Perancangan dan Percepatan Enkripsi Homomorfik Total Skema RNS-CKKS secara Paralel dengan GPU

Jevant Jedidia Augustine
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132
jevantjedidia@gmail.com

Rinaldi Munir
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132
rinaldi@staff.stei.itb.ac.id

Infall Syafalni
Program Studi Teknik Elektro
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132
infall@staff.stei.itb.ac.id

***Abstrak*— Enkripsi homomorfik adalah skema enkripsi yang memungkinkan dilakukannya komputasi pada *ciphertext*. Salah satu skema enkripsi homomorfik yang banyak digunakan adalah CKKS. *Ciphertext* skema CKKS direpresentasikan dalam sebuah polinom. Untuk keamanan yang baik, derajat serta koefisien dari polinom perlu memiliki nilai yang tinggi. Akibat hal ini, CKKS dikembangkan lebih lanjut menjadi RNS-CKKS untuk meningkatkan efisiensi dalam operasi dan penyimpanan data. Walaupun demikian, beberapa operasi seperti perkalian *ciphertext* masih memiliki waktu eksekusi yang tinggi untuk dijalankan terutama pada derajat polinom yang tinggi. Maka dari itu paralelisasi dapat diutilisasi untuk mempercepat skema RNS-CKKS. Teknik utama yang digunakan untuk mengoptimasi implementasi secara paralel adalah perataan *array* dan *kernel fusion*. Pengujian dilakukan dengan membandingkan waktu eksekusi implementasi serial dan paralel kemudian menghitung percepatan atau perlambatannya. Alhasil, proses NTT, iNTT, enkripsi, dekripsi, dan perkalian mengalami percepatan hingga 3.7x, 3.7x, 3.26x, 3.96x dan 4.68x secara berurutan, sedangkan proses *decoding* dan penjumlahan mengalami perlambatan hingga 0.57x, dan 0.86x secara berurutan. Tidak terjadi percepatan yang signifikan pada proses *encoding*.**

Kata kunci—enkripsi homomorfik, RNS-CKKS, GPU, percepatan

I. PENDAHULUAN

Enkripsi homomorfik merupakan sebuah skema enkripsi yang memungkinkan dilakukannya komputasi terhadap *ciphertext* tanpa harus mengetahui pesan aslinya. Telah dikembangkan berbagai macam skema enkripi homomorfik seperti BGV dan BFV. Kedua skema tersebut hanya dapat melakukan enkripsi pada bilangan bulat sehingga kurang praktis dan fleksibel bila diaplikasikan ke dunia nyata. Maka dari itu dirancang sebuah skema enkripsi homomorfik yang dapat mengenkripsi bilangan kompleks, yaitu Cheon-Kim-Kim-Song atau yang biasanya dikenal sebagai CKKS [1]. Kemampuan CKKS untuk melakukan enkripsi terhadap bilangan kompleks menetapkan posisi CKKS sebagai skema enkripsi homomorfik yang *state of the art.*

Ciphertext skema CKKS direpresentasikan dengan sebuah polinom dalam sistem aritmatika modular. Keamanan dari skema CKKS ditentukan dari tingginya derajat polinomial tersebut sedangkan ketepatan proses dekripsi dari skema CKKS ditentukan dari besarnya bilangan yang dikalikan terhadap pesan awal yang mempengaruhi besarnya koefisien polinom pada *ciphertext*. Akibat kedua hal tersebut, idealnya, polinom pada *ciphertext* CKKS memiliki derajat serta koefisien yang tinggi, akan tetapi hal tersebut menyebabkan degradasi performa [2]. Alhasil, skema CKKS dikembangkan lebih lanjut lagi menjadi RNS-CKKS. RNS-CKKS menggunakan konsep *residue number system* dalam implementasinya yang berguna untuk mengurangi biaya yang diperlukan untuk menyimpan serta mengomputasi data pada CKKS dengan cara memecah koefisien polinom menjadi bilangan yang jauh lebih kecil sehingga performa dari skema enkripsi tersebut menjadi jauh lebih baik.

Walaupun performa RNS-CKKS jauh lebih baik dibandingkan dengan skema CKKS dasar. Waktu yang dibutuhkan untuk menjalankan proses yang berat seperti perkalian masih memakan waktu serta biaya komputasi yang cukup signifikan sehingga masih membatasi penggunaan praktikal RNS-CKKS dalam dunia nyata. Paralelisme dengan bantuan *graphical processing unit* atau yang biasa disebut sebagai GPU menjadi salah satu solusi untuk mempercepat proses tersebut. Paralelisasi skema RNS-CKKS secara menyeluruh akan menghasilkan sebuah skema RNS-CKKS yang lebih cepat dan efisien untuk kegunaan praktikal seperti *machine learning*.

II. DASAR TEORI

A. Enkripsi Homomorfik

Enkripsi homomorfik merupakan sebuah skema enkripsi yang memungkinkan dilakukannya operasi pada *ciphertext*. Enkripsi homomorfik dibagi menjadi 2 macam berdasarkan jumlah operasi yang dapat dilakukan terhadap *ciphertext*, yaitu enkripsi homomorfik parsial dan enkripsi homomorfik total. Enkripsi homomorfik parsial adalah skema enkripsi homomorfik yang hanya memungkinkan operasi penjumlahan

atau perkalian tetapi tidak keduanya antara 2 *ciphertext*. Contoh dari algoritma enkripsi homomorfik parsial adalah algoritma RSA, algoritma ElGamal, dan algoritma Pallier. Enkripsi homomorfik total adalah skema enkripsi homomorfik yang memungkinkan proses perkalian dan penjumlahan antara 2 *ciphertext*. Contoh dari algoritma enkripsi homomorfik total adalah BFV, BGV, dan CKKS.

B. Residue Number System

Residue number system (RNS) merupakan sebuah sistem representasi angka yang banyak digunakan dalam kriptografi, pemrosesan sinyal digital, dan aplikasi lainnya dimana operasi aritmatika yang efisien sangat penting. RNS menyatakan bahwa apabila terdapat sebuah bilangan bulat *A* dan sebuah set $\{m_1, m_2, \dots, m_k\}$ yang berisikan *k* buah bilangan bulat yang koprima satu sama dengan yang lainnya, maka bilangan bulat *A* dapat direpresentasikan dalam RNS sebagai $\{a_1, a_2, \dots, a_k\}$ dengan

$$a_k \equiv A \bmod m_k \tag{1}$$

Bila dimiliki sebuah modulo $M = \prod_{i=0}^{k} m_i$, maka persamaan 2.11 akan dipenuhi [3].

$$A \bmod M = \sum_{i=0}^{k} a_i \tag{2}$$

Persamaan berikut menyatakan bahwa bila terdapat sebuah bilangan bulat A serta modulo M yang sangat besar, hasil modulo dari bilangan tersebut dapat direpresentasikan dalam bilangan yang lebih kecil. Hal tersebut menjadi dorongan utama penggunaan RNS pada banyak skema kriptografi. Dengan menggunakan RNS, komputasi modulo tidak dilakukan pada bilangan bulat yang besar, tetapi pada bilangan yang lebih kecil sehingga efisiensi dalam komputasi serta penyimpanan bilangan menjadi jauh lebih baik.

C. Number Theoretic Transform

Number Theoretic Transform (NTT) adalah bentuk terspesialisasi dari DFT yang mengevaluasikan polinom pada *nth root of unity* dalam sebuah *ring*. Dengan $n = 2^k$, $k \in Z$, dan $q \equiv 1 \ (mod\ 2n)$, sebuah *nth root of unity* ω pada *ring* Z_q dinyatakan dalam Persamaan (3).

$$\omega^n \equiv 1 (mod\ q) \tag{3}$$

Serupa dengan FFT, NTT dapat dipercepat dengan menggunakan metode *divide-and-conquer*. NTT dan iNTT dapat digunakan untuk melakukan perkalian antara 2 polinom secara efisien. NTT mengubah polinom menjadi representasi *point-value* dan iNTT mengubah representasi *point-value* menjadi polinom. Perkalian polinom secara tradisional memiliki kompleksitas $O(n^2)$, sedangkan perkalian polinom dalam representasi *point-value* memiliki kompleksitas O(n) [4].

D. Enkripsi Homomorfik CKKS

Skema CKKS (Cheon-Kim-Kim-Song) merupakan skema enkripsi homomorfik total yang pertama kali dikemukakan pada tahun 2017 oleh Jung Hee Cheon, Andrey Kim, Miran Kim, dan Yongsoo Song. Perbedaan utama dari skema CKKS dengan skema enkripsi homomorfik lainnya seperti BGV dan BFV adalah skema CKKS mendukung operasi aproksimasi pada bilangan kompleks, sedangkan skema BGV dan BFV hanya mendukung operasi pada bilangan bulat [1]. CKKS menggunakan polinomial untuk merepresentasikan sebuah *ciphertext*. Berdasarkan fakta tersebut, Cheon et al. mengembangkan skema dasar CKKS lebih lanjut lagi menjadi skema RNS-CKKS yang merepresentasikan polinomial CKKS dengan *double-CRT* untuk meningkatkan efisiensi implementasi dari aritmatika polinomial pada skema CKKS.

Sesuai dengan namanya, sebuah representasi polinomial pada RNS-CKKS memiliki 2 lapisan CRT. Lapisan pertama CRT menggunakan RNS untuk mendekomposisi sebuah polinomial menjadi sebuah *tuple* polinomial dengan moduli yang lebih kecil. Lapisan kedua mengubah setiap polinom hasil dekomposisi menjadi vektor bilangan modulo dengan menggunakan *number theoretic transform* (NTT) [2]. Secara umum, skema CKKS dan RNS-CKKS terdiri atas 7 algoritma utama, yaitu: *encoding*, pembangkitan kunci, enkripsi, penjumlahan *ciphertext*, perkalian *ciphertext*, dekripsi, dan *decoding*.

Proses *encoding* mengambil sebuah vektor bilangan kompleks kemudian memroses vektor tersebut menjadi koefisien polinom dengan FFT. Sesudah itu koefisien polinom dikalikan dengan sebuah skala untuk menjaga bit presisi. Proses *encoding* digambarkan dengan Persamaan (4) [2].

$$pt = \Delta * (iFFT(m)) \tag{4}$$

Proses *decoding* merupakan *inverse* dari proses *encoding*. Koefisien polinom dibagi dengan skala kemudian dipetakan menjadi bilangan kompleks dengan FFT. Proses *decoding* digambarkan dengan Persamaan (5) [2].

$$m = FFT(\Delta^{-1} * pt) \tag{5}$$

Proses enkripsi memerlukan sebuah kunci publik *pk* dan polinom acak *a* dan *e*, sedangkan proses dekripsi memerlukan sebuah kunci privat *s*. Proses enkripsi digambarkan dengan Persamaan (6) sedangkan proses dekripsi digambarkan dengan Persamaan (7) [2].

$$ct = pt + pk = (pt - a * s + e, a) \tag{6}$$
$$pt = ct_0 + ct_1 * s \tag{7}$$

Penjumlahan dua *ciphertext* dilakukan dengan menjalankan Persamaan (8).

$$ct + ct' = (ct_0 + ct'_0, ct_1 + ct'_1) \tag{5}$$

Proses perkalian antara dua *ciphertext*, *c* dan *c'*dilakukan dengan menjalankan prosedur berikut [2]:

1. Untuk $0 \leq j \leq l$ hitung

$$d_0^{(j)} \leftarrow c_0^{(j)} c'^{(j)}_0 \ (mod\ q_j)$$
$$d_1^{(j)} \leftarrow c_0^{(j)} c'^{(j)}_1 + c_1^{(j)} c'^{(j)}_0 (mod\ q_j)$$
$$d_2^{(j)} \leftarrow c_1^{(j)} c'^{(j)}_1 \ (mod\ q_j).$$

2. Hitung

$$ModUp_{C_l \to D_l}(d_2^{(0)}, \dots, d_2^{(l)}) = (\tilde{d}_2^{(0)}, \dots, \tilde{d}_2^{(k-1)}, d_2^{(0)}, \dots, d_2^{(l)})$$

3. Hitung

$$\widetilde{ct} = (\widetilde{ct}^{(0)} = (\widetilde{ct}_0^{(0)}, \widetilde{ct}_1^{(0)}), \dots, \widetilde{ct}^{(k+l)} = (\widetilde{ct}_0^{(k+l)}, \widetilde{ct}_1^{(k+l)}))$$

dimana $\widetilde{ct}^{(i)} = \tilde{d}_2^{(i)} \cdot evk^{(i)}$ dan $\widetilde{ct}^{(k+j)} = \tilde{d}_2^{(j)} \cdot evk^{(k+j)}$

4. Hitung

$$(\hat{c}_0^{(0)}, \dots, \hat{c}_0^{(l)}) \leftarrow ModDown_{D_l \to C_l}(\tilde{c}_0^{(0)}, \dots, \tilde{c}_0^{(k+l)})$$
$$(\hat{c}_1^{(0)}, \dots, \hat{c}_1^{(l)}) \leftarrow ModDown_{D_l \to C_l}(\tilde{c}_1^{(0)}, \dots, \tilde{c}_1^{(k+l)})$$

5. Hitung $ct_{mult}^{(j)} \leftarrow \left(\hat{c}_0^{(j)} + d_0^{(j)}, \hat{c}_1^{(j)} + d_1^{(j)}\right)$.

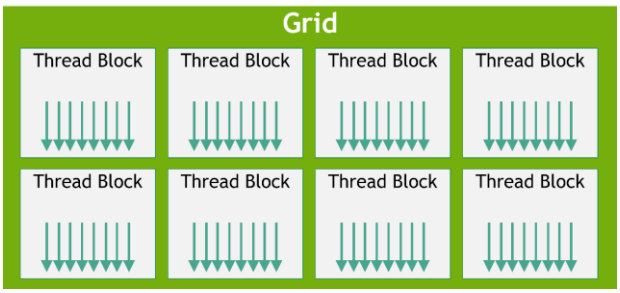
Setelah dilakukan perkalian, *ciphertext* hasil perlu di-*rescale* untuk menjaga skala yang dikalikan saat proses *encoding* tetap konstan. Persamaan (6) menggambarkan proses *rescaling*.

$$c'^{(j)}_i \leftarrow q_l^{-1} \cdot (c_i^{(j)} - c_i^{(l)}) \qquad (6)$$

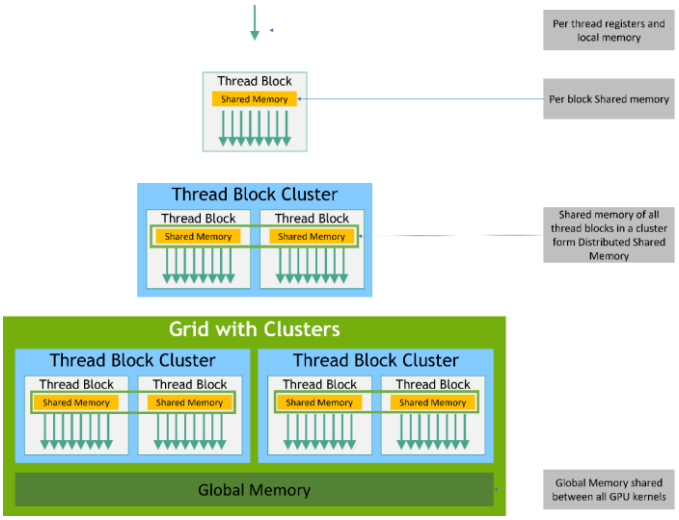
E. *GPU* dan *CUDA*

Graphical processing unit atau yang biasa disebut sebagai GPU merupakan prosesor yang didesain untuk melakukan sebuah task secara paralel. Paralelisme pada *programming* itu sendiri berarti pembagian sebuah tugas atau *task* menjadi bagian-bagian yang kecil yang akan diproses secara bersamaan sehingga waktu yang dibutuhkan untuk mengolah sebuah *task* yang besar berkurang secara signifikan. Pada November 2006, Nvidia memperkenalkan CUDA sebagai *platform* komputasi dan model pemrograman yang dapat digunakan untuk memanfaatkan kemampuan paralelisme Nvidia GPU untuk dapat menyelesaikan permasalahan dengan waktu komputasi yang cepat [5]. Model pemrograman CUDA meluaskan C++ dengan memperbolehkan pengguna untuk memanggil sebuah fungsi C++ bernama kernel yang dapat dieksekusi sebanyak N kali secara paralel oleh N *threads* CUDA. Sebuah *kernel* didefinisikan menggunakan deklarasi *global* dan jumlah *thread* CUDA yang akan mengeksekusi kernel tersebut dispesifikasikan dengan konfigurasi <<< … >>>. Setiap *thread* diberikan thread ID unik yang dapat diakses melalui *kernel* dengan menggunakan fungsi bawaan CUDA [5].

ID dari sebuah *thread* merupakan vektor dengan 3 komponen sehingga dapat didefinisikan dengan *thread index* satu dimensi, dua dimensi, atau tiga dimensi yang membentuk blok *thread* satu dimensi, dua dimensi, atau tiga dimensi yang dipanggil *thread block*. Jumlah *thread* pada setiap *block* memiliki batas tertentu. Untuk GPU terkini, batas tersebut adalah 1024 *threads*. Setiap *block* itu sendiri disusun dalam sebuah grid satu dimensi, dua dimensi, atau tiga dimensi [5]. Gambar II.1 menggambarkan ilustrasi dari konsep tersebut dan Gambar II.2 menggambarkan hierarki ketiga objek tersebut.



Gambar II.1 Ilustrasi hubungan antara thread, block, dan grid [5]



Gambar II.2 Ilustrasi hierarki memori pada CUDA [5]

III. Rancangan Solusi

Implementasi RNS-CKKS secara serial dilakukan dengan bahasa pemrograman C++, sedangkan secara paralel dilakukan dengan bahasa pemrograman CUDA. Implementasi serial mengacu pada implementasi yang dilakukan oleh andru47 [6].

A. Rancangan Solusi Serial

Proses NTT mengubah representasi polinom menjadi representasi *point-value*, sedangkan proses iNTT mengubah representasi *point-value* menjadi representasi polinom. Gambar III.1 merupakan algoritma proses NTT dan iNTT. Perbedaan dari kedua operasi terdapat pada masukan variabel *R* dan *S*. Pada

proses iNTT, kedua variabel tersebut di-*inverse* sebelum dimasukkan sebagai argumen fungsi.

ALGORITHM 1: NTT

```
    Input: Vector of Integer I, Roots R, Modulus Q, Scale S
    Output: Vector of Integer O
1   bitReverse(I)
2   N ← len(I)
3   for len=2 to N do
4       for i=0 to N do
5           for j=i to i + len/2 do
6               u ← I[j]
7               v ← I[j + len/2] * R[(j-i) * (N/len)] % Q
8               O[j] ← u + v % Q
9               O[j + len/2] ← u – v % Q
10          endfor
11          i ← i + len
12      endfor
13      len ← len * 2
14  endfor
15  for i=0 to len(O) do
16      O[i] ← O[i] * S % Q
17  endfor
```

Gambar III.1 Algoritma NTT/iNTT Serial

Proses *encoding* dan *decoding* terdiri atas 2 tahap utama, yaitu iFFT dan scaleUp untuk *encoding*, dan FFT dan scaleDown untuk *decoding*. Gambar III.2 merupakan algoritma scaleUp sedangkan Gambar III.3 merupakan algoritma scaleDown. Algoritma FFT dan iFFT serupa dengan algoritma NTT dan iNTT dengan perbedaan variabel *R* sesuai dengan konstanta FFT.

ALGORITHM 2: SCALEUP

```
    Input: Vector of Complex V, Scale S, Modulo mod
    Output: Vector of Integer I
1   for j=0 to len(V) do
2       V[j] ← V[j] * S
3       sign ← false
4       if V[j].real < 0 then
5           sign ← true
6           V[j].real ← -(V[j].real)
7       endif
8       V[j] ← V[j] + 0.5
9       if sign then
10          I[j] ← (mod – V[j]) % mod
11      else
12          I[j] ← V[j] % mod
13      endif
14  endfor
```

Gambar III.2 Algoritma scaleUp Serial

ALGORITHM 3: SCALEDOWN

```
    Input: Vector of Integer I, Modulo mod, Scale S
    Output: Vector of Complex V
1   qHalf ← mod / 2
2   for j=0 to len(I) do
        bigIntCoeff ←I[j] % mod
3       sign ← false
4       if bigIntCoeff >= qHalf then
5           sign ← true
6           bigIntCoeff ← mod - bigIntCoeff
7       endif
8       quotient ← bigIntCoeff / S
9       remainder ← bigIntCoeff % S
10      realPart ← (remainder * 1.0) / (1.0 * S)
11      if sign then
12          V[j] ← -(quotient + realPart)
13      else
14          V[j] ← quotient + realPart
15      endif
16  endfor
```

Gambar III.3 Algoritma scaleDown Serial

Proses enkripsi dan dekripsi memerlukan kunci publik dan kunci privat secara berurutan. Sebelum dilakukan enkripsi, polinom diubah terlebih dahulu menjadi representasi *point-value* dengan NTT dan sesudah dilakukan dekripsi, representasi tersebut diubah menjadi polinom dengan iNTT. Gambar III.4 merupakan algoritma proses enkripsi sedangkan Gambar III.5 merupakan algoriitma proses dekripsi.

ALGORITHM 4: ENCRYPTION

```
    Input: Plaintext m, PublicKey pk, Level l, Modulus Qs
    Output: Ciphertext ct
1   v ← dist(enc)
2   e ← dist(err)
3   for j=0 to l do
4       NTT(m)
5       ct0[j]← v · pk0[j] + (m + e0) % Qs[j]
6       ct1[j]← v · pk1[j] + e1 % Qs[j]
7   endfor
```

Gambar III.4 Algoritma Enkripsi Serial

ALGORITHM 5: DECRYPTION

```
    Input: Ciphertext ct, SecretKey sk, Modulo Q
    Output: Plaintext m
1   m ← ct0[0] + ct1[0] · sk % Q
2   iNTT(m)
```

Gambar III.5 Algoritma Dekripsi Serial

Penjumlahan kedua *ciphertext* dilakukan dengan mengambil polinom per-*level* dari setiap *ciphertext* kemudian menjumlahkan 2 polinomial tersebut dalam modulo *level*-nya sesuai dengan Gambar III.6.

ALGORITHM 6: CIPHERTEXT ADDITION

```
Input: Ciphertext ct, Ciphertext ct', Modulus Qs
Output: Ciphertext ct''
l ← min(getLevel(ct), getLevel(ct'))
for j=0 to l do
    ct''0[j]= ct0[j]+ ct'0[j] % Qs[j]
    ct''1[j]= ct1[j]+ ct'1[j] % Qs[j]
endfor
```

Gambar III.6 Algoritma Penjumlahan Ciphertext Serial

Sesuai dengan penjelasan pada Bagian II.C, Proses perkalian terdiri atas beberapa langkah. Langkah 1, 3, dan 5 merupakan operasi perkalian dan penjumlahan polinom biasa, sehingga dapat merujuk kepada proses yang melakukan proses serupa seperti enkripsi dan dekripsi. Langkah 2 merupakan ModUp yang digunakan untuk mengubah polinom dari sebuah basis RNS menjadi basis RNS lainnya dengan modulo yang lebih besar. Langkah 4 merupakan ModDown yang digunakan untuk mengubah basis RNS polinom menjadi basis yang lebih kecil. Gambar III.7 merupakan algoritma ModUp sedangkan Gambar III.8 merupakan algoritma ModDown.

ALGORITHM 7: MODUP

```
Input: Vector of Polynomial d2, Vector of Integer qHat, Vector of Integer qHatInverse, Modulus Q, Special Modulus P
Output: Vector of Polynomial d2'
res ← initiateVector(len(P))
for i=0 to len(P) do
    for j=0 to d2[0].getDegree do
        convRes ← 0
        for k=0 to len(d2) do
            curRes ← (d2[k][j] * qHatInverse[len(d2) - 1][k]) mod Q[k]
            curRes ← (curRes * qHat[i][len(d2) - 1][k]) mod P[i]
            convRes ← (convRes + curRes) mod P[i]
        endfor
        res[i].push(convRes)
    endfor
    res[i].convertToPolynomials()
endfor
d2' ← res.append(d2)
```

Gambar III.7 Algoritma ModUp Serial

Setelah dilakukan perkalian, proses *rescaling* dilakukan untuk menjaga skala yang dikalikan dengan pesan awal pada proses *encoding* tetap konstan. Gambar III.9 merupakan algoritma *rescaling.*

B. Rancangan Solusi Paralel

Implementasi paralel menggunakan dua teknik untuk melakukan percepatan terhadap skema RNS-CKKS. Teknik pertama adalah perataan *array* dimana koefisien polinom yang telah dipecah menjadi representasi RNS digabung menjadi sebuah vektor yang panjang. Dengan meratakan vektor, pemanggilan kernel yang dilakukan memroses polinom berkurang menjadi sekali pemanggilan saja. Teknik kedua adalah *kernel fusion*. Teknik ini dilakukan dengan menggabung dua *kernel* menjadi satu dengan tujuan mengurangi pemanggilan *kernel*.

ALGORITHM 8: MODDOWN

```
Input: Vector of Polynomial d2, Vector of Integer pHat, Vector of Integer pHatInverse, Vector of Integer pInverse, Modulus Q, Special Modulus P
Output: Vector of Polynomial d2'
res ← initiateVector(len(Q))
for i=0 to len(Q) do
    for j=0 to d2[0].getDegree do
        convRes ← 0
        for k=0 to len(P) do
            curRes ← (d2[k][j] * pHatInverse[k]) mod P[k]
            curRes ← (curRes * pHat[i][k]) mod Q[i]
            convRes ← (convRes + curRes) mod Q[i]
        endfor
        res[i].push(convRes)
    endfor
    res[i].convertToPolynomials()
endfor
for i=0 to len(res) do
    for j=0 to len(res[i]) do
        res[i][j] ← ((Q[i]-res[i][j]) + d2[i + len(P)][k]) mod Q[i]
        res[i][j] ← (pInverse[i] * res[i][j]) mod Q[i]
    endfor
endfor
d2' ← res
```

Gambar III.8 Algoritma ModDown Serial

ALGORITHM 9: RESCALE

```
Input: Ciphertext ct, Modulus Qs
Output: Ciphertext ct'
newLevel ← ct.getLevel - 1
for i=0 to newLevel do
    qInv ← modInverse(Qs[ct.getLevel], Qs[i])
    ct'0[i] ← ct0[i] - ct0[ct.getLevel] * qInv % Qs[i]
    ct'1[i] ← ct1[i] - ct1[ct.getLevel] * qInv % Qs[i]
endfor
```

Gambar III.9 Algoritma Rescale Serial

Ozerk et al. mengusulkan teknik paralelisasi yang optimal untuk NTT dan iNTT [7]. Pada teknik yang diusulkan, NTT dan iNTT dilakukan pada 2 *method* yang terpisah, yaitu NTTPar dan iNTTPar. Ide utama dalam paralelisasi NTT dan iNTT adalah membagi perhitungan *terms* pada setiap babak iterasi sehingga dibangun N/2 buah *thread* dengan setiap *thread* melakukan log(N) buah iterasi. Gambar III.10 dan Gambar III.11 merupakan algoritma NTT dan iNTT paralel.

Algoritma scaleUp paralel pada proses *encoding* digambarkan pada Gambar III.12, sedangkan algoritma scaleDown paralel pada proses *decoding* digambarkan pada Gambar III.13. Paralelisasi proses FFT dan iFFT serupa dengan paralelisasi proses NTT dan iNTT. Gambar III.14 merupakan algoritma enkripsi paralel dan Gambar III.15 merupakan algoritma dekripsi paralel. Paralelisasi kedua proses tersebut menggunakan teknik *kernel fusion* dan perataaan vektor untuk mempercepat paralelisasi. Algoritma penjumlahan yang

ditunjukkan pada Gambar III.16 menggunakan teknik yang serupa dengan proses enkripsi dan dekripsi.

ALGORITHM 10: NTTPAR

```
    Input: Vector of Integer I, Roots R, Modulus Q
    Output: Vector of Integer O
1   i ← threadIdx.x + blockIdx.x * blockDim.x
2   N ← len(I)
3   for len=2 to N do
4       step ← (N/len) / 2
5       psi_step ← i / step
6       j ← psi_step * step * 2 + i % step
7       u ← I[j]
8       v ← I[j + step] * R[len + psi_step] % Q
9       O[j] ← u + v % Q
10      O[j + step] ← u – v % Q
11      len ← len * 2
12  endfor
13  bitReverse(O)
```

Gambar III.10 Algoritma NTT Paralel

ALGORITHM 11: INTTPAR

```
    Input: Vector of Integer I, Roots R, Modulus Q, Scale S
    Output: Vector of Integer O
1   i ← threadIdx.x + blockIdx.x * blockDim.x
2   N ← len(I)
3   bitReverse(O)
4   for len=N/2 to 1 do
5       step ← (N/len) / 2
6       psi_step ← i / step
7       j ← psi_step * step * 2 + i % step
8       u ← I[j]
9       v ← I[j + step]
10      O[j] ← u + v % Q
11      v ← u – v % Q
12      O[j + step] ← v * R[len + psi_step] % Q
13      if len=1 then
14          O[i] ← O[i] * S % Q
15          O[i + N/2] ← O[i + N/2] * S % Q
16      endif
17      len ← len / 2
18  endfor
```

Gambar III.11 Algoritma iNTT Paralel

Serupa dengan implementasi serial, tahap 1, 3, dan 5 pada proses perkalian memiliki implementasi yang serupa dengan proses dekripsi, enkripsi, atau penjumlahan dalam sisi operasi dasar polinom. Tahap 2, yaitu ModUp, digambarkan dengan Gambar III.17, sedangkan tahap 4, yaitu ModDown, digambarkan dengan Gambar III.18. Proses *rescaling* secara paralel digambarkan dengan Gambar III.19.

ALGORITHM 12: SCALEUPPAR

```
    Input: Vector of Complex V, Scale S, Modulus Qs, Degree deg
    Output: Vector of Integer I
1   idx ← threadIdx.x + blockDim.x * blockIdx.x
2   i ← idx % deg
3   level ← idx / deg
4   mod ← Qs[level]
5   V[idx] ← V[idx] * S
6   sign ← false
7   if V[idx].real < 0 then
8       sign ← true
9       V[idx].real ← -(V[idx].real)
10  endif
11  V[idx] ← V[idx] + 0.5
12  if sign then
13      I[idx] ← (mod – V[idx]) % mod
14  else
15      I[idx] ← V[idx] % mod
16  endif
```

Gambar III.12 Algoritma scaleUp Paralel

ALGORITHM 13: SCALEDOWNPAR

```
    Input: Vector of Integer I, Modulo mod, Scale S
    Output: Vector of Complex V
    idx ← threadIdx.x + blockDim.x * blockIdx.x
1   qHalf ← mod / 2
2   bigIntCoeff ← I[idx] % mod
3   sign ← false
4   if bigIntCoeff >= qHalf then
5       sign ← true
6       bigIntCoeff ← mod - bigIntCoeff
7   endif
8   quotient ← bigIntCoeff / S
9   remainder ← bigIntCoeff % S
10  realPart ← (remainder * 1.0) / (1.0 * S)
11  if sign then
12      V[idx] ← -(quotient + realPart)
13  else
14      V[idx] ← quotient + realPart
15  endif
```

Gambar III.13 Algoritma scaleDown Paralel

ALGORITHM 14: ENCRYPTPARALLEL

```
    Input: Plaintext m, PublicKey pk, Degree n, Vector of Integer v, Error e,
    Modulus Qs
    Output: Ciphertext ct
1   idx ← threadIdx.x + blockDim.x * blockIdx.x
2   level ← idx / deg
3   for j=0 to level do
4       NTT(m[j*n: j*n+n])
5   endfor
6   ct0[idx]← pk0[idx] * v[idx] + m[idx] + e0[idx] % Qs[level]
7   ct1[idx]← pk1[idx] * v[idx] + e1[idx] % Qs[level]
```

Gambar III.14 Algoritma Enkripsi Paralel

ALGORITHM 15: DECRYPTPARALLEL

Input: *Ciphehrtext ct, SecretKey sk, Modulo Q*
Output: *Plaintext m*

```
1  idx ← threadIdx.x + blockDim.x * blockIdx.x
2  m[idx] ← ct1[idx] * sk[idx] + ct0[idx] % Q
3  iNTT(m)
```

Gambar III.15 Algoritma Dekripsi Paralel

ALGORITHM 16: PARALEL CIPHERTEXT ADDITION

Input: *Ciphertext ct, Ciphertext ct', Degree deg, Modulus Qs*
Output: *Ciphertext ct''*

```
1  idx ← threadIdx.x + blockDim.x * blockIdx.x
2  level ← idx / deg
3  ct''0[idx]= ct0[idx]+ ct'0[idx] % Qs[level]
4  ct''1[idx]= ct1[idx]+ ct'1[idx] % Qs[level]
```

Gambar III.16 Algoritma Penjumlahan Ciphertext Paralel

ALGORITHM 17: MODUPPARALLEL

Input: *Polynomial d2, Vector of Integer qHat, Vector of Integer qHatInverse, Modulus Q, Special Modulus P, Degree deg*
Output: *Polynomial res*

```
1  idx ← thread.idx + blockIdx.x * blockDim.x
2  level ← idx / deg
3  k ← idx % deg
4  convRes ← 0
5  for j=0 to len(Q) do
6      curRes ← d2[j * deg + k]
7      curRes ← curRes * qHatInverse[(len(Q)-1) * len(Q) + j] % Q[j]
8      curRes ← curRes * qHat[level * (len(Q) * len(Q)) + (len(Q)-1) * len(Q)
       + j] % P[level]
9      convRes ← convRes + curRes % P[level]
10 endfor
11 res[idx] ← convRes
```

Gambar III.17 Algoritma ModUp Paralel

ALGORITHM 18: MODDOWNPARALLEL

Input: *Polynomial d2, Vector of Integer pHat, Vector of Integer pHatInverse, Vector of Integer pInv, Modulus Q, Special Modulus P, Degree deg*
Output: *Polynomial res*

```
1  idx ← thread.idx + blockIdx.x * blockDim.x
2  level ← idx / deg
3  k ← idx % deg
4  convRes ← 0
5  for j=0 to len(P) do
6      curRes ← d2[j * deg + k]
7      curRes ← curRes * pHatInverse[j] % P[j]
8      curRes ← curRes * pHat[level * len(P) + j] % Q[level]
9      convRes ← convRes + curRes % Q[level]
10 endfor
11 res[idx] ← convRes
12 res[idx] ← (Q[level]–res[idx]) + d2[(level + len(P)) * deg + k] % Q[level]
13 res[idx] ← res[idx] * pInv[level] % Q[level]
```

Gambar III.18 Algoritma ModDown Paralel

ALGORITHM 19: RESCALEPARALLEL

Input: *Ciphertext ct, Ciphertext ctSub, Modulus Q, Degree deg, Level l*
Output: *Ciphertext ct'*

```
1  idx ← threadIdx.x + blockDim.x * blockIdx.x
2  level ← idx / deg
3  qInv ← modInverse(Q[l], Q[level])
4  ct'0[idx] ← (ct0[idx] – ctSub0[idx]) * qInv % Q[level]
5  ct'1[idx] ← (ct1[idx] – ctSub1[idx]) * qInv % Q[level]
```

Gambar III.19 Algoritma Rescale Paralel

IV. PENGUJIAN

A. Lingkungan Pengujian

Pengujian dilakukan pada lingkungan perangkat dengan spesifikasi sebagai berikut.

1. Perangkat Keras:
   a. *Processor:* AMD Ryzen 9 5950X 16-Core Processor
   b. GPU: NVIDIA GeForce RTX 3090 Ti
2. Perangkat Lunak:
   a. Sistem Operasi: Ubuntu 22.04
   b. *Compiler* Kode Serial: GCC Version 11.3.0
   c. *Compiler* Kode Parallel: NVCC V12.5.40

B. Hasil Pengujian

Pengujian implementasi RNS-CKKS secara serial dan paralel dilakukan untuk mengukur performa setiap versi. Performa diuji dengan mengukur waktu eksekusi proses-proses yang ada di RNS-CKKS pada derajat polinom yang berbeda. Selain untuk mengetahui percepatan atau perlambatan antara implementasi serial dan paralel pada derajat polinom tertentu, hal ini juga dilakukan untuk mengobservasi peningkatan waktu eksekusi seiring meningkatnya derajat polinom. Tabel IV.1 merupakan hasil pengujian implementasi serial dan paralel.

Tabel IV.1 Perbandingan Hasil Implementasi Serial dan Paralel

Operasi	Derajat Polinom (2^N)	Waktu Eksekusi Serial (ms)	Waktu Eksekusi Paralel (ms)	*Speedup*
NTT	10	1,1664	0,52879	2,20571
	11	2,5888	0,83686	3,09351
	12	5,7362	1,67330	3,42806
	13	12,4317	3,45623	3,59689
	14	27,1848	7,23717	3,75627
iNTT	10	1,4229	0,63438	2,24292
	11	3,0899	0,98869	3,12522
	12	6,6437	1,95737	3,39419
	13	14,3667	4,04242	3,55400
	14	30,8995	8,31446	3,71635

Encoding	10	0,6364	0,80909	0,78658
	11	1,3162	1,52013	0,86581
	12	2,6842	2,81756	0,95267
	13	5,3006	5,60039	0,94647
	14	11,2184	11,15867	1,00535
Decoding	10	0,1370	1,90788	0,07180
	11	0,2865	2,03976	0,14046
	12	0,6977	2,58489	0,26991
	13	1,2926	3,36681	0,38391
	14	3,1641	5,46795	0,57866
Enkripsi	10	17,7055	7,49705	2,36166
	11	38,0537	13,02347	2,92193
	12	82,5913	26,71893	3,09112
	13	176,8433	55,62030	3,17947
	14	377,0243	115,30600	3,26977
Dekripsi	10	1,7193	0,70463	2,44007
	11	3,7651	1,08779	3,46123
	12	7,8702	2,11043	3,72921
	13	15,7671	4,34131	3,63187
	14	35,8865	9,05989	3,96103
Penjumlahan	10	0,1009	0,16986	0,59382
	11	0,2030	0,26720	0,75983
	12	0,3891	0,48003	0,81067
	13	0,6692	0,81287	0,82320
	14	1,3689	1,57944	0,86670
Perkalian	10	69,6860	27,05980	2,57526
	11	146,9937	39,57310	3,71448
	12	313,8543	74,37683	4,21979
	13	663,8180	146,07500	4,54436
	14	1403,6267	299,79267	4,68199

C. Analisis

Berdasarkan hasil pengujian yang dilakukan, operasi NTT, iNTT, enkripsi, dekripsi, dan perkalian mengalami percepatan yang signifikan terutama pada derajat polinom tertinggi, yaitu 2^{14}. Walaupun demikian, operasi *encoding*, *decoding*, dan penjumlahan tidak mengalami percepatan melainkan mengalami perlambatan. Hal ini dikarenakan *overhead* yang dihasilkan oleh pemrograman paralel dalam inisialisasi serta alokasi vektor pada GPU. Sebagai perbandingan, Tabel IV.2 merupakan perbandingan ketiga proses tersebut tanpa menghitung *overhead* yang dihasilkan pada proses paralel.

Tabel IV.2 Perbandingan Hasil Implementasi Serial dan Paralel Tanpa Overhead

Operasi	**Derajat Polinom (2^N)**	**Waktu Eksekusi Serial (ms)**	**Waktu Eksekusi Paralel (ms)**	***Speedup***
Encoding	10	0,6364	0,16629	3,82715
	11	1,3162	0,31228	4,21462
	12	2,6842	0,61977	4,33103
	13	5,3006	1,25983	4,20742
	14	11,2184	2,55177	4,39631
Decoding	10	0,1370	0,18179	0,75350
	11	0,2865	0,34160	0,83871
	12	0,6977	0,67611	1,03191
	13	1,2926	1,36551	0,94658
	14	3,1641	2,78044	1,13798
Penjumlahan	10	0,1009	0,00760	13,27682
	11	0,2030	0,00784	25,88432
	12	0,3891	0,00776	50,16682
	13	0,6692	0,00816	82,03756
	14	1,3689	0,00843	162,37190

Bila *overhead* dihiraukan, operasi penjumlahan dan *encoding* mengalami percepatan yang signifikan, khususnya operasi penjumlahan. Operasi *decoding* memiliki *speedup* yang lebih baik dibandingkan dengan hasil pada Tabel IV.1, akan tetapi hasil yang didapat masih kurang baik. Terdapat dua faktor yang memengaruhi perlambatan ini. Faktor pertama adalah implementasi paralel yang kurang efisien akibat perlunya memanggil beberapa kernel dalam proses *decoding*. Proses *decoding* terdiri atas beberapa tahap, yaitu scaleDown dan FFT. Pada implementasi, kedua tahap tersebut diimplementasikan pada kernel yang berbeda sehingga perlu memanggil kernel sebanyak dua kali yang berkontribusi dalam waktu eksekusi paralel. Faktor kedua adalah waktu eksekusi serial yang relatif cepat. Proses *encoding* dan *decoding* secara paralel memiliki waktu eksekusi yang serupa, akan tetapi *speedup* yang didapat berbeda. Hal ini dikarenakan secara serial, proses *encoding* memakan waktu yang lebih signifikan karena perlu membagi polinom menjadi representasi RNS dibandingkan dengan proses *decoding* yang hanya perlu mengoperasikan satu buah polinom.

V. KESIMPULAN

Skema enkripsi homomorfik RNS-CKKS berhasil diimplementasikan secara serial maupun paralel. Dengan menggunakan teknik seperti perataan vektor dan *kernel fusion*, implementasi proses pada skema secara paralel mengalami percepatan dalam waktu eksekusi. Walaupun demikian, pada beberapa proses seperti penjumlahan, *encoding*, dan *decoding*, percepatan yang didapa kurang optimal. Hal tersebut dikarenakan *overhead* yang dihasilkan oleh pemrograman

paralel relatif besar sehingga berpengaruh secara signifikan dalam waktu eksekusi pemrograman paralel.

UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa karena atas berkat dan karunia-Nya lah penulis dapat menyelesaikan *paper* ini dengan tepat waktu, Penulis juga mengucapkan terima kasih kepada Dr. Ir. Rinaldi Munir, M.T. dan Dr. Eng. Infall Syafalni, S.T., M.Sc. selaku pembimbing penulis yang telah membimbing penulis selama mengerjakan *paper* ini.

REFERENSI

[1] Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2017). Homomorphic Encryption for Arithmetic of Approximate Numbers. Advances in Cryptology – ASIACRYPT 2017, 409-437.

[2] Cheon, J. H., Han, K., Kim, A., Kim, M., & Song, Y. (2018). A Full RNS Variant of Approximate Homomorphic Encryption. Selected Areas in Cryptography: … annual international workshop, SAC, 347-368.

[3] H. L. Garner. (1959). *The Residue Number System*. IRE Transactions on Electronic Computers, vol. EC-8, no. 2, 140-147.

[4] Cormen, T. H., & Leiserson, C. E. (2009). *Introduction to algorithms, 3rd Edition*. The MIT Press.

[5] CUDA C++ Programming Guide. (2023, October 11). Retrieved October 11, 2023, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[6] https://github.com/andru47/smkhe

[7] Ozerk, O., Elgezen, C., Mert, A., C., Ozturk, E., & Savas, E. (2021). Efficient Number Theoretic Transform Implementation on GPU for Homomorphic Encryption. Cryptology ePrint Archive, 124.