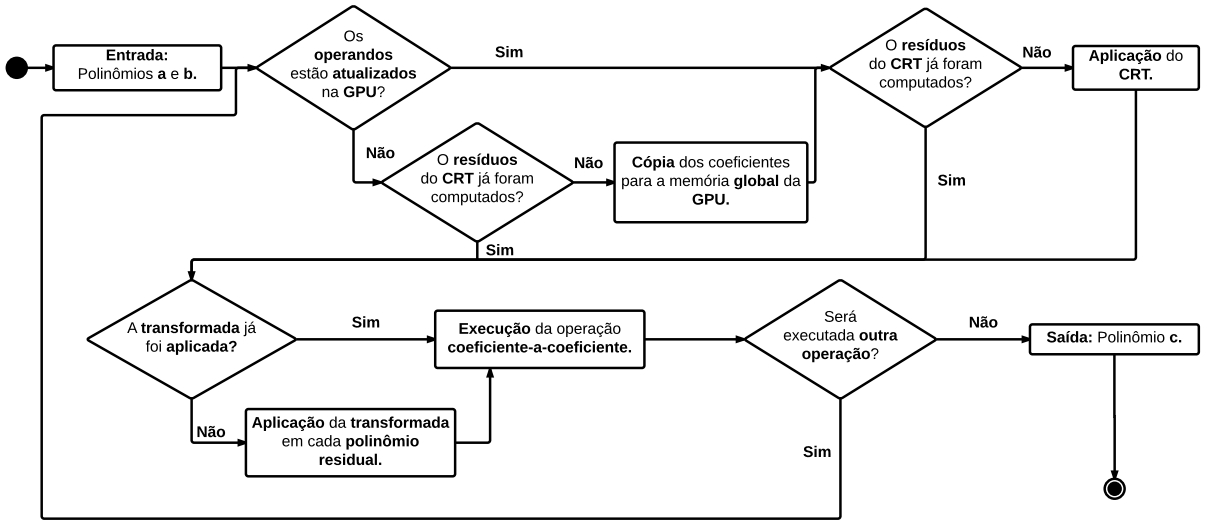


Figura 3.6: Diagrama de fluxo de operações da cuYASHE. Antes de aplicar uma operação é preciso garantir que os operandos estão atualizados na memória global da *GPU* e que os polinômios residuais foram calculados.

3.5 Fluxo de operações

O diagrama de fluxo apresentado na Figura 3.6, expõe os passos necessários para a aplicação de uma operação de adição ou multiplicação pela cuYASHE. Esses passos garantem a consistência dos dados e evitam processamento redundante.

Inicialmente verifica-se se os operandos possuem cópias atualizadas na memória global da *GPU*, se seus resíduos foram calculados e se a transformada, *FFT* ou *NTT*, foi aplicada sobre eles. Entre esses testes os mais importantes são os dois últimos. As operações da YASHE são aplicadas sobre os resíduos no domínio da transformada. Dessa forma, mesmo que os operandos não possuam seus coeficientes atualizados na memória da *GPU* mas apenas os resíduos, ainda assim pode-se realizar adições e multiplicações.

A operação de redução modular segue um fluxo diferente, uma vez que ela não pode ser aplicada sobre os resíduos. Essa operação realiza apenas o teste de localidade de memória.

Com os testes e ajustes descritos, resultados prévios da cópia de dados e da computação dos resíduos são reaproveitados. Essa otimização implica em ganhos consideráveis de desempenho em operações consecutivas, como discutido na Seção 4.5 com a Tabela 4.5.

3.6 Implementação das transformadas

Enquanto a *FFT* é uma transformada amplamente conhecida pela comunidade e utilizada por diversos trabalhos na literatura, o fato de ser baseada no corpo dos complexos traz conflitos com esquemas construídos sobre o *RLWE*. As conversões inteiro/ponto-

[12] Acrônimo de *Residue Number System*.

flutuante impedem o uso de otimizações da aritmética de corpos finitos e ainda podem trazer problemas sérios de precisão. A *NTT*, por outro lado, se mostra uma candidata mais promissora uma vez que substitui completamente a aritmética de ponto-flutuante da *FFT* pela a aritmética de corpos finitos. Detalhes teóricos das duas transformadas são discutidos na Seção 2.6, especialmente nas Seções 2.6.2 e 2.6.3.

3.6.1 cuFFT

A multiplicação polinomial por meio da *FFT* foi implementada utilizando a biblioteca cuFFT, na versão 7.5. Ela é mantida pela NVIDIA e implementa uma versão não-normalizada da proposta de Cooley-Tukey [14]. Utilizou-se o modo de funcionamento *Complex-to-Complex* da cuFFT, requerindo então o mapeamento dos operandos de $\mathbb{Z}_q^N$ em $\mathbb{C}^N$, conforme visto no Algoritmo 5.

Algoritmo 5: Algoritmo de multiplicação polinomial específico para a *FFT*. As rotinas MUL e SCALE são descritas nos Algoritmos 2 e 3.

Entrada: $A \in \mathbb{Z}_{p_i}^n$, $B \in \mathbb{Z}_{p_i}^n$.
Saída : $C = A \cdot B$.

begin
  $A_{Complex} \leftarrow convert\,(A)$;
  $B_{Complex} \leftarrow convert\,(B)$;
  $A_{fft} \leftarrow FFT(A_{Complex})$;
  $B_{fft} \leftarrow FFT(B_{Complex})$;
  $C_{fft} \leftarrow mul(A_{fft}, B_{fft})$;
  $C_{Complex} \leftarrow IFFT(C_{fft})$;
  $C \leftarrow round\_to\_nearest\,(C_{Complex})$;
  $C \leftarrow Scale(C, n)$;
end

Durante o desenvolvimento do trabalho foram observados erros de precisão causados pela cuFFT e diretamente relacionados com o tamanho do grau dos operandos e de seus coeficientes. Como ela é executada sobre os polinômios residuais, o aumento dos primos utilizados pelo *CRT* afeta a incidência desses erros.

A Figura 3.7 demonstra a falha de precisão no resultado da multiplicação de polinômios residuais de acordo com o tamanho dos operandos. A medida que o grau dos polinômios cresce, o tamanho dos coeficientes precisa diminuir para que o resultado da operação se mantenha correto. Não foram detectados erros para polinômios de grau menor ou igual a 8.192 e com coeficientes menores ou iguais a 19 *bits*.

A Figura 3.8 apresenta o desvio padrão dos erros apresentados na Figura 3.7. Para coeficientes pequenos, menores do que 21 *bits*, ele é irrisório. A partir daí ele cresce rapidamente até se estabilizar em 1. Como a aplicação do *ICRT* potencializa esses erros, mesmo pequenos eles quebram de maneira irrecuperável a correção da operação.

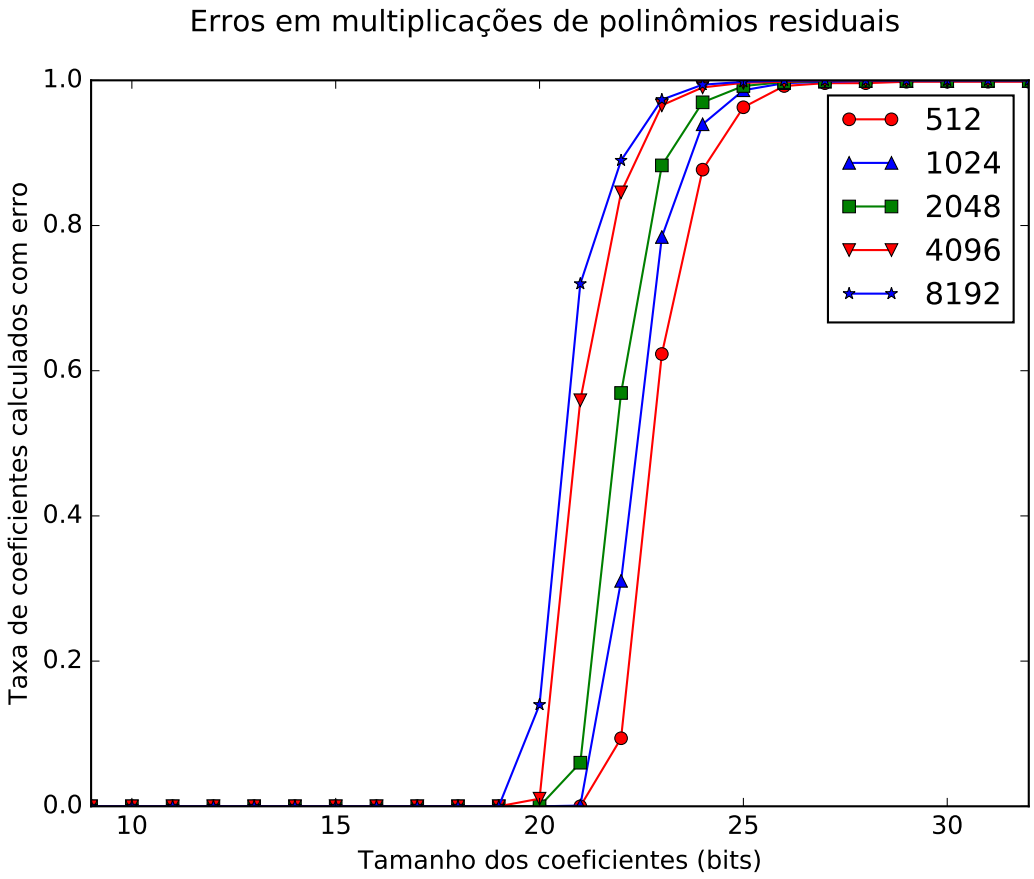


Figura 3.7: Percentual de coeficientes de polinômios residuais oriundos de multiplicação que apresentam erro de precisão. Não foram detectados problemas para operandos com grau igual ou menor a 8.192 e coeficientes menores do que 19 *bits*. A partir disso, entretanto, a taxa de coeficientes sendo calculados erroneamente cresce até atingir seu pico em 23 *bits*, quando o erro se manifesta em todos. Esses resultados foram obtidos com a CUFFT 7.5 e utilizando valores de precisão dupla.

Após ser disponibilizada, a versão 7.0 da CUFFT recebeu uma correção para um problema que gerava erros de precisão. Apesar disso, foi demonstrado que a versão 7.5 continua apresentando esse tipo de erro. A Figura 3.9 compara o percentual de erros no cálculo da multiplicação de polinômios de grau 8.192 para as duas versões da biblioteca. Como pode ser visto, a versão mais recente realmente possui uma incidência menor de erros de precisão.

3.6.2 *NTT*

Não se encontrou biblioteca semelhante a CUFFT para a *NTT*. Por isso, foi necessário que se projetasse e implementasse um algoritmo próprio para essa transformada. A formulação de Stockham para raios 2 e 4 foi utilizada como base [31].

Para aplicar a *NTT* em uma sequência de N elementos o algoritmo faz uso de $\log_R N$

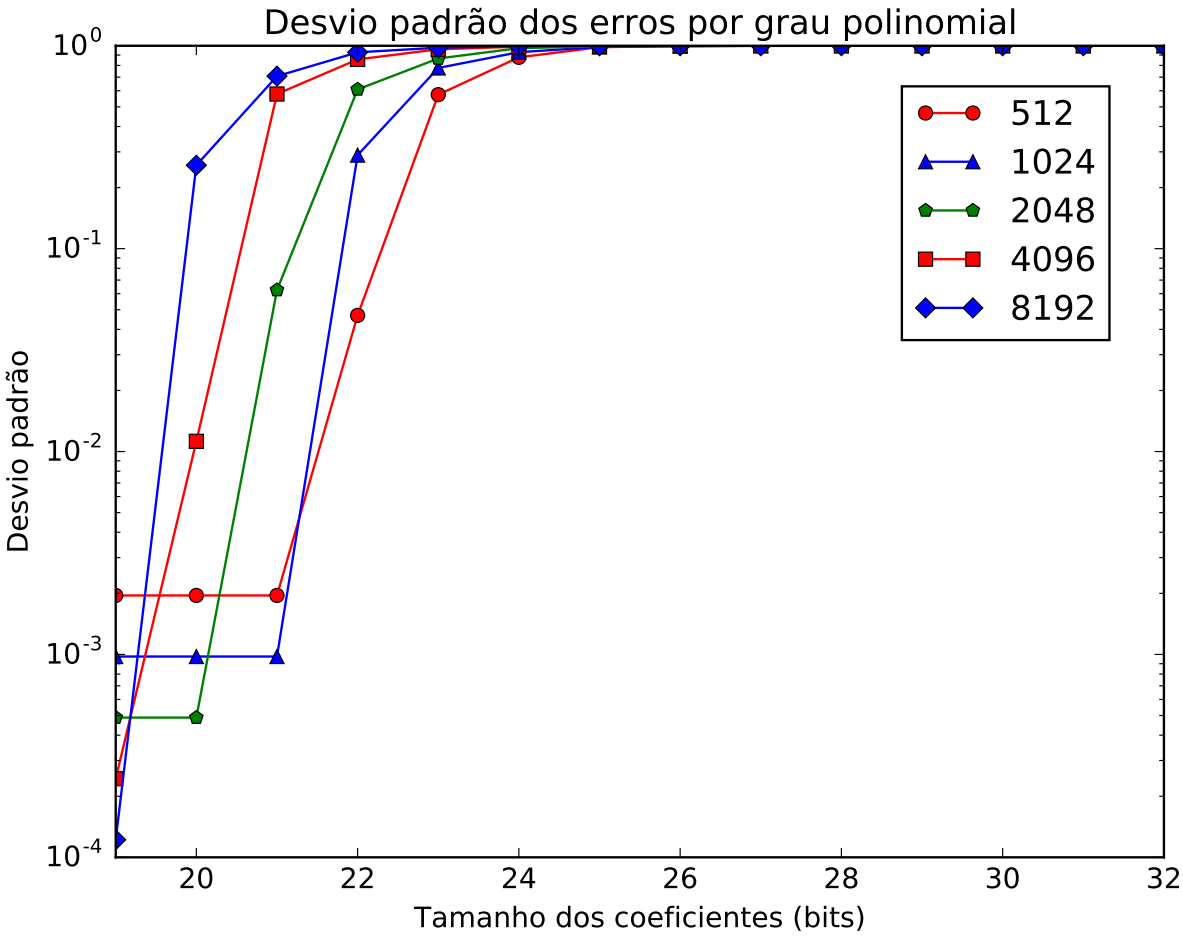


Figura 3.8: Desvio padrão dos erros apresentados na Figura 3.7. Percebe-se que até 21 bits o desvio é irrisório. Aos 26 bits ele atinge o valor de 1 e se estabiliza. O desvio padrão dos erros aparenta ter direta relação com o grau dos operandos e o tamanho dos coeficientes.

iterações. Em cada iteração, a transformada é aplicada em R subsequências de tamanho N_s, começando com $N_s = 1$. Ao final, essas subsequências são agrupadas em novas de tamanho $N_s = RN_s$ e uma nova iteração é iniciada, se necessário.

As iterações da formulação de Stockham são dependentes entre si, o que requer sua sequenciação. CUDA não provê ferramentas de sincronismo com escopo global, apenas local dentro de um bloco de *threads*. Dessa forma, é preciso que cada iteração seja feita em diferentes *kernels*, executados sequencialmente.

Para aproveitar os segmentos consecutivos acessados em cada transação de memória e reduzir o total de iterações do algoritmo, é recomendável que o raio R seja tão grande quanto possível. O limitante do tamanho desse raio é a quantidade de registradores disponíveis para cada *thread* em um *núcleo* CUDA.

O código no Anexo A.2 apresenta o algoritmo de Stockham para a NTT com os raios 2 e 4. As funções CPU_NTT e GPU_NTT são, como o nome sugere, referentes ao funcio-

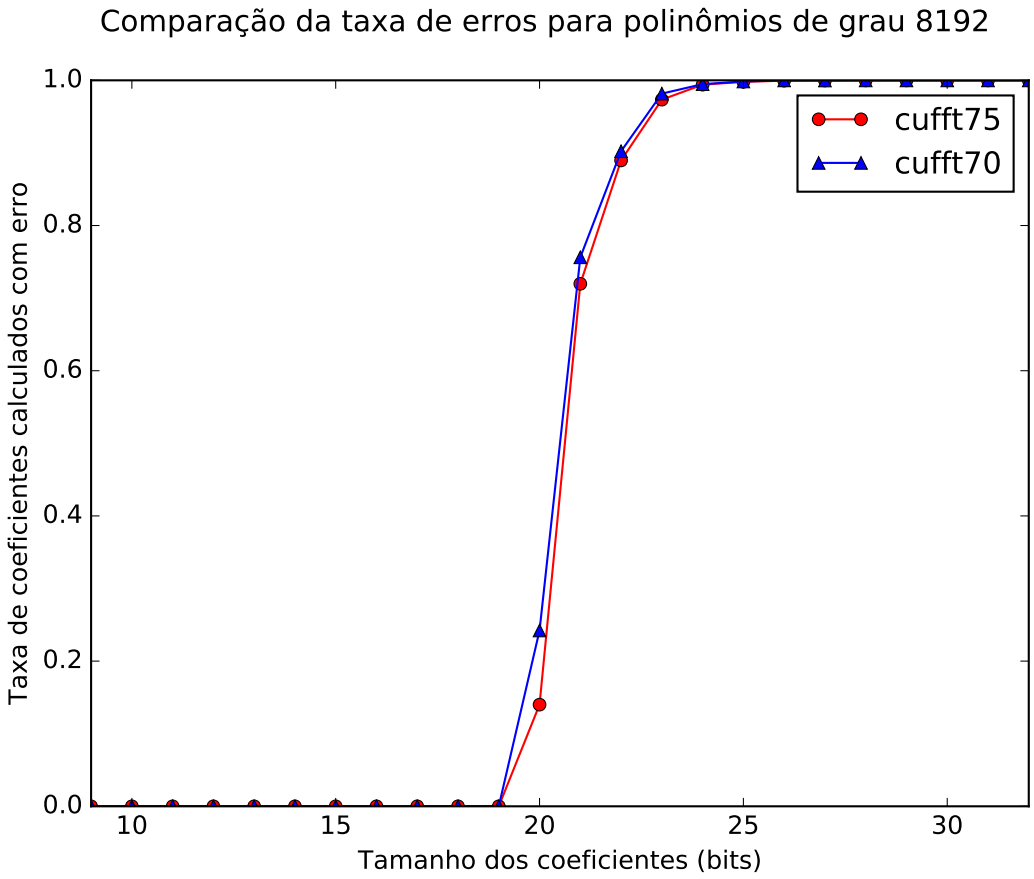


Figura 3.9: Comparação do percentual de erros no cálculo da multiplicação de polinômios de grau 8.192 entre as versões 7.0 e 7.5 da CUFFT. Percebe-se a redução na incidência de erros na versão mais recente da biblioteca. Esses resultados foram obtidos com a CUFFT utilizando valores de precisão dupla.

namento do algoritmo em modo sequencial e paralelo pela *CPU* e *GPU*, respectivamente.

A principal diferença entre essas funções é que na *GPU* o índice j é calculado a partir de uma função do índice global do *thread*. Além disso, as iterações do algoritmo são executadas por múltiplas chamadas da função NTTITERATION, que nesse caso é um *kernel*.

O cálculo de uma única *NTT* é relativamente barato e não consegue fazer uso adequado da capacidade de paralelismo da *GPU*. Dessa forma, para melhor aproveitamento dos recursos, pode-se calcular várias *NTT* em uma mesma execução. Essa característica se mostra adequada neste trabalho já que, por conta do *CRT*, a multiplicação polinomial requer a multiplicação de vários polinômios residuais.

Assim como aconteceu com a CUFFT, percebeu-se a existência de um limitante superior para o tamanho dos operandos. Quando os coeficientes ultrapassam esse valor, a aritmética modular causa a perda de informação e a quebra da função inversa. Não foi possível deduzir uma fórmula para o tamanho máximo dos coeficientes dado o anel de

polinômios $R_q = \mathbb{Z}_q[X]/\phi_n(X)$. Assim, esses limites foram levantados empiricamente por meio de uma análise de pior caso para uma multiplicação polinomial por meio da NTT. A Figura 3.10 apresenta esses valores de acordo com o grau dos operandos.

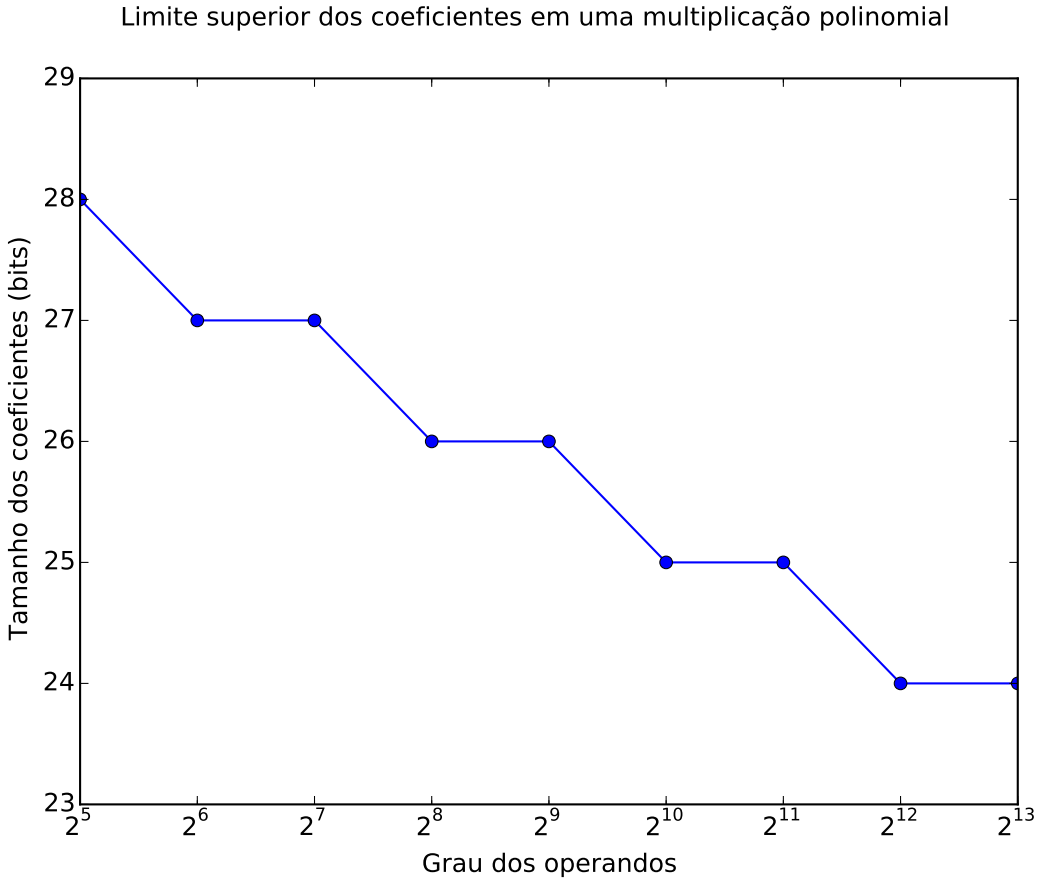


Figura 3.10: Limite superior dos coeficientes dos operandos para o funcionamento correto de uma multiplicação polinomial por meio da NTT.

3.7 Tabelas de busca

Como forma de reduzir o processamento redundante, a cuYASHE faz uso de tabelas de busca [13]. Elas armazenam valores que são requisitados diversas vezes durante a execução do programa. Essas tabelas são utilizadas pelas funções do CRT e da NTT.

CRT Os primos utilizados pelo CRT são calculados durante a fase de configuração e armazenados na memória constante da GPU. Por simplicidade e por conta das permissões de acesso às memórias seu cálculo é feito pela CPU e copiado posteriormente. O produto de todos os primos M assim como os resultados de M/p_i e $(M/p_i)^{-1} \mod p_i$, para cada

[13] Do inglês, *lookup tables*.

primo p_i, também são pré-calculados e armazenados na mesma memória. Por conta de seu tamanho, M/p_i e M são armazenados como inteiros grandes, enquanto $(M/p_i)^{-1} \mod p_i$ utiliza uma palavra simples de 64 *bits*.

NTT As primeiras N-potências da raiz primitiva e N-ésima da unidade são valores fundamentais para a aplicação da NTT, tanto em seu modo direto quanto em seu modo inverso [14]. Por isso, somado ao alto custo do seu cálculo, a cuYASHE realiza o pré-calculo desses valores e os armazena em duas tabelas, uma com as potências e outra com suas inversas. Essas tabelas são muito grandes para serem armazenadas na memória constante, o que torna necessário o uso da memória global.

3.8 Resumo do capítulo

Este trabalho produziu uma implementação do YASHE, a biblioteca cuYASHE. Ela foi escrita em C++, utiliza CUDA para acelerar a aritmética polinomial, *CRT* para simplificar a manipulação dos operandos pela *GPU* e compara as transformadas *FFT* ou *NTT* para diminuir a complexidade de tempo de uma multiplicação polinomial.

As operações sobre inteiros grandes são realizadas na *CPU* por meio da biblioteca *NTL*, enquanto que na *GPU* são realizadas com código portado da biblioteca RELIC. A aritmética de inteiros grandes ficou limitada às funções do *CRT*, assim as operações polinomiais foram implementadas com instruções nativas da arquitetura.

Como ferramenta para acelerar a multiplicação polinomial foram testadas duas transformadas, *FFT* e *NTT*. A primeira foi fornecida pela biblioteca cuFFT, enquanto a segunda foi implementada com código próprio baseado na formulação de Stockham. Dessa forma, foi possível a análise cuidadosa das vantagens e desvantagens no uso de cada transformada.

Por fim, são apresentadas as tabelas de busca utilizadas pela biblioteca para evitar calculo redundante. Elas são empregadas na implementação do *CRT* e da *NTT*.

[14] No modo indireto da *NTT*, ou *INTT*, utiliza-se a inversa multiplicativa modular.

Capítulo 4

Resultados

A biblioteca cuYASHE faz uso da plataforma CUDA para acelerar as operações homomórficas do criptossistema YASHE. Essas operações são construídas por meio de aritmética polinomial, o que implica que otimizações na implementação desta refletem diretamente no desempenho das operações homomórficas.

Este Capítulo apresenta medições de velocidade para as operações homomórficas e polinomiais implementadas na cuYASHE. Dessa forma, pode-se avaliar o ganho obtido com as estratégias descritas anteriormente.

4.1 Trabalhos relacionados

Com o intuito de avaliar a qualidade da implementação e das estratégias utilizadas na cuYASHE, foram tomados quatro trabalhos no estado da arte: Bos *et al.* [9], BLLN; a implementação de Lepoint-Naehrig [40], *LN*; a biblioteca de Dowlin *et al.*, SEAL [23]; e o trabalho de Pöppelmann *et al.* [52], *PNPM*. Os três primeiros são baseadas em *CPUs*, enquanto o último apresenta uma implementação em *FPGAs*.

Bost *et al.* - BLLN[9] O trabalho que propõe o criptossistema YASHE apresenta sua primeira implementação prova-de-conceito. Os autores afirmam haver pouca variedade de implementações de criptossistemas *FHE* e por esse motivo comparam sua implementação unicamente com o trabalho de Lauter *et al.* [39]. Essa é a única implementação baseada em *CPUs* e comparada com a cuYASHE que não teve sua implementação disponibilizada à comunidade. Dessa forma, foram utilizados os tempos fornecidos pelos autores.

Lepoint-Naehrigh - *LN*[40] A implementação de Lepoint-Naehrig foi escrita com o intuito de comparar os esquemas YASHE e FV. Ela foi escrita em C++ e utilizou a aritmética fornecida pela biblioteca FLINT [35]. Sobre essa implementação os autores aplicaram o esquema simétrico SIMON, proposto pela *NSA*[1] [7].

[1] Acrônimo de *National Security Agency*. Refere-se a agência de segurança dos EUA.

Nathan *et al.* - SEAL[23] A biblioteca SEAL[2] foi disponibilizada pela *Microsoft Research* no final de 2015. Entre os autores assinam dois dos quatro que apresentaram a YASHE no trabalho BLLN. Dessa forma, apesar de ser recente e ainda estar sob intenso desenvolvimento, é interessante compara-la com os demais trabalhos e observar a influência do compartilhamento de autores. Não são oferecidos dados oficiais de desempenho para a biblioteca, de forma que as medidas de seus tempos de execução foram feitas por este trabalho.

Pöppelmann *et al.* - *PNPM*[52] Esse trabalho se caracteriza pela investigação de como adaptar as estruturas computacionais requeridas por criptossistemas baseados no *RLWE*, em especial o YASHE, para *FPGAs*. Ele faz uso de uma formulação alternativa da *NTT*, a *cached-NTT*, voltada para simplificar o acesso à memória em sistemas com uma *cache* pequena. Ao contrário deste trabalho a implementação de *PNPM* não considerou o uso do *CRT*, apenas sugerindo sua investigação como um trabalho futuro. Os autores negaram a solução de Roy *et al.* [55] quanto a utilizar uma memória externa para facilitar o armazenamento dos operandos, argumentando que os custos de movimentação de dados seriam proibitivos. Dessa forma, sua contribuição se voltou para a otimização dos acessos à memória. A *FPGA* utilizada é a Altera Stratix V 5GSND5H.

Um trabalho complementar ao *PNPM* é o de Roy *et al.* [55]. Assim como este trabalho, os autores utilizam tanto a *NTT* quanto a *CRT*, mas em uma implementação em *FPGA*. Entre suas contribuições, consta a implementação eficiente de processadores independentes no cálculo da decomposição do *CRT*, além de uma implementação do SIMON como aplicação para a computação em dados cifrados. Contudo, como não oferecem tempos de execução para um conjunto de parâmetros próximo àqueles utilizados pelos outros trabalhos, sua implementação não é comparada com a cuYASHE.

Como comparação adicional, desejou-se avaliar operações intermediárias implementadas na cuYASHE. Para isso, tomou-se como referência a cuHE, apresentada no trabalho de Dai-Sunar [17]. Ela faz uso de paralelismo com a plataforma CUDA e propõe uma série de otimizações na implementação da aritmética polinomial de esquemas baseados em reticulados. Apesar de utilizarem criptossistemas diferentes, a cuHE e a cuYASHE compartilham a estratégia de empregar a *NTT* e a *CRT*. Dessa forma, decidiu-se desconsiderar os tempos de execução para operações do criptossistema e apenas manter a comparação entre essas funções.

4.2 Ambiente de testes e parâmetros

Este trabalho utilizou os parâmetros de configuração do YASHE propostos por BLLN [9] e *LN* [40] como padrão para as comparações de tempo de execução:

[2]Acrônimo de *Simple Encrypted Arithmetic Library*.

- $R = \mathbb{Z}[X] / (x^{4096} + 1)$,
- $q = 2^{127} - 1$,
- $w = 2^{32}$,
- $t = 2^{10}$.

Estes parâmetros definem um nível de segurança de 80 *bits*, de acordo com Lepoint-Naehrig [40].

PNPM oferece seus tempos de execução substituindo q pelo primo de Solinas $2^{124} - 2^{64} + 1$. Os autores argumentam que essa alteração não tem implicações na segurança do criptossistema. Dessa forma, a comparação com a cuYASHE foi realizada com o conjunto padrão de parâmetros.

As máquinas utilizadas em cada teste tem sua descrição completa apresentada no Anexo A.4. Para tornar as comparações justas, realizou-se as medições de tempo na Máquina 1 quando possível em uma comparação complementar. Ela dispunha de uma *CPU* Intel Xeon E5-2630 @ 2,60GHz e uma *GPU* GeForce GTX Titan Black. Além disso, a biblioteca *NTL* foi compilada com suporte à *GMP*.

As medidas dos tempos de execução de cada operação da cuYASHE foram obtidas por meio do cálculo do tempo médio de 100 execuções isoladas. Esse mesmo método foi aplicado para a avaliação feita por este trabalho das implementações *LN* e SEAL.

As metodologias para medição dos tempos de execução das implementações dos trabalhos BLLN, *LN*, Dai-Sunar e *PNPM* são fornecidas pelos autores em suas respectivas publicações.

4.3 Implementação eficiente do *CRT*

Como observado na Seção 3.3, a redução módulo q não é aplicada no domínio do *CRT* ou da transformada. Os operandos precisam ser reconstruídos para que se aplique um algoritmo usual.

Duas estratégias foram testadas por este trabalho: aplicação do $CRT/ICRT$ e redução modular na *GPU*; e a cópia dos dados de volta para a memória principal, com $CRT/ICRT$ e a redução modular executados na *CPU*.

A implementação das funções do *CRT* na *CPU* foi paralelizada com a biblioteca OpenMP, que apresentou ganho sobre a execução sequencial. Já na *GPU*, as funções do *CRT* fizeram uso da aritmética de inteiros grandes fornecida por código portado da RELIC [4]. A comparação das duas implementações pode ser vista na Tabela 4.1.

A cuYASHE tomou como padrão a execução das funções do *CRT* na *GPU*. Como pode ser visto na Tabela, essa estratégia se mostrou consideravelmente mais rápida do que a alternativa na *CPU*. Além disso, também se evita o tempo gasto com cópias de dados entre as memórias.

Tabela 4.1: Comparação entre os tempos de execução dos algoritmos direto e indireto do *CRT* em uma *CPU* e em uma *GPU*. A implementação na *CPU* faz uso de paralelismo com a biblioteca OpenMP. As medidas foram tomadas na Máquina 1, descrita no Anexo A.4.

Função/Grau do operando	***CPU* (ms)**	***GPU* (ms)**	***CPU*/*GPU***
***CRT*/ 1024**	0,55	0,03	18x
***ICRT*/ 1024**	7,74	0,43	18x
***CRT*/ 2048**	0,89	0,03	30x
***ICRT*/ 2048**	13,30	0,38	35x
***CRT*/ 4096**	1,80	0,05	36x
***ICRT*/ 4096**	26,86	0,54	50x

4.4 Multiplicação polinomial

Duas estratégias foram testadas para a implementação da multiplicação polinomial: *FFT*, implementada pela cuFFT; e *NTT*, implementada com código próprio. A implementação da *NTT* foi avaliada para os raios 2 e 4. Como esperado, a execução com raio 4 se mostrou mais rápida, como visto na Tabela 4.2. Dessa forma, definiu-se esse raio como o padrão para a transformada.

A Tabela 4.3 apresenta uma comparação do tempo necessário para aplicar uma multiplicação polinomial por meio da implementação da *NTT* e da cuFFT. Como pode ser visto, a cuFFT se mostrou de 4 até 7 vezes mais rápida.

Tabela 4.2: Comparação do tempo necessário para a aplicação da *NTT* com raios 2 e 4. Parâmetros: $\mathbf{R} = \mathbb{Z}[\mathbf{X}] / (x^{4096} + 1)$ e $q = 2^{127} - 1$.

Grau	**Raio 2 (ms)**	**Raio 4 (ms)**
1024	0,03	0,02
2048	0,06	0,02
4096	0,13	0,05
8192	0,29	0,11

Tabela 4.3: Comparação dos tempos necessários para a multiplicação de dois operandos de certo grau utilizando a cuFFT e a implementação da *NTT* baseada na formulação de Stockham de raio 4. Ambos os testes foram executados na Máquina 1, com o conjunto padrão de parâmetros definido na Seção 4.2. O *CRT* foi executado com primos de 19 *bits* para a cuFFT e 24 para a *NTT*.

Grau	**cuFFT (ms)**	***NTT* (ms)**
1024	0,3	2,16
2048	0,53	2,06
4096	0,9	4,61
8192	1,71	9,05

A implementação da *NTT* foi submetida à ferramenta *NVIDIA Visual Profiler* [46] [3] para otimização do código. Mesmo com a resolução dos problemas de divergência condicional apontados pela ferramenta, não se conseguiu otimizar os acessos à memória global. Além disso, a necessidade de se sequenciar parte do algoritmo também pode ter contribuído para a lentidão percebida. Assim, apesar da desvantagem aritmética e dos problemas de precisão, a cuFFT apresentou uma implementação mais rápida em *GPUs*, o que não é surpresa para uma biblioteca mantida pela própria NVIDIA.

Por serem transformadas tão parecidas, causa estranheza tamanha disparidade no tempo de execução. Dessa forma, foi levantada a hipótese de que a conversão da formulação de Stockham tenha trazido penalidades ao desempenho do algoritmo. O experimento apresentado na Tabela 4.4 foi concebido com isso em foco. Nele, a versão original do algoritmo teve seu tempo de execução avaliado junto de sua versão convertida para *NTT* sem qualquer interferência de processamento paralelo.

Tabela 4.4: Comparação dos tempos de execução de uma aplicação da transformada *FFT* e *NTT* implementadas com a formulação de Stockham com raio 2, a partir dos códigos apresentados nos Anexos A.1 e A.2, respectivamente. Os tempos foram medidos na Máquina 1 e as execuções foram realizadas de forma serial pela *CPU*.

Grau	***FFT* (ms)**	***NTT* (ms)**
1024	0,20	0,21
2048	0,44	0,47
4096	0,98	0,98
8192	1,82	1,91

Pode-se perceber que a conversão do algoritmo não trouxe danos ao desempenho, uma vez que os tempos de execução das duas versões é praticamente o mesmo. Dessa forma, conclui-se que a diferença expressiva em comparação com a cuFFT vem da implementação da formulação de Stockham, que não é tão eficiente quanto o algoritmo utilizado por aquela biblioteca em *GPGPUs*.

A Tabela 4.9 apresenta a comparação dos tempos de execução do algoritmo *NTT* de raio 4 para a cuYASHE e a cuHE. A *NTT* implementada pela cuHE é uma versão iterativa do algoritmo de Cooley-Tukey [14], adaptada para aritmética de corpos finitos. Em relação a cuHE, a cuYASHE apresenta ganho de 7 vezes em um anel de grau 8.192, 3,5 vezes em um anel de grau 16.384 e 4 vezes em um anel de grau 32.768.

O desempenho da cuYASHE em comparação com a cuHE sugere que, apesar de não ter sido capaz de trazer ganho de velocidade em comparação com a cuFFT, a *NTT* implementada por este trabalho tem desempenho compatível ou até mesmo superior ao estado da arte.

A maior velocidade da cuFFT implicou em seu uso como estratégia padrão para multiplicação polinomial na cuYASHE.

[3] Ferramenta proprietária da NVIDIA voltada para a análise do código e de sua execução.

4.5 Localidade de memória

A Tabela 4.5 demonstra a contribuição que a máquina de estados, apresentada nas Figuras 3.5 e 3.6, trouxe à cuYASHE. A aplicação de operações de adição ou multiplicação polinomial na *GPU* requer a cópia dos coeficientes para a memória global, seguida do cálculo dos polinômios residuais e posteriormente a aplicação das operações inversas do *CRT* e da transformada. Contudo, operações consecutivas conseguem reaproveitar valores já calculados, o que evita processamento redundante. As otimizações realizadas nesse sentido implicaram em ganhos de 962 vezes em velocidade na reutilização de operandos em operações de adição e de 250 vezes para operações de multiplicação.

Tabela 4.5: Tempos para a aritmética polinomial em um anel de grau 4096. As colunas registram os tempos médios para cada operação, considerando o custo de configuração, composto pela de copia dados para a memória da *GPU* e do cálculo dos resíduos. Os parâmetros de execução seguem aqueles apresentados na Seção 4.2.

Operação	**Com configuração (ms)**	**Sem configuração (ms)**
Adição	19,24	0,02
Multiplicação	32,28	0,13

Durante a realização deste trabalho ficou claro para os autores que a aplicação ótima da plataforma CUDA depende da minimização do movimento de dados entre memórias. Dessa forma, o desenvolvimento da cuYASHE foi pautado pela manutenção dos operandos na memória global da *GPU*.

4.6 Operações do YASHE

As operações da YASHE são dependentes apenas de adições e multiplicações polinomiais. Uma vez que adições são operações bastante baratas do ponto de vista de complexidade, o espaço para otimização se concentra na operação de multiplicação, como pode ser visto na Tabela 4.6.

Considerando a adição homomórfica, a cuYASHE foi 35 vezes mais rápida do que o trabalho *LN*, ao mesmo tempo que manteve o tempo da implementação BLLN. Acredita-se que o ganho sobre a *LN* se deve por falta de esmero dos autores em sua implementação, uma vez que é completamente sequencial. Apesar disso, os tempos absolutos de execução são pequenos em comparação com as outras operações do YASHE. Logo, o impacto no desempenho pode não ter sido julgado significativo por Lepoint-Naehrig.

As operações de cifração, decifração e multiplicação homomórfica tem forte dependência da multiplicação polinomial. Dessa forma, as otimizações obtidas impactaram diretamente o desempenho dessas operações. As três obtiveram ganho de velocidade da ordem de 8 até 9 vezes em relação a *LN*. Ao mesmo tempo, enquanto a cifração e a multiplicação homomórfica atingiram ganho de 15 e 6 vezes, respectivamente, a decifração foi 2,5 vezes mais rápida que o BLLN.

Tabela 4.6: Tempos para o cuYASHE e comparação com os resultados fornecidos pelos trabalhos de *LN* [40] e BLLN [9], respectivamente. A máquina utilizada na obtenção dos tempos da cuYASHE é descrita na Máquina 1, enquanto que os trabalhos *LN* e BLLN são avaliados nas Máquinas 2 e 3, respectivamente. Os parâmetros de execução seguem aqueles apresentados na Seção 4.2. Os tempos para multiplicação homomórfica incluem o custo de relinearização.

Operação	**cuYASHE(ms)**	***LN*(ms)**	**BLLN(ms)**
Cifração	1,85	16	27
Decifração	2,01	15	5
Adição Homomórfica	0,02	0,7	0,02
Multiplicação Homomórfica	5,49	49	31

A Tabela 4.7 repete essa comparação mas com execuções realizadas na mesma máquina. Ao contrário dos tempos citados anteriormente, que foram fornecidos pelos autores em seus respectivos trabalhos, dessa vez mediu-se os tempos das operações a partir do código fonte, executado na Máquina 1.

O BLLN não pôde ser utilizado nessa comparação uma vez que seu código não está disponível à comunidade. A biblioteca SEAL, por outro lado, não participou das comparações da Tabela 4.6 por não terem sido encontradas medições de seus tempos de execução fornecidas pelos autores. Contudo, como seu código é público, ela substituiu a BLLN nas comparações de tempo de execução na mesma máquina, apresentadas pela Tabela 4.7.

Em relação a *LN*, pode-se perceber uma sutil redução nos ganhos de tempo. Isso já era esperado por conta da Máquina 1 ter uma configuração de *hardware* superior a Máquina 2, utilizada no trabalho original. A cuYASHE apresentou a operação de cifração 8 vezes mais rápida que a *LN*, enquanto que a decifração e a multiplicação homomórfica atingiram 7 e 6 vezes. Sobre a SEAL esses ganhos foram ainda maiores, atingindo 19, 17 e 35 vezes, respectivamente.

Tabela 4.7: Comparação entre a cuYASHE e as implementações *LN* e SEAL, avaliadas na Máquina 1. Os parâmetros de execução seguem aqueles apresentados na Seção 4.2. Os tempos para multiplicação homomórfica incluem o custo de relinearização.

Operação	**cuYASHE(ms)**	***LN*(ms)**	**SEAL(ms)**
Cifração	1,85	15,4	34,93
Decifração	2,01	13,71	34,1
Adição Homomórfica	0,02	0,59	0,18
Multiplicação Homomórfica	5,49	31,07	194,94

A Tabela 4.8 compara as operações homomórficas da *PNPM* com a cuYASHE. Apesar da adição homomórfica não ser tão eficiente, a multiplicação se mostra bastante próxima, o que demonstra que a implementação do YASHE em *FPGA* pode ser feita de maneira tão eficiente quanto em *GPGPU*.

A principal motivação para a utilização de um criptossistema homomórfico é poder

Tabela 4.8: Tempos para o cuYASHE e comparação com os resultados fornecidos pelo trabalho de *PNPM* [52]. A máquina utilizada na obtenção dos tempos da cuYASHE é descrita na Máquina 1. Os parâmetros de execução seguem aqueles apresentados na Seção 4.2. Os tempos para multiplicação homomórfica incluem o custo de relinearização.

Operação	**cuYASHE(ms)**	***PNPM* (ms)**
Adição Homomórfica	0,02	0,19
Multiplicação Homomórfica	5,49	6,75

operar sobre criptogramas. Dessa forma, apesar de ganhos na cifração e decifração serem importantes, operações homomórficas são o alvo principal para otimizações. Mesmo pequenos ganhos de velocidade são significativos, uma vez que serão amplificados por diferentes tipos de algoritmo.

4.7 *CRT*

A Tabela 4.9 oferece uma comparação do *CRT* implementado pela cuHE, de Dai-Sunar [17], e pela cuYASHE. Quando comparada com a cuHE, percebe-se que a função direta da *CRT* é executada até 15 vezes mais rapidamente pela cuYASHE. Ao mesmo, a cuHE apresenta uma implementação da *ICRT* até 3 vezes mais rápida.

A função direta do *CRT* é bastante simples, dependendo apenas de uma redução modular. A função inversa, por outro lado, é uma operação mais complexa, exigindo multiplicações e adições de inteiros grandes, além da redução modular. A implementação da *ICRT* feita por Dai-Sunar se mostra mais eficiente que a deste trabalho. Ela usa intensamente operações em *assembly* e consegue explorar mais adequadamente acessos coalescidos à memória.

Tabela 4.9: Comparação dos tempos de execução para *NTT*, *CRT* e *ICRT* nas bibliotecas cuYASHE e cuHE. A cuYASHE foi medida na Máquina 1, enquanto que a cuHE apresenta valores medidos na Máquina 4 e fornecidos pelo trabalho de Dai-Sunar [17]. Os parâmetros de execução são definidos por Dai-Sunar onde os primos usados pelo *CRT* possuem 24 *bits* e são usados respectivamente 15, 25 e 40 polinômios residuais para anéis de grau 8.192, 16.384 e 32.768.

Biblioteca	**Grau**	***NTT*-4 (ms)**	***CRT* (ms)**	***ICRT* (ms)**
cuYASHE	8.192	0,12	0,14	1,58
cuHE	8.192	0,84	0,70	0,54
cuYASHE	16.384	0,51	0,44	8,78
cuHE	16.384	1,78	4,00	3,73
cuYASHE	32.768	1,57	1,42	39,63
cuHE	32.768	6,24	21,31	17,94

4.8 Resumo do capítulo

Este Capítulo apresentou tempos de execução para diferentes rotinas da cuYASHE. Esses tempos foram comparados com outras implementações conhecidas da literatura de forma a ser possível avaliar sua qualidade.

A Seção 4.3 apresenta uma comparação da implementação das funções do *CRT* em *CPU* utilizando OpenMP para o paralelismo e em *GPU* sobre a plataforma CUDA. Ela demonstra ganhos de 18 até 50 vezes com o uso do paralelismo em *GPUs* para esses operações.

Este trabalho analisou a multiplicação polinomial sendo executada por meio da *FFT* e da *NTT*. A Seção 4.4 compara a implementação dessa operação para cada uma das transformadas, concluindo que a *NTT* não conseguiu atingir o mesmo desempenho da *FFT* implementada pela cuFFT.

A Seção 4.5 demonstra os ganhos obtidos pela manutenção dos dados na memória global da *GPU*, enquanto a Seção 4.7 compara as funções da *NTT* e *CRT* com o trabalho de Dai-Sunar [17]. Ela demonstra que apesar da implementação da transformada na cuYASHE ter tempos de execução semelhantes, as operações sobre inteiros grandes são mais rápidas naquele trabalho.

Por fim, a Seção 4.6 compara os tempos de execução das operações do YASHE deste trabalho com quatro outros que também implementam o criptossistema: BLLN [9], *LN* [40], SEAL [23] e *PNPM* [52]. Os três primeiros forneceram tempos de execução para *CPU*, enquanto que o *PNPM* utiliza *FPGA*. Pode-se perceber ganhos de velocidade para todas as operações do criptossistema, inclusive as homomórficas, que são o foco deste trabalho.

Capítulo 5

Conclusão

Este trabalho investigou estratégias para a implementação do criptossistema YASHE em *GPGPUs*. Com os resultados obtidos a biblioteca cuYASHE foi desenvolvida, otimizada e disponibilizada à comunidade [3].

O *CRT* foi utilizado para reduzir a complexidade da aritmética de inteiros grandes. Por meio dele, conseguiu-se executar a maior parte das operações utilizando instruções nativas da arquitetura. Além disso, analisou-se o desempenho de suas implementações *CPU* e em *GPU*. Assim, ficou evidente que as funções do *CRT* são adequadas para paralelização, obtendo ganhos de até 50 vezes com sua execução paralela na *GPU*.

As transformadas *FFT* e *NTT* foram utilizadas para reduzir a complexidade de uma multiplicação polinomial. A implementação da *FFT* foi fornecida pela biblioteca cuFFT, enquanto a *NTT* foi implementada com código próprio, baseado na formulação de Stockham. Este trabalho não conseguiu gerar uma implementação da *NTT* que se equiparasse em velocidade com a cuFFT. Isso é atribuído a necessidade de se sequenciar parte da execução da *NTT* e a um padrão ruim de acesso à memória global.

Durante a implementação da multiplicação polinomial percebeu-se um erro inesperado de precisão gerado pela cuFFT. Os autores realizaram uma análise e relacionaram a intensidade desse erro com o grau dos operandos e tamanho dos coeficientes. Foram apresentados a frequência e o desvio padrão desses erros, além de uma comparação de sua ocorrência em uma versão anterior da biblioteca. Assim, percebeu-se que para polinômios de grau igual ou menor a 8.192 os coeficientes devem ter tamanho limitado a 19 *bits* para se evitar esses erros de precisão.

Foram desenvolvidas formulações otimizadas para a redução por polinômios ciclotômicos ou por primos de Mersenne. Essas formulações tiram proveito da estrutura desses elementos e possibilitaram a simplificação dessas operações em adições, subtrações e deslocamentos.

Como método de avaliação da eficiência da implementação, comparou-se os tempos de execução da cuYASHE com cinco outros trabalhos da literatura, três relativos a implementações em *CPU*, um em CUDA e um em *FPGA*.

Os três trabalhos que implementaram o YASHE em *CPUs*, o BLLN, *LN* e SEAL,

apresentaram tempos de execução consideravelmente maiores do que este trabalho. A operação de multiplicação homomórfica, por exemplo, apresentou redução do tempo de execução de 6 até 35 vezes.

As vantagens oriundas da aplicação de *GPGPUs* na aceleração das operações do YASHE ficaram evidentes com a execução deste trabalho. Acredita-se que esses ganhos possam ser estendidos para outros criptossistemas baseados no *RLWE*, como Dai-Sunar [17] já haviam demonstrado para o LTV.

Futuras investigações poderiam explorar o espaço para ganhos de velocidade na multiplicação polinomial. Como demonstrado, a implementação da *NTT* gerada para a cuYASHE não se mostrou tão eficiente quanto a cuFFT. Com a devida otimização, essa transformada pode atingir o mesmo desempenho da cuFFT, com a possibilidade de haver ganho por conta do aumento do tamanho dos primos utilizados pelo *CRT*.

Outra área a ser explorada é a aplicação do *RNS* em substituição ao *CRT*. Ele permitiria a aplicação da redução modular no domínio dos resíduos, o que reduziria o uso de operações sobre inteiros grandes.

Além disso, espera-se a implementação utilizando a cuYASHE de um protocolo que preserve privacidade. Dessa forma, as contribuições apresentadas por este trabalho seriam ilustradas com a aplicação em um cenário realista.

Por fim, a programação orientada a objetos se mostrou incompatível com o objetivo de minimizar os tempos de execução. Dessa forma, sugere-se que futuras implementações que visem ganhos de velocidade não utilizem esse paradigma.

Referências Bibliográficas

[1] Pedro Alves and Diego Aranha. cuYASHE: Computação sobre dados cifrados em GPGPUs. In *XV Simpósio Brasileiro de Segurança da Informação e Sistemas Computacionais (SBSeg 2015)*, pages 55–60, 2015.

[2] Pedro Alves and Diego Aranha. cuYASHE: Computação sobre dados cifrados em GPGPUs. In *X Workshop de Teses, Dissertações e Trabalhos de Iniciação Científica em Andamento do IC-Unicamp*, pages 198–210, 2015.

[3] Pedro Alves and Diego Aranha. cuYASHE. https://github.com/cuyashe-library/cuyashe, 2016. Último acesso: 09/06/2016.

[4] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.

[5] Jean-Claude Jc Bajard, Nicolas Meloni, and Thomas Plantard. Efficient RNS bases for Cryptography. *IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation*, 2005.

[6] Paul Barrett. *Advances in Cryptology — CRYPTO' 86: Proceedings*, chapter Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor, pages 311–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[7] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404, 2013.

[8] Bloomberg News. China Said to Plan Sweeping Shift From Foreign Technology to Own. Bloomberg, 2014.

[9] JoppeW. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme. In Martijn Stam, editor, *Cryptography and Coding*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer Berlin Heidelberg, 2013.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

[11] Rajkumar Buyya. Market-Oriented Cloud Computing: Vision, Hype, and Reality of Delivering Computing As the 5th Utility. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 1–, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Amazon Q4 2015 Report, January 2016.

[13] William T. Cochran, James W. Cooley, David L. Favin, Howard D. Helms, Reginald A. Kaenel, William W. Lang, Jr. George C. Maling, David E. Nelson, Charles M. Rader, and Peter D. Welch. What is the fast Fourier transform? *IEEE Transactions on Audio and Electroacoustics*, 15:45–55, 1967.

[14] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[15] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. *Advances in Cryptology — EUROCRYPT '97: International Conference on the Theory and Application of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings*, chapter A Secure and Optimally Efficient Multi-Authority Election Scheme, pages 103–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[16] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael, 1999.

[17] Wei Dai and Berk Sunar. cuHE: A Homomorphic Encryption Accelerator Library. Cryptology ePrint Archive, Report 2015/818, 2015.

[18] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The Cost of Doing Science on the Cloud: The Montage Example. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[19] T. Dierks and C. Allen. The TLS Protocol Version 1.0, 1999.

[20] W. Diffie and M.E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, Nov 1976.

[21] C. Ding, D. Pei, and A. Salomaa. *Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[22] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, 2013.

[23] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for Using Homomorphic Encryption for Bioinformatics, 2015.

[24] Dave Durkee. Why Cloud Computing Will Never Be Free. *Queue*, 8(4):20:20–20:29, April 2010.

[25] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In GeorgeRobert Blakley and David Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Berlin Heidelberg, 1985.

[26] Paul Erdős, Carl Pomerance, and Eric Schmutz. Carmichael's lambda function. *Acta Arithmetica*, 58(4):363–385, 1991.

[27] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[28] G1. WhatsApp bloqueado: operadoras são intimadas a barrar app no país por 48h, 2015.

[29] Craig Gentry. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM*, 53(3):97–105, March 2010.

[30] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption &Amp; How to Play Mental Poker Keeping Secret All Partial Information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 365–377, New York, NY, USA, 1982. ACM.

[31] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[32] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. Último acesso: 05/03/2016.

[33] Glenn Greenwald and Ewen MacAskill. NSA Prism program taps in to user data of Apple, Google and others, 2013.

[34] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[35] W. B. Hart. Fast Library for Number Theory: An Introduction. In *Proceedings of the Third International Congress on Mathematical Software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag. `http://flintlib.org`.

[36] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[37] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good. On the Use of Cloud Computing for Scientific Workflows. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 640–645, Dec 2008.

[38] Eric Knorr. 2016: The year we see the real cloud leaders emerge, January 2016. Último acesso 18/03/2016.

[39] Kristin Lauter, Michael Naehrig, and Vinod Vaikuntanathan. Can Homomorphic Encryption be Practical? Cryptology ePrint Archive, Report 2011/405, 2011.

[40] Tancrède Lepoint and Michael Naehrig. A Comparison of the Homomorphic Encryption Schemes FV and YASHE. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology – AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 318–335. Springer International Publishing, 2014.

[41] Shu-Shen Li, Gui-Lu Long, Feng-Shan Bai, Song-Lin Feng, and Hou-Zhi Zheng. Quantum computing. *Proceedings of the National Academy of Sciences*, 98(21):11847–11848, 2001.

[42] Richard Lindner and Chris Peikert. *Topics in Cryptology – CT-RSA 2011: The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, chapter Better Key Sizes (and Attacks) for LWE-Based Encryption, pages 319–339. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[43] Adriana Lopez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2013/094, 2013.

[44] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 – June 3, 2010. Proceedings*, chapter On Ideal Lattices and Learning with Errors over Rings, pages 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[45] NVIDIA. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/curand/, 2015. Último acesso: 23/03/2016.

[46] NVIDIA. NVIDIA Visual Profiler. `https://developer.nvidia.com/nvidia-visual-profiler`, 2016. Último acesso: 20/03/2016.

[47] NVIDIA Corporation. NVIDIA CUDA C Best practices guide, 2016. Versão 7.5, Último acesso: 20/03/2016.

[48] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[49] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg, 1999.

[50] Chris Peikert. An Efficient and Parallel Gaussian Sampler for Lattices. Cryptology ePrint Archive, Report 2010/088, 2010.

[51] Chris Peikert. A Decade of Lattice Cryptography. Cryptology ePrint Archive, Report 2015/939, 2015.

[52] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating homomorphic evaluation on reconfigurable hardware. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9293, pages 143–163, 2015.

[53] R L Rivest, L Adleman, and M L Dertouzos. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[54] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[55] Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil Dimitrov, and Ingrid Verbauwhede. Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation. In *IACR-CHES-2015*, 2015.

[56] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[57] Victor Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl`, 2003. Último acesso: 05/03/2016.

[58] Jerome A. Solinas. Generalized Mersenne Numbers. Technical report, 1999.

[59] Jerome A Solinas. *Generalized Mersenne Prime*, chapter Generalize, pages 509–510. Springer US, Boston, MA, 2011.

[60] Richard Stallman. GNU General Public License, 2007. `http://www.gnu.org/licenses/gpl.html`.

[61] Damien Stehlé and Ron Steinfeld. Making NTRUEncrypt and NTRUSign as Secure as Standard Worst-Case Problems over Ideal Lattices. Cryptology ePrint Archive, Report 2013/004, 2013.

[62] C. Vecchiola, S. Pandey, and R. Buyya. High-Performance Cloud Computing: A View of Scientific Applications. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pages 4–16, Dec 2009.

[63] Harrison Weber. How the NSA & FBI made Facebook the perfect mass surveillance tool, 2014.

[64] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[65] Zhifeng Xiao and Yang Xiao. Security and Privacy in Cloud Computing. *IEEE Communications Surveys Tutorials*, 15(2):843–859, Second 2013.

Apêndice A

Anexo

A.1 Código da FFT-Stockham

Código A.1: "Implementação base da transformada *FFT* utilizando a formulação de Stockham."

```
1  float2* CPU_FFT( int N, int R,
2                   float2* data0, float2* data1){
3          for( int Ns = 1; Ns < N; Ns*=R ){
4                   for( int j = 0; j < N/R; j++ )
5                           FftIteration( j, N, R, Ns, data0, data1 );
6                   swap( data0, data1 );
7          }
8          return data0;
9  }
10
11 void GPU_FFT( int N, int R, int Ns,
12                 float2* dataI, float2* data0){
13         int j = b*N+t;
14         FftIteration( j, N, R, Ns, dataI, data0 );
15 }
16
17 void FftIteration( int j, int N, int R, int Ns,
18                       float2* data0, float2* data1){
19         float2 v[R];
20         int idxS = j;
21         float angle = -2*PI*(j%Ns)/(Ns*R);
22         for( int r = 0; r < R; r++ ){
23                  v[r] = data0[idxS+r*N/R];
24                  v[r] *= (cos(r*angle), sin(r*angle));
25         }
26         butterfly<R>(v);
27         int idxD = expand(j,Ns,R);
28         for( int r = 0; r < R; r++)
29                  data1[idxD+r*Ns] = v[r];
```

```
30 }
31
32 void butterfly<2>(float2* v){
33         float2 v0 = v[0];
34         v[0] = v0 + v[1];
35         v[1] = v0 - v[1];
36 }
37
38 void butterfly<4>(int*v){
39         float2 v0 = v[0];
40         float2 v1 = v[1];
41         float2 v2 = v[2];
42
43         v[0] = v0 + v1 + v2 + v[3];
44         v[1] = v0 + W4*v1 - v2 - W4*v[3];
45         v[2] = v0 - v1 + v2 - v[3];
46         v[3] = v0 - W4*v1 - v2 + W4*v3;
47
48 }
49
50
51 int expand(int idxL, int N1, int N2 ){
52         return (idxL/N1)*N1*N2 + (idxL%N1);
53 }
```

A.2 Código da NTT-Stockham

Código A.2: "Implementação da transformada *NTT* utilizando a formulação de Stockham baseada na versão para a *FFT*."

```
1  int* CPU_NTT( int N, int R,
2                  int* data0, int* data1){
3          for( int Ns = 1; Ns < N; Ns*=R ){
4                  for( int j = 0; j < N/R; j++ )
5                          NttIteration( j, N, R, Ns, data0, data1 );
6                  swap( data0, data1 );
7          }
8          return data0;
9  }
10
11 void GPU_NTT( int N, int R, int Ns,
12       int* dataI, int* data0){
13         int j = b*N+t;
14         NttIteration( j, N, R, Ns, dataI, data0 );
15 }
16
17 void NttIteration( int j, int N, int R, int Ns,
18            int* data0, int* data1){
19         int v[R];
20         int idxS = j;
21         int w_index = ((j%Ns)*N)/(Ns*R)
22
23         for( int r = 0; r < R; r++ ){
24                 v[r] = data0[idxS+r*N/R];
25                 v[r] *= W[w_index*r];
26         }
27         butterfly<R>(v);
28         int idxD = expand(j,Ns,R);
29         for( int r = 0; r < R; r++)
30                 data1[idxD+r*Ns] = v[r];
31 }
32
33 void butterfly<2>(int* v){
34         int v0 = v[0];
35         v[0] = v0 + v[1];
36         v[1] = v0 - v[1];
37 }
38
39 void butterfly<4>(int*v){
40         int v0 = v[0];
41         int v1 = v[1];
42         int v2 = v[2];
43         int W4 = 281474976710656;
44
```

```
45          v[0] = v0 + v1 + v2 + v[3];
46          v[1] = v0 + W4*v1 - v2 - W4*v[3];
47          v[2] = v0 - v1 + v2 - v[3];
48          v[3] = v0 - W4*v1 - v2 + W4*v3;
49
50  }
51
52  int expand(int idxL, int N1, int N2 ){
53          return (idxL/N1)*N1*N2 + (idxL%N1);
54  }
```

A.3 Prova de correção do algoritmo de redução modular de Barrett

Este Anexo se propõe a demonstrar a correção do algoritmo de redução de Barrett, apresentado na Seção 2.7.

Demonstração. O algoritmo de redução de Barrett tem como objetivo encontrar $R = W \mod M$, para $M \geq 3$ não sendo uma potência de 2 e $W < M^2$. Este anexo demonstra que o resultado da função principal da redução de Barrett mora em $[0, 2M)$ e por isso pode ser necessário que uma subtração por M seja aplicada ao final.

Como definido na Equação 2.8, $N' = \lfloor \frac{4^k}{M} \rfloor$. Deixando de lado o arredondamento de N' para o conjunto dos inteiros discutido na Seção 2.7, como M não é uma potência de 4, então $\frac{4^k}{M}$ não é uma divisão inteira. Dessa forma,

$$\frac{4^k}{M} - 1 < N' < \frac{4^k}{M}. \tag{A.1}$$

Multiplicando a Equação A.1 por $W \geq 0$,

$$W\frac{4^k}{M} - W \leq WN' \leq W\frac{4^k}{M}. \tag{A.2}$$

Dividindo por 4^k,

$$\frac{W}{M} - \frac{W}{4^k} \leq \frac{WN'}{4^k} \leq \frac{W}{M}.$$

Por construção, $W < M^2 < 4^k$, então $\frac{W}{4^k} < 1$. Logo,

$$\frac{W}{M} - 1 < \frac{WN'}{4^k} \leq \frac{W}{M}. \tag{A.3}$$

É fácil ver que $\frac{W}{M} - 2 < \lfloor \frac{W}{M} - 1 \rfloor$. Além disso, da Equação A.3, $\lfloor \frac{W}{M} - 1 \rfloor \leq \lfloor \frac{WN'}{4^k} \rfloor$. Portanto,

$$\frac{W}{M} - 2 < \lfloor \frac{W}{M} - 1 \rfloor \leq \frac{WN'}{4^k} \leq \frac{W}{M}.$$

Multiplicando por M e negando a expressão,

$$W - 2M < \frac{WN'}{4^k}M \leq W \tag{A.4}$$

$$-W \leq -\frac{WN'}{4^k}M < 2M - W. \tag{A.5}$$

Por fim, adicionando W à Equação A.5,

$$0 \leq W - \frac{WN'}{4^k}M < 2M. \tag{A.6}$$

Como $\frac{WR}{4^k}M$ é multiplo de M, então $W \equiv R = W - \frac{WR}{4^k}M \mod M$. Se $W - \frac{WR}{4^k}M \geq$

M, define-se o resto $R = \left(W - \frac{WR}{4^k}M\right) - M$, que continua respeitando a equivalência explicitada anteriormente. Caso contrário, $R = W - \frac{WR}{4^k}M$, o que conclui o objetivo enunciado no inicio deste Anexo. □

A.4 Máquinas

Esta Seção é destinada para descrever as máquinas usadas em testes mencionados por este trabalho.

Máquina 1. *Configuração da máquina utilizada por este trabalho.*

	Versão / Modelo	***Frequência***
CPU	*Intel Xeon E5-2630*	*2,60 GHz*
GPU	*GeForce GTX TITAN Black*	*0,98GHz*
CUDA	*7.5*	-
NTL	*9.1.0*	-
GMP	*6.0.0*	-
FLINT	*2.4*	-

Máquina 2. *Configuração da máquina utilizada pelo trabalho de Lepoint-Naehrig[40].*

	Versão / Modelo	***Frequência***
CPU	*Intel Core i7-2600*	*3.4GHz*
FLINT	*2.4*	-

Máquina 3. *Configuração da máquina utilizada pelo trabalho de Bos et al.[9].*

	Versão / Modelo	***Frequência***
CPU	*Intel Core i7-3520*	*2893,484 MHz*

Máquina 4. *Configuração da máquina utilizada pelo trabalho de Dai-Sunar[17].*

	Versão / Modelo	***Frequência***
CPU	*Intel Core i7 3770k*	*3,50 GHz*
GPU	*GeForce GTX690*	*1,020GHz*
GPU	*GeForce GTX770*	*1,163GHz*
CUDA	*7.0*	-
NTL	*9.2.0*	-
GMP	*6.0.0*	-
FLINT	*2.4*	-