cuYASHE : Computação sobre dados cifrados em *GPGPUs*

Pedro Geraldo M. R. Alves[1], Diego de Freitas Aranha[1]

[1] Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Cidade Universitária Zeferino Vaz – CEP 13083-970 – Campinas – SP – Brazil

pdroalves@gmail.com, dfaranha@ic.unicamp.br

Abstract. *Under the dominant cloud computing paradigm, employing encryption for data storage and transport may not be enough. Security guarantees should also be extended to data processing, to preserve the privacy of data owners and contractors. Homomorphic encryption schemes are a natural candidate for computing over encrypted data, satisfying these new security requirements. In this work,* cuYASHE *is presented as a GPGPU implementation of the leveled fully homomorphic scheme* YASHE*. The implementation employs the CUDA platform, the Chinese Remainder Theorem and the Fast Fourier Transform to obtain significant performance improvements. When compared with the state-of-the-art implementation in CPUs, speedups of* 20% *for homomorphic addition and* 58% *for multiplication were observed. These operations are performance-critical for evaluating any function over encrypted data, demonstrating that GPUs are an appropriate technology for bootstraping privacy-preserving cloud computing environments.*

Resumo. *Em tempos de computação em nuvem, há interesse em se utilizar criptossistemas que não apenas garantam a segurança dos dados no transporte e armazenamento, mas também durante o processamento, de forma a preservar a privacidade dos contratantes e detentores dos dados. Esquemas de cifração homomórfica são candidatos promissores para computação sobre dados cifrados, satisfazendo novos requisitos de segurança. Neste trabalho, é apresentada* cuYASHE*, uma implementação em GPGPUs do criptossistema completamente homomórfico em nível,* YASHE*. A implementação emprega a plataforma CUDA, o Teorema Chinês do Resto e a Transformada de Fourier para obter ganhos significativos de desempenho. Quando comparada com a implementação estado da arte em CPUs, foram obtidos ganhos de* 20% *de velocidade na adição e* 58% *na multiplicação homomórfica, operações críticas do ponto de vista de desempenho para se avaliar qualquer função sobre dados cifrados. Isso demonstra que GPGPUs são uma tecnologia adequada para se implementar serviços de computação em nuvem que preservam a privacidade.*

1. Introdução

O recente barateamento do poder computacional pelo paradigma de computação em nuvem tornou popular o uso desse tipo de serviço e o espalhou por toda a indústria. A possibilidade de terceirizar a instalação, manutenção e a escalabilidade de servidores, somada a preços competitivos, faz com que esses serviços se tornem bastante atraentes [Buyya 2009].

Atualmente, diversos esquemas criptográficos são usados como padrão no armazenamento e troca de dados. Contudo, no caso da computação em nuvem existe a possibilidade de se lidar com um atacante classificado como honesto mas curioso [1]. Nesse contexto, as partes

[1] Do Inglês, *honest-but-curious*.

 ©2015 SBC — Soc. Bras. de Computação

envolvidas com a nuvem concordam em respeitar os protocolos pré-definidos para a manipulação e troca dos dados mas não há garantia de que, caso haja interesse na quebra de privacidade, elas não usem seu conhecimento e acesso ao sistema para subverter a segurança. Dessa forma, existe a necessidade que o processamento dos dados seja feito com estes em estado cifrado, garantindo uma trajetória completamente sigilosa para os dados: transporte, processamento e armazenamento.

A criptografia homomórfica se mostra ideal para ser usada nesse cenário. Nesta classe de criptossistemas existe o suporte a operações que podem ser aplicadas sobre dados em estado cifrado, sem conhecimento das chaves criptográficas. O resultado é nativamente cifrado com a chave original, sem revelá-la em nenhum momento.

O objetivo deste trabalho consistiu em obter ganho de desempenho no estado da arte do criptossistema YASHE [Bos et al. 2013], uma variante da família de criptossistemas baseados em reticulados [Hoffstein et al. 1998]. Para isso, foram aplicadas técnicas de computação paralela em processadores gráficos de propósito geral, ou *GPGPUs* [2], através da arquitetura CUDA [3], junto do Teorema Chinês do Resto e da Transformada de Fourier. Quando comparado com a implementação estado da arte em *CPUs*, essa estratégia implicou em ganhos de até 20% de velocidade na adição e 58% na multiplicação homomórfica, operações críticas do ponto de vista de desempenho para se avaliar qualquer função sobre dados cifrados.

O documento é organizado como se segue. Na Seção 2 as bases teóricas para o trabalho são apresentadas. É oferecida uma definição formal para a propriedade homomórfica em criptossistemas, assim como é apresentado o criptossistema YASHE, usado neste trabalho, além de uma breve passagem pelo Teorema Chinês do Resto e a Transformada Rápida de Fourier. Nas Seções 3 e 4 são apresentadas, respectivamente, a implementação CUYASHE e uma análise do ganho de desempenho sobre o estado da arte. Por fim, a Seção 5 apresenta as conclusões e expectativas para trabalhos futuros.

2. Base teórica

2.1. Criptografia Homomórfica

Um esquema criptográfico homomórfico pode ser definido como na Definição 1.

Definição 1. *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:*

$$D\left(E\left(m_1\right) \circ E\left(m_2\right)\right) = D\left(E\left(m_1 \diamond m_2\right)\right) \Rightarrow D\left(E\left(m_1\right) \circ E\left(m_2\right)\right) = m_1 \diamond m_2.$$

A operação $\circ$ é equivalente à operação $\diamond$ no espaço de criptogramas.

Para esquemas criptográficos que possuam a propriedade apresentada, temos que a nuvem (ou um atacante) pode manipular dados cifrados aplicando o operador $\diamond$ sem aprender nada sobre o texto claro correspondente.

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

 ©2015 SBC — Soc. Bras. de Computação

Criptossistemas parcialmente homomórficos são criptossistemas que satisfazem a Definição 1 para operações de adição ou multiplicação. Existem diversas propostas de esquemas desse tipo, dentre elas o esquema de Paillier [Paillier 1999] e ElGamal [ElGamal 1985]. Estes são bastante conhecidos e se caracterizam não apenas pelo bom desempenho como também por atingirem o mais alto nível de segurança em comparação a outros esquemas parcialmente homomórficos.

Criptossistemas completamente homomórficos são criptossistemas que satisfazem a Definição 1 para ambas as operações de adição e multiplicação.

Em uma posição intermediária aos criptossistemas anteriormente citados, existem criptossistemas ditos ligeiramente homomórficos [4] e completamente homomórficos em nível [5].

Criptossistemas ligeiramente homomórficos são aqueles que satisfazem a Definição 1 para uma quantidade limitada de operações de adição e multiplicação. No trabalho de [Gentry 2010] é demonstrado como construir um criptossistema completamente homomórfico a partir de um criptossistema ligeiramente homomórfico, usando uma técnica chamada de *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 são evoluções da proposta anterior de Gentry. Em [Brakerski et al. 2012], define-se 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. Nesse mesmo trabalho é demonstrado como construir um criptossistema completamente homomórfico a partir de um completamente homomórficos em nível. Este trabalho usou o criptossistema YASHE, definido na Seção 2.2, que é definido como um criptossistema completamente homomórfico em nível.

2.2. YASHE - *Yet Another Somewhat Homomorphic Encryption*

YASHE é um criptossistema completamente homomórfico em nível baseado em reticulados [Bos et al. 2013]. Ele opera sobre elementos de um anel gerado por um polinômio ciclotômico. O esquema é descrito na Definição 2.

Definição 2. *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. *Escolha um inteiro* q *que define o espaço dos coeficientes dos criptogramas.*
2. *Escolha um inteiro* t *que define o espaço dos coeficientes dos textos claros. Garanta que* $1 < t < q$.
3. *Escolha um inteiro* $w > 1$, *que define o tamanho da palavra nas operações de troca de chave.*

[4] Do Inglês, *Somewhat Homomorphic Encryption*.

[5] Do Inglês, *Leveled Fully Homomorphic Encryption*.

 ©2015 SBC — Soc. Bras. de Computação

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

Geração de chaves:

1. *Amostre dois polinômios* f' *e* g *da distribuição* χ_{key} *e defina* $f = \left[tf' + 1\right]_q$.
2. *Se* f *não for inversível em* R_q, *escolha um novo* f'.
3. *Defina* $h = [tgf^{-1}]_q$.
4. *Defina* $\gamma = [PowersOf_{w,q}(f) + e + h \cdot s]_q \in R_q^{\log_w q}$.
5. *Retorne* $(pk, sk, evk) = (h, f, \gamma)$.

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

1. *Amostre dois polinômios* s *e* e *da distribuição* χ_{err}.
2. *Retorne 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* $c \in R_q$,

Retorne $m = \left[\left\lfloor \frac{t}{q} \cdot [fc]_q \right\rceil\right] \in R_t$.

Adição: *Sejam os criptogramas* c_1 *e* c_2,

Retorne $c_{add} = [c_1 + c_2]_q$.

Multiplicação: *Sejam os criptogramas* c_1 *e* c_2 *e a chave* evk,

1. *Defina* $c^*_{mult} = \left[\left\lfloor \frac{t}{q} \cdot c_1 \cdot c_2 \right\rceil\right]_q$.
2. *Retorne* $c_{mult} = KeySwitch(c^*_{mult}, evk)$.

PowersOf $_{w,q}$ *:*

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

KeySwitch: *Seja o criptograma* c *e a chave* evk,

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

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

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

2.3. Teorema Chinês do Resto

Como ferramenta para simplificar a manipulação de polinômios com coeficientes bastante grandes, utilizou-se o Teorema Chinês do Resto, ou *CRT* [6]. Dessa forma, pode-se representar grandes coeficientes inteiros usando resíduos arbitrariamente pequenos e que podem ser manipulados utilizando uma aritmética mais simples.

O funcionamento do *CRT* implementado se baseia em mapear um polinômio A em um conjunto de polinômios $\{A_0, A_1, \ldots, A_{l-1}\}$ tal que $A_i \in R_{p_i}$, para um primo p_i escolhido arbitrariamente, como pode ser visto nas Definições 3 e 4.

[6] Do Inglês, *Chinese Remainder Theorem*.

 ©2015 SBC — Soc. Bras. de Computação

Definição 3. - *Transformação CRT*
Seja x *um polinômio em* R_q *e* $\{p_0, p_1, \ldots, p_{l-1}\}$ *um conjunto de* l *primos.*

Define-se: $CRT : x \rightarrow \{x \mod p_0, x \mod p_1, \ldots, x \mod p_{l-1}\}$.

Definição 4. - *Transformação Inversa da CRT , ou ICRT*
Seja x *um conjunto de* l *polinômios residuais oriundos do CRT ,* $\{p_0, p_1, \ldots, p_{l-1}\}$ *o respectivo conjunto de primos e* $M = \Pi_{i=0}^{l-1} p_i$.

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

Para o funcionamento correto da *ICRT*, é necessário que o produto dos primos seja maior do que o maior possível coeficiente de um polinômio residual, inclusive após uma operação de adição ou multiplicação. Ou seja, para o YASHE é preciso que $M = \Pi_{i=0}^{l-1} p_i > n \cdot q^2$, onde n e q são os parâmetros de formação do anel.

No domínio de resíduos do *CRT* as operações de adição e multiplicação são definidas como na Definição 5.

Definição 5. - *Operações sobre o domínio do CRT*
Sejam x *e* y *polinômios em* R_q *com* X *e* Y *sendo seus respectivos conjuntos de resíduos,* $\{p_0, p_1, \ldots, p_{l-1}\}$ *um conjunto de* l *primos e uma operação* $\diamond$.

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

2.4. Transformada Rápida de Fourier

Neste trabalho, foi utilizado o algoritmo da Transformada Rápida de Fourier, ou *FFT*[7] [Cochran et al. 1967], para a redução da complexidade computacional de uma operação de multiplicação polinomial. Ela faz uso da Transformada Discreta de Fourier [8] para reduzir a complexidade computacional de $\Theta(N^2)$, de um algoritmo trivial de multiplicação polinomial, para $\Theta(N \log N)$, com N representando o grau dos polinômios operandos.

Dessa forma, a multiplicação de dois polinômios é feita como na Definição 6.

Definição 6. - *Multiplicação de polinômios usando FFT*

Sejam dois polinômios p *e* q *escritos como:*

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

[7] Do Inglês, *Fast Fourier transform.*
[8] Do Inglês, *Discrete Fourier transform.*

 ©2015 SBC — Soc. Bras. de Computação

Além disso, seja $\omega_N = e^{i\frac{2\Pi}{N}}$ *a* n*-ésima raíz complexa de* 1. *O produto*

$$(p \cdot q)(x) = p(x) \cdot q(x) = c_0 + c_1 x + \ldots + c_{2N-2} x^{2N-2}$$

é calculado da seguinte forma:

1. *Avalie eficientemente* $p(x)$ *e* $q(x)$ *nos pontos* $\omega_{2N}^0, \omega_{2N}^1, \ldots, \omega_{2N}^{2N-1}$.
2. *Compute os valores de* $p(x) \cdot q(x)$ *nesses pontos através da multiplicação ponto-a-ponto*

$$\begin{aligned}(p \cdot q)(\omega_{2N}^0) &= p(\omega_{2N}^0) \cdot q(\omega_{2N}^0),\\ (p \cdot q)(\omega_{2N}^1) &= p(\omega_{2N}^1) \cdot q(\omega_{2N}^1),\\ &\vdots\\ (p \cdot q)(\omega_{2N}^{2N-1}) &= p(\omega_{2N}^{2N-1}) \cdot q(\omega_{2N}^{2N-1}).\end{aligned}$$

3. *Aplique a transformação inversa da Transformada Discreta de Fourier e recupere os coeficientes* c_i *tal que*

$$(p \cdot q)(x) = \sum_{i=0}^{2N-2} c_i x^i.$$

A etapa de avaliação dos polinômios em $2N$ pontos tem custo $\Theta\left(N \log N\right)$ quando realizada com uma estratégia elegante, como o método de dividir-para-conquistar proposto por [Cooley and Tukey 1965].

3. A implementação CUYASHE

A implementação proposta, batizada de CUYASHE [Alves and Aranha 2015], foi produzida em C++, com foco na revisão C++11 e utilizou-se a biblioteca *NTL* compilada com suporte a *GMP* para prover a aritmética de grandes inteiros. Dessa forma, foi possível concentrar os esforços de otimização na aritmética polinomial e apenas herdar as operações elementares sobre inteiros da *NTL*. O modelo de classes usado representa polinômios-genéricos e polinômios-criptogramas com as classes *Polynomial* e *Ciphertext*. Como visto na Figura 1, há uma relação de herança entre essas classes. Assim, pôde-se reutilizar operações comuns e especializar aquelas exclusivas a criptogramas. As distribuições probabilísticas usadas pelo YASHE, Gaussiana discreta e estreita, foram concentradas na classe *Distribution*. Essas três classes são agregadas à classe *YASHE*, responsável pelas operações de cifração, decifração e troca de chave do criptossistema.

3.1. Manipulando inteiros grandes

Operações sobre inteiros muito grandes podem ser bastante caras e adicionar um grau de complexidade considerável ao trabalho, principalmente com o uso da arquitetura CUDA que não possui suporte nativo a operações com operandos maiores do que 64 *bits*. Por isso, o Teorema Chinês do Resto, como visto na Seção 2.3, foi aplicado para gerar polinômios com coeficientes menores e mais adequados à plataforma alvo. Essa estratégia implicou na troca do

 ©2015 SBC — Soc. Bras. de Computação

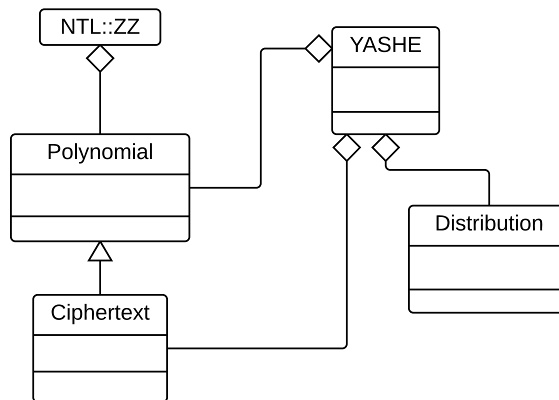

Figura 1. Diagrama de classes simplificado. A classe *Polynomial* (usada para a representação de polinômios genéricos) é superclasse da *Ciphertext* (usada para a representação de criptogramas). A classe YASHE contém a implementação do criptossistema e agrega instâncias dessas duas classes, além da *Distribution*, responsável pelas distribuições probabilísticas requeridas.

custo computacional de operar sobre dois polinômios com coeficientes arbitrariamente grandes pelo custo de operar sobre vários polinômios residuais com coeficientes tão pequenos quanto se queira, conforme visto na Definição 5.

A capacidade de paralelismo de *GPGPUs* se mostra grande o suficiente para mascarar o custo adicional de processamento e ainda prover ganho de velocidade em relação a outras implementações conhecidas, que fazem uso da aritmética de grandes inteiros em *CPUs*, como demonstrado na Seção 4.

3.2. Localidade de memória

Na modelagem utilizada, toda cópia dos resíduos do *CRT* para a memória *global* da *GPU* é feita concatenando os polinômios em um único vetor de inteiros. Dessa forma, as operações polinomiais podem tirar proveito do acesso de memória *coalesced*, onde vários blocos de memória consecutivos são lidos em uma única instrução *read* e aproveitados por diferentes *threads*, assim como podem ser aplicadas com a chamada de um único *CUDA Kernel*. A vantagem dessa modelagem é que operações consecutivas são realizadas sem a necessidade da cópia de dados entre a memória da *GPU* e a memória principal, além de tornar desnecessário a aplicação do *CRT* e *ICRT* nesse momento. Como pode ser visto nas Tabelas 2 e 3, isso pode implicar em ganhos bastante expressivos de desempenho em algoritmos que tirem proveito dessa característica.

O diagrama de fluxo na Figura 2 expõe as verificações feitas pela CUYASHE que precedem todas as operações de adição e multiplicação. Essas verificações tem como objetivo reduzir custos de processamento redundante e desnecessários. Inicialmente, verifica-se se os operandos já possuem valores calculados e atualizados dos polinômios residuais do *CRT* . Da mesma maneira, verifica-se se esses resíduos (caso já tenham sido calculados anteriormente)

 ©2015 SBC — Soc. Bras. de Computação

existem em um estado atualizado na memória da *GPU*. Resultados negativos a esses predicados são resolvidos e só então é dado início ao processamento de cada operação. Dessa maneira, como já comentado, resultados prévios das etapas de aplicação do *CRT* e da cópia de dados são reaproveitados e permite-se adiar a aplicação do *ICRT*, que passa a ser necessária apenas no momento em que os dados são requeridos para pela *CPU*. Essa otimização implica em ganhos consideráveis de desempenho para operações consecutivas.

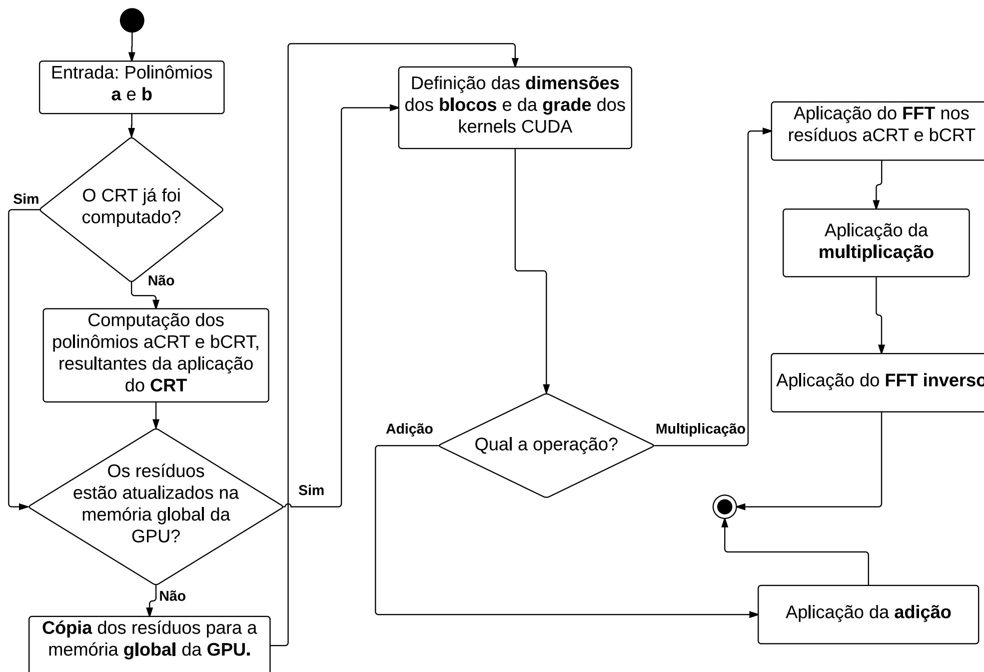

Figura 2. Diagrama de fluxo das operações de adição e multiplicação. Antes de aplicar a operação requisitada, a CUYASHE tem de calcular os polinômios residuais do *CRT* e copiar para a memória da *GPU*. Essas duas etapas preliminares são descartadas, se possível.

3.3. Aritmética polinomial

As operações de multiplicação e adição polinomial são realizadas em paralelo pela *GPU*. Para isso, foi definido um modelo de *CUDA Threads* que tira proveito da estrutura de memória e implementa as operações em paralelo, escondendo o custo computacional oriundo do *CRT* e reduzindo o tempo necessário para cada operação.

A aritmética polinomial implementada tira proveito dessas duas operações básicas e de características do YASHE para implementar versões eficientes das operações de subtração, divisão e resto.

Adição e subtração

Na adição, cada bloco de *CUDA Threads* é composto por *threads* em uma única dimensão de tamanho $\lceil \frac{N}{32} \rceil$, para um polinômio de grau $N-1$. O $kernel$ de execução soma elemento a

elemento dois vetores de inteiros, compostos pela concatenação dos coeficientes dos polinômios residuais do *CRT* de cada operando. Essa concatenação é feita cuidadosamente para garantir o correto alinhamento dos coeficientes. A subtração é realizada da mesma maneira, apenas substituindo a operação aplicada em cada coeficiente.

Multiplicação

A multiplicação polinomial é feita através do algoritmo *FFT*, descrito na Seção 2.4, implementado pela biblioteca *cuFFT* [NVIDIA 2015a]. Ela é uma biblioteca proprietária da NVIDIA e implementa uma variação não-normalizada do *FFT* .

Uma vez que o algoritmo usado opera sobre valores reais, enquanto que o YASHE opera sobre inteiros, foi necessário implementar $kernels$ exclusivos para a conversão dos coeficientes de inteiro para complexo e de complexo para inteiro. Dessa forma, foi usado o modo de funcionamento *Complex-to-Complex* da *cuFFT*.

Divisão e resto

O cálculo do quociente e do resto em uma divisão de polinômios pode ser consideravelmente caro sem a exploração de alguma propriedade especial dos polinômios.

O YASHE faz uso de polinômios ciclotômicos para definir o anel de criptogramas R_q. 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.

Fazendo uso disso, a divisão por esses polinômios pode ser feita como no Lema 1, 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 1. *Divisão e resto pelo n-ésimo polinômio ciclotômico. Sejam $\Phi_n(x)$ o n-ésimo polinômio ciclotômico com n potência de 2 e $P(x) = \sum_{i=0}^{m-1} a_i x^i$, com $m > n$. Então o quociente $Q(x)$ e o resto $R(x)$ da divisão de $P(x)$ por $\Phi_n(x)$ são dados por:*

$$Q(x) = \frac{P(x)}{x^{\frac{n}{2}}} = \sum_{i=0}^{\frac{n}{2}-1} a_{i+n/2} x^i,$$

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

Demonstração. Sejam o polinômio $P(x)$ e $\Phi_n(x)$ o n-ésimo polinômio ciclotômico com 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,

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

 ©2015 SBC — Soc. Bras. de Computação

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 1,

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

o que encerra a demonstração do Lema 1. □

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. Em casos em que a divisão não pode ser acelerada por essa propriedade, a implementação passa o cálculo da operação para a *NTL* .

3.4. cuFFT

Pela necessidade de se trabalhar com valores reais na *FFT*, foi necessário que se implementasse rotinas de conversão e que se tomassem cuidados para evitar possíveis erros de precisão causados pela aritmética de ponto flutuante. Além das citadas, a aplicação do *CRT* foi feita usando primos de tamanho 9 *bits*, enquanto a *cuFFT* foi executada utilizando precisão dupla. Apesar da mantissa de valores *float* ser de 23 *bits* e a de valores *double* ser 53 *bits*, foi constatado empiricamente que operandos próximos, mas inferiores, dessas mantissas ainda poderiam resultar em erros de precisão. Esses erros são evidenciados com a aplicação da *ICRT* e comprometem de forma irrecuperável o funcionamento do programa, dado o contexto. Utilizando operandos com coeficientes limitados a 9 *bits* e o modo de precisão dupla da *cuFFT*, conseguiu-se evitar completamente esse tipo de erro, ao custo do aumento da quantidade de resíduos e da diminuição de desempenho da *cuFFT* referente a precisão dupla. O modo de precisão única requer a diminuição dos coeficientes para valores ainda menores, o que pode impedir que se satisfaça a condição comentada na Seção 2.3 e atrapalhe o funcionamento da *ICRT* .

4. Resultados

Para medir o impacto das otimizações, os tempos das operações da CUYASHE foram comparados com os trabalhos de [Lepoint and Naehrig 2014] e [Bos et al. 2013], como visto na Tabela 1. Ambos são relativos a implementações que tiram proveito apenas da *CPU* da máquina e tem pouco ou nenhum esforço em processamento paralelo. Apesar disso, do ponto de vista da aplicação, o principal interesse é o tempo de processamento, e uma vez que no contexto de computação na nuvem é comum a presença de *GPGPUs* , como descrito por [NVIDIA 2015b], a comparação se mantêm relevante.

Os valores apresentados neste trabalho para a CUYASHE foram obtidos através do cálculo do tempo médio para cada operação. Para isso foram usadas 100 amostras de execuções isoladas. A máquina utilizada na obtenção dos tempos da CUYASHE possui uma *CPU* Intel Xeon E5-2630 @ 2.60GHz e uma *GPGPU* NVIDIA GeForce GTX TITAN Black @ 0.98GHz. Foi utilizada a versão 7.0 do kit de desenvolvimento CUDA, além das versões 9.1.0 e 6.0.0 das bibliotecas *NTL* e *GMP* respectivamente. Por sua vez, o trabalho de [Lepoint and Naehrig 2014]

 ©2015 SBC — Soc. Bras. de Computação

usa uma *CPU* Intel Core i7-2600 @ 3.4GHz, enquanto que [Bos et al. 2013] utiliza uma *CPU* Intel Core i7-3520M @ 2893.484 MHz.

A Tabela 1 demonstra que há pouco ou nenhum ganho de desempenho durante as etapas de cifração e decifração, uma vez que a atual implementação da cuYASHE possui pouca reutilização de dados na memória da *GPU* e, em especial na operação de cifração, há uma intensa cópia de operandos para a memória da *GPU*. Apesar disso, foi obtido ganho de 20% em velocidade na adição e 58% na multiplicação homomórfica em relação ao trabalho de [Lepoint and Naehrig 2014].

A motivação principal para a utilização de um criptossistema homomórfico é justamente aplicar operações aritméticas homomórficas sobre criptogramas. Dessa forma, essas operações são o alvo principal para melhorias de desempenho uma vez que serão aplicadas recorrentemente em diferentes tipos de algoritmo, ao passo que as operações de geração de chave, cifração e decifração são consideravelmente menos usadas e por isso ganhos de desempenho são menos relevantes.

Tabela 1. Tempos para o cuYASHE e comparação com os trabalhos de [Lepoint and Naehrig 2014] e [Bos et al. 2013], respectivamente. A máquina utilizada na obtenção dos tempos da cuYASHE possui uma *CPU* Intel Xeon E5-2630 @ 2.60GHz e uma *GPGPU* NVIDIA GeForce GTX TITAN Black @ 0.98GHz e possui instalado a versão 7.0 do kit de desenvolvimento CUDA, além das versões 9.1.0 e 6.0.0 das bibliotecas *NTL* e *GMP* respectivamente, enquanto que o trabalho de [Lepoint and Naehrig 2014] usa uma *CPU* Intel Core i7-2600 @ 3.4GHz e [Bos et al. 2013] utiliza uma *CPU* Intel Core i7-3520M @ 2893.484 MHz. Os valores apresentados para a cuYASHE foram obtidos pelo cálculo do tempo médio de 100 amostras de execuções isoladas. Parâmetros: $\mathbf{R} = \mathbb{Z}[\mathbf{X}] / (x^{4096}+1)$, $q = 2^{127}-1$, $w = 2^{32}$, $t = 2^{10}$.

Operação	**cuYASHE (ms)**	**LN (ms)**	**BLLN (ms)**
Cifração	32,28	16	27
Decifração	13,87	15	5
Adição Homomórfica	0,56	0,7	0,024
Multiplicação Homomórfica + KeySwitch	20,29	49	31

A Tabela 2 demonstra uma importante característica da cuYASHE. Para a aplicação de operações de adição ou multiplicação polinomial há gasto de tempo no cálculo dos resíduos do *CRT*, na cópia desses valores para a memória da *GPU* e posteriormente na aplicação da *ICRT*. Contudo, operações consecutivas, com operandos já na memória da *GPU*, conseguem reaproveitar valores calculados anteriormente e evitar tempo desperdiçado em operações redundantes. Dessa forma, as otimizações realizadas nesse sentido implicam em ganhos de 14 vezes em velocidade na reutilização de operandos em operações de adição e de 4 vezes para operações de multiplicação. O ganho de desempenho em algoritmos que utilizem a cuYASHE para operar sobre dados cifrados acompanha a capacidade do algoritmo em tirar proveito dessa característica da implementação.

5. Conclusão e trabalhos futuros

Uma implementação do criptossistema YASHE foi apresentada, a cuYASHE, e se estabeleceu uma comparação de seu desempenho em relação a duas outras implementações co-

[9]Do Inglês, *overhead*.

Tabela 2. Tempos para a aritmética polinomial em um anel de grau 4096. As colunas registram os tempos médios para cada operação, considerando o custo necessário de copia dados para a memória da *GPU*. Pode-se perceber os ganhos significativos de desempenho que podem ser obtidos por algoritmos que possam manter operandos na memória da *GPU*.

Operação	Com sobrecarga[9] de memória (ms)	Sem sobrecarga de memória (ms)
Adição	19,24	1,32
Multiplicação	32,28	7,01

nhecidas, demonstrando ganhos de até 20% na adição e 58% na multiplicação homomórfica. Além disso, conseguiu-se demonstrar que é possível obter ganhos de desempenho bastante expressivos ao aplicá-la em um algoritmo que possa manter os operandos na memória da *GPU*. Nesses casos, notou-se ganhos de até 14 e 4 vezes para adição e multiplicação homomórfica respectivamente, ao mesmo tempo que adiciona-se o benefício da segurança oriundo do criptossistema utilizado.

Em trabalhos futuros se espera a investigação dos possíveis ganhos com a substituição do *FFT* e da biblioteca *cuFFT* por alternativas que proporcionem aumento na densidade aritmética. Uma alternativa promissora é a *NTT* [10], que aplica a mesma lógica da *FFT* mas usa corpos finitos ao invés do corpo complexo. A Tabela 3 apresenta resultados preliminares da substituição da *cuFFT* por uma implementação própria da *NTT* .

Tabela 3. Tempos preliminares para a aritmética polinomial em um anel de grau 4096 utilizando a transformada NTT ao invés da FFT. As colunas registram os tempos médios para cada operação, considerando o custo necessário de copia dados para a memória da *GPU*. Há ganho de velocidade em todos os tempos em comparação com a Tabela 2 por causa da redução do custo computacional da aritmética da *NTT* e pela redução do número de resíduos do *CRT*.

Operação	Com sobrecarga[11] de memória (ms)	Sem sobrecarga de memória (ms)
Adição	6,28	1,00
Multiplicação	14,36	6,27

Há um ganho de velocidade evidente por conta da substituição das transformadas. A aritmética de corpos finitos, usada pela *NTT*, possui uma implementação mais eficiente do que a aritmética de ponto-flutuante, usada pela *FFT*. Isso reduz o tempo necessário para a aplicação da operação de multiplicação. Além disso, por não haver mais a necessidade de se limitar o tamanho dos primos usados pelo *CRT* em 9 *bits*, pode-se reduzir a quantidade de polinômios residuais, o que atinge diretamente o custo de cópia de dados entre as memórias e consequentemente reduz também o tempo necessário para ambas as operações com sobrecarga de cópia de memória.

[10] Acrônimo de *Number-Theoretic Transform*.

[11] Do Inglês, *overhead*.

 ©2015 SBC — Soc. Bras. de Computação

Referências

Alves, P. and Aranha, D. (2015). cuYASHE. `https://github.com/pdroalves/cuYASHE`. Acessado: 08/09/2015.

Bos, J., Lauter, K., Loftus, J., and Naehrig, M. (2013). Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme. In Stam, M., editor, *Cryptography and Coding*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer Berlin Heidelberg.

Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2012). (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. ACM.

Buyya, R. (2009). 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. IEEE Computer Society.

Cochran, W. T., Cooley, J. W., Favin, D. L., Helms, H. D., Kaenel, R. A., Lang, W. W., George C. Maling, J., Nelson, D. E., Rader, C. M., and Welch, P. D. (1967). What is the fast Fourier transform? *IEEE Transactions on Audio and Electroacoustics*, 15:45–55.

Cooley, J. W. and Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301.

ElGamal, T. (1985). A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In Blakley, G. and Chaum, D., editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Berlin Heidelberg.

Gentry, C. (2010). Computing Arbitrary Functions of Encrypted Data. *Commun. ACM*, 53(3):97–105.

Hoffstein, J., Pipher, J., and Silverman, J. (1998). NTRU: A ring-based public key cryptosystem. In Buhler, J., editor, *Algorithmic Number Theory*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer Berlin Heidelberg.

Lepoint, T. and Naehrig, M. (2014). A Comparison of the Homomorphic Encryption Schemes FV and YASHE. In Pointcheval, D. and Vergnaud, D., editors, *Progress in Cryptology – AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 318–335. Springer International Publishing.

NVIDIA (2015a). CUDA Toolkit Documentation. `http://docs.nvidia.com/cuda/cufft/`. Acessado: 12/08/2015.

NVIDIA (2015b). GPU Cloud Computing. `http://www.nvidia.com/object/gpu-cloud-computing-services.html`. Acessado: 11/09/2015.

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

 ©2015 SBC — Soc. Bras. de Computação