CentraleSupélec

université PARIS-SACLAY

Circuit Optimization

QComp

06/10/2025

Centrale-Supélec

Étienne BONNAND

Professeur référent : Benoît Valiron

Sommaire

1 Optimizing the oracle

1.1 Removing useless controls

```python
def removeUselessControl(circ):
    valuation = { k : 0 for k in circ.ancillas} # Ancillas are originally at 0

    new_circ = circ.copy() # Create a copy of circ

    new_gates = [] # To store the (possibly updated) gates

    for g in new_circ.gates:
        new_control_list = []
        skip_gate = False

        target = g["target"]
        control_list = g["ctl"]

        if target in valuation:
            if len(control_list) != 0:
                del valuation[target]
            else:
                valuation[target] = (valuation[target] + 1) % 2

        for ctl in control_list:
            if ctl in valuation:
                if valuation[ctl] == 0:
                    skip_gate = True
                    break
            else:
                new_control_list.append(ctl)

        if not skip_gate:
            new_gates.append({"target" : target ,"ctl" : new_control_list})

    new_circ.gates = new_gates # Update the list of gates
    return new_circ
```

Listing 1: Fonction removeUselessControl

Après avoir testé la cellule, le code renvoyé est bien le bon.

1.2 Optimizing The oracle for Inv

Pour calculer un circuit implémentant l'inverse , j'ai utilisé la fonction prédéfinie : `polyCirc(coeffs, n_frac, N)`. La fonction `InvCirc` est définie comme suit :

```python
def InvCirc(n_frac, N, degree, at_x):
    x = sympy.symbols('x')
    f = sympy.asin(1/(16*x))
    poly = sum( (f.diff(x, k).subs(x, at_x) / math.factorial(k)) * (x - at_x)**k for k
        in range(degree+1))
    coeffs = sympy.poly(poly, x).all_coeffs()
    return polyCirc(coeffs, n_frac, N)
```

Listing 2: Définition de la fonction `InvCirc` pour la construction du circuit de l'opérateur Inv

En utilisant la fonction `removeUselessControl` précédemment construite pour `degree = 2`, `N = 16`, `n_frac = 10` plusieurs fois jusqu'à convergence, cela a permis de passer de **38837** gates dans le circuit initial à uniquement **15720** dans le circuit final. Soit un ratio de 2.47.
L'optimisation du circuit est réalisée en appliquant itérativement la fonction `removeUselessControl` jusqu'à ce que le nombre de portes se stabilise. Cette convergence garantit que toutes les simplifications possibles basées sur la propagation de la constante (ancillas à 0 ou 1) ont été effectuées.

```
previous = len(new_circ.gates)
new_circ = removeUselessControl(new_circ)
while previous != len(new_circ.gates):
    previous = len(new_circ.gates)
    new_circ = removeUselessControl(new_circ)
print(len(new_circ.gates))
```

Listing 3: Application itérative de la fonction `removeUselessControl` jusqu'à convergence

Afin de vérifier que l'optimisation n'a pas altéré la fonctionnalité logique du circuit un test d'équivalence est effectué. Ce test compare les résultats du circuit original (`invcirc`) et du circuit optimisé (`new_circ`) sur un ensemble d'entrées aléatoires.

```
import copy
nombre_tests = 10
for i in range(nombre_tests):
    input_random = {j : random.randint(0,1) for j in range(N)}

    state_orig = copy.deepcopy(input_random)
    state_opti = copy.deepcopy(input_random)

    val_orig = invcirc.run(state_orig)
    val_opti = new_circ.run(state_opti)

    result_orig = getValue(val_orig, res)
    result_opti = getValue(val_opti, res)

    assert(result_orig == result_opti), f"Échec au test {i+1} : Les résultats diffèrent.
        Original: {result_orig}, Optimisé: {result_opti}"

print(f"Test de robustesse réussi sur {nombre_tests} entrées aléatoires : Les circuits
    sont logiquement équivalents.")
```

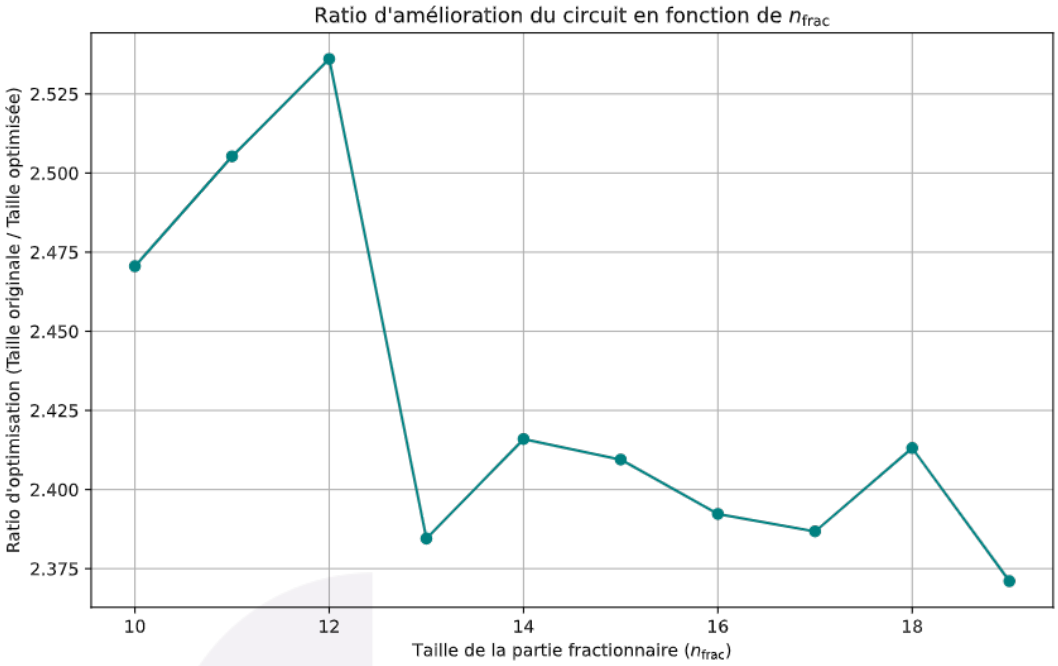
Listing 4: Vérification de l'équivalence logique entre le circuit original et le circuit optimisé

J'ai ensuite réalisé successivement les calculs en augmentant N et n_{frac} de 1 à chaque itération en conservant un polynôme de degré 2. Le ratio correspondant est présenté dans le graphique suivant: 1.2. Nous pouvons observer une légère diminution du ratio tout en conservant une valeur relativement égale.
Le tableau 1 résume les résultats de la simplification du circuit en utilisant `removeUselessControl` pour un polynôme de degree = 3, tout en augmentant progressivement la taille du registre (N) et la précision fractionnaire (n_{frac}).

Table 1: Évolution du ratio de simplification pour degree = 3

Cas	N	$n_{\textbf{frac}}$	**Gates Initiales**	**Gates Finales**	**Ratio** ($\frac{\textbf{Initial}}{\textbf{Final}}$)
1	16	10	77558	38052	$\mathbf{2,04}$
2	17	11	87709	42896	$\mathbf{2,04}$

Malgré une augmentation de la complexité du circuit l'efficacité de la simplification reste extrêmement stable.



1.3 High-level optimization

La fonction ci-dessous implémente le calcul du polynôme de manière plus astucieuse afin de minimiser le nombre de multiplications, réduisant ainsi la complexité du circuit d'évaluation du polynôme.

```
def new_PolyCirc(coeffs, n_frac, N):
    inputs = [i for i in range(N)]
    circ = RevCirc(inputs)

    res = constFPsigned(circ, n_frac, N, coeffs[0])

    for c in coeffs[1:]:
        res = multFPSigned(circ, n_frac, inputs, res)
        bin_c = constFPsigned(circ, n_frac, N, c)
        res = adderFPSigned(circ, n_frac, res, bin_c)

    return res, circ
```

Listing 5: Évaluation du polynôme par factorisation imbriquée

Analyse de la Complexité

Dans la session de programmation précédente (QComp-TP-4-Optim), l'implémentation naïve du calcul de l'inverse, qui reposait sur la puissance x^k calculée par multiplications répétées, présentait une complexité en portes dominée par les multiplications de la puissance, soit :

$$\text{Complexité}_{\text{Naïve}} = O(d^2 N^2)$$

(où d est le degré du polynôme et N est la taille du registre).
Ici, avec la méthode de (`new_PolyCirc`), il n'y a qu'une seule multiplication par degré. Le nombre total de multiplications devient proportionnel au degré d. Par conséquent, la complexité du circuit est réduite à :

$$\text{Complexité}_{\text{Horner}} = O(dN^2)$$

Cette approche permet un gain de complexité moyen en $O(d)$ par rapport à l'implémentation naïve.)

2 2.1 - Implementing the successor operator

```python
def successor(N):

    q = QuantumRegister(N)
    qc = QuantumCircuit(q)

    if N == 1:
        qc.x(q[0])
    else:
        cNx_gate = XGate().control(N-1)
        qubits_to_use = list(q[1:]) + [q[0]]
        qc.append(cNx_gate, qubits_to_use)
        qc.append(successor(N - 1), q[1:])

    return Operator(qc)
```

Listing 6: Implémentation de l'opérateur de succession

Après implémentation et test la construction de l'opérateur fonctionne bien.

2.1 2.2 - Optimizing Trotterization

```python
n = 3 # Register size
t = 1/2
a = 1
b = 2

def Bn(n):
    N = 2 ** n

    B = [[0 for i in range(N)] for j in range(N)]
    for i in range(N):
        B[i][i] = 1
        if i > 0:
            B[i-1][i] = 2
            B[i][i-1] = 2
    B[0][N - 1] = 2
    B[N- 1][0] = 2
    return np.array(B)


P = successor(n)
P_gate = P.to_instruction()
P_inverse = P_gate.inverse()


def IX(qc,q,N):
    qc.rx(-2*b*t/N, q[0])

def P(qc, q):
    qc.append(successor(n),q)

def trotterBn(N):
    q = QuantumRegister(n, name="q")
    qc = QuantumCircuit(q)
```

```
    qc.global_phase = a*t
    for i in range(N):

        IX(qc, q, N)

        qc.append(P_inverse, q[::-1])
        IX(qc, q, N)
        qc.append(P_gate, q[::-1])


    return qc


qc = trotterBn(699) # adjust to get to 10^-3

# approx = flipEndian(np.array(Operator.from_circuit(qc)))
approx = np.array(Operator.from_circuit(qc))
exact = linalg.expm(1j * t * Bn(n))
diff = linalg.interpolative.estimate_spectral_norm(exact - approx)
print("Error is", diff)
```

Listing 7: Implémentation du circuit de Trotter pour e^{itB_n}

Le code 7 implémente la Trotterisation de premier ordre. Le nombre d'itérations N requis pour obtenir une précision de 10^{-3} est :

- Pour un registre de taille $n = 3$ (8×8 matrices) : $\mathbf{N = 699}$ itérations.
- Pour un registre de taille $n = 5$ (32×32 matrices) : $\mathbf{N = 744}$ itérations.

On observe que le nombre d'itérations N est presque constant avec la taille n..

Pour la Trotterisation de second ordre, il suffit de changer la boucle d'itération de la manière suivante : 8.L'efficacité est grandement améliorée grâce à la convergence quadratique :

- Pour un registre de taille $n = 3$: $\mathbf{N = 17}$ itérations.
- Pour un registre de taille $n = 5$: $\mathbf{N = 17}$ itérations.

Le passage à l'ordre 2 a permis de réduire le coût de la simulation par un facteur $\approx \mathbf{41}$.

```
    for i in range(N):

        IX(qc, q, 2*N)

        qc.append(P_inverse, q[::-1])
        IX(qc, q, N)
        qc.append(P_gate, q[::-1])

        IX(qc, q, 2*N)
```

Listing 8: Boucle d'itération du pas de Trotter d'Ordre 2

2.2 2.3 - Computing circuit sizes

2.2.1 Question 1 et 2

Nous estimons la taille du circuit, $S(n)$,l'hypothèse simplifiée que le coût de décomposition des portes multi-contrôlées C^kX est unitaire..

Détermination du Nombre d'Étapes de Trotter (N_{Trotter}) Nous utilisons la formule de Trotter-Suzuki symétrique de second ordre, avec une erreur de $O(t^2/N^2)$. La précision requise est $\epsilon = 10^{-3}$, avec $t = 1/2$:

$$\epsilon \approx O\left(\frac{t^2}{N^2}\right) \leq 10^{-3} \tag{1}$$

$(O(t^2/N^2))$:

$$\frac{(t^2)}{N^2} \leq 10^{-3} \implies \frac{(1/2)^2}{N^2} \leq 10^{-3}$$

$$\frac{1}{4N^2} \leq \frac{1}{1000} \implies 4N^2 \geq 1000 \implies N \geq \sqrt{250} \approx 15.8$$

Nous choisissons le nombre de pas de Trotter par itération de QPE, N_{Trotter}, comme :

$$N_{\text{Trotter}} = \mathbf{16} \text{ itérations.}$$

Coût d'un Pas de Trotter (U_{step}) Nous utilisons la décomposition simplifiée : $U_{\text{step}} = U_{M_1} \cdot U_{M_2} \approx U_{M_1} \cdot (P \cdot U_{M_1} \cdot P^{-1})$.

- **Opérateur IX (U_{M_1})** : Coût = 1 (Porte Rx sur q_0).
- **Opérateur P (P^{-1})** : L'opérateur de succession nécessite n portes C^kX. Sous l'hypothèse simplifiée : Coût $\approx \mathbf{n-1}$ portes C^kX.

Le coût total d'un pas U_{step} (qui contient $2 \times U_{M_1}$ et $2 \times P$) est alors :

$$\begin{aligned}\text{Coût}(U_{\text{step}}) &= \text{Coût}(U_{M_1}) + \text{Coût}(P) + \text{Coût}(U_{M_1}) + \text{Coût}(P^{-1}) \\ &\approx 1 + n + 1 + n \\ &= \mathbf{2n + 2} \text{ portes}\end{aligned}$$

Taille Totale du Circuit (Complexité) En ne considérant que le coût des portes C^kX et R_x dans le pas de Trotter, la taille du circuit (par itération de QPE) est :

$$S(n) = N_{\text{Trotter}} \times \text{Coût}(U_{\text{step}})$$

$$S(n) \approx 16 \times (2n + 2)$$

$$\mathbf{S(n) \approx 32 \cdot n + 32}$$

La taille du circuit, sous l'hypothèse d'un coût unitaire pour chaque C^kX, est donc linéaire en n (le nombre de qubits du registre d'état).

2.2.2 Question 3

Dans *Coding Session #3: Trotterization*, nous avions vu que la décomposition naïve mise en place par Qiskit était de 4^n portes pour n registres. Ainsi, avec l'optimisation précédente, c'est à partir de 4 registres que notre approche devient plus efficace que la décomposition naïve de Qiskit (voir Fig. 1).

2.3 A Complete HHL Algorithm

Pour l'algorithme HHL complet, plusieurs blocs doivent être évalués en termes de portes et d'ancillas : les opérations contrôlées pour le QPE, l'inversion des valeurs propres, la Trotterization pour les puissances de l'unitaire, ainsi que les QFT direct et inverse.

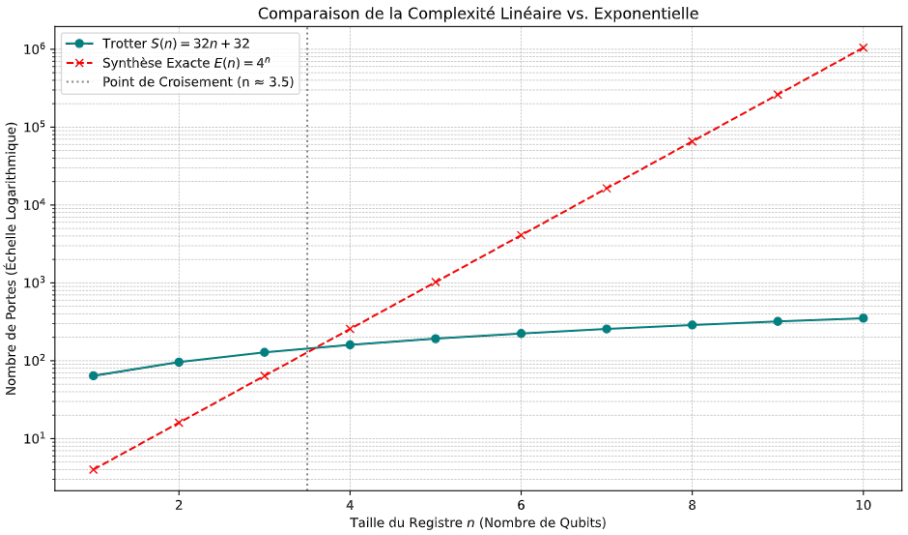


Figure 1: Comparaison du nombre de gates nécessaire entre la méthode naïve et la méthode optimisée en échelle log

Implémentation d'un opérateur C^kX. Un opérateur C^kX peut être construit en combinant les états de contrôle deux à deux à l'aide de portes Toffoli, puis en répétant ce procédé en arborescence sur de nouvelles ancillas. On utilise alors :

$$k-1 \text{ Toffoli pour la construction}, \qquad k-1 \text{ ancillas}, \qquad k-1 \text{ Toffoli pour désintriquer}.$$

Ainsi, au total :

$$2k-2 \text{ Toffoli}, \qquad k-1 \text{ ancillas}.$$

Implémentation de successeur(n). Pour réaliser l'opérateur successeur(n), on applique successivement

$$C^{n-1}X,\ C^{n-2}X,\ \ldots,\ C^1X.$$

Cela nécessite

$$(n-2)+(n-3)+\cdots+1=(n-1)^2 \text{ ancillas},$$

et donc

$$2(n-1)^2 \text{ Toffoli}.$$

Pour une précision de 10^{-3}, on utilise 16 étapes de Trotterization, d'où :

$$16(n-1)^2 \text{ ancillas}, \qquad 32(n-1)^2 \text{ Toffoli}.$$

Dans notre cas $n=7$, ce qui donne :

$$576 \text{ ancillas}, \qquad 1152 \text{ Toffoli}.$$

Implémentation de l'inverse. Pour atteindre une précision globale de 10^{-3}, on choisit 10 bits de précision puisque $2^{-10} \approx 10^{-3}$. Comme vu dans *Coding Session #4: Oracle Synthesis*, un degré 2 suffit avec $N=16$ et $n_{\text{frac}}=10$.
Avec les optimisations précédentes, le circuit nécessite environ :

$$15720 \text{ gates}, \qquad 18383 \text{ ancillas}.$$

Quantum Fourier Transform (QFT). Pour une QFT sur n qubits (ici $n = 10$) :

$$n \text{ Hadamard}, \qquad \frac{n(n-1)}{2} \text{ rotations contrôlées.}$$

Soit:

$$55 \text{ gates.}$$

Quantum Phase Estimation (QPE). Pour $n = 10$ qubits de précision :

- n portes Hadamard,
- une application de U,
- 10 Trotterizations successives (avec angle doublé à chaque étape dans la trotterizations pour ne pas avoir à augmenter la profondeur du circuit).

On obtient alors :

$$5760 \text{ ancillas}, \qquad 11520 \text{ Toffoli.}$$

Après cela, on applique la QFT (55 portes), puis l'ensemble est inversé. On double donc le nombre de portes (mais pas les ancillas, réutilisables) :

$$23205 \text{ gates}, \qquad 5760 \text{ ancillas.}$$

Synthèse du HHL complet. Le QPE et son inverse nécessitent :

$$46520 \text{ gates}, \qquad 11520 \text{ ancillas.}$$

L'inversion des valeurs propres requiert :

$$15720 \text{ gates}, \qquad 18383 \text{ ancillas.}$$

Total. On obtient donc :

$$\boxed{62240 \text{ gates}}, \qquad \boxed{29903 \text{ ancillas}}.$$

Appliquer l'algorithme HHL nécessite environ 70 000 portes et près de 30 000 ancillas. Nous sommes encore très loin de disposer d'ordinateurs quantiques capables de supporter une telle charge de ressources.