

High level programming language for quantum computing

Davide Camino



UNIVERSITÀ
DI TORINO

davide.camino@edu.unito.it

Obbiettivo del lavoro

Reasoning su quantum computer

- 1 Base di conoscenza classica + query
- 2 Embedding su quantum computer
- 3 Esecuzione e recupero risultati

Perché il quantum computing

Grazie a

- Entanglement
- Superposition

immagine

Violiamo

Strong Church-Turing Thesis

immagine

Due paradigmi di programmazione

Quantum Gate

immagine

Quantum Annealing

immagine

Quantum Gate

- Formalismo molto studiato
- Gate Reversibili
- Set di porte universali
- Algoritmi di Shore, ecc.

immagine

Quantum Annealer

immagine

- Problemi in forma QUBO
- Ispirato a Simulated Annealling
- Effetto tunneling
- Adiabatic Quantum Computing (AQC)
- $\mathcal{H}(t) = s(t)\mathcal{H}_i + (1 - s(t))\mathcal{H}_f$

QA-Prolog

Il progetto

- Sviluppato da Sott Pakin
- Proof of concept
- Trasformazioni successive
- $\text{Prolog} + \text{Query} \rightarrow \mathcal{H}_f$
- Interagisce direttamente con il solver
- Raccoglie e organizza i risultati

immagine

Pipeline

immagine

Trasformazioni

- 1 Prolog \rightarrow Verilog (HDL)
- 2 Verilog \rightarrow Circuito digitale
- 3 Circuito digitale $\rightarrow \mathcal{H}_f$ simbolica
- 4 \mathcal{H}_f simbolica $\rightarrow \mathcal{H}_f$ fisica

QMASM

\mathcal{H}_f simbolica $\rightarrow \mathcal{H}_f$ fisica

Cos'è

- Quantum macro assembler
- Sviluppato in Python
- Basso livello di astrazione
- Si interfaccia con Ocean

Cosa permette di fare

- Riferimento simbolico a *qubit*
- *Qubit* “pinnati” a TRUE o FALSE
- Incapsulare pattern in macro
- Creazione di librerie di macro
- Pulizia dell'output:
 - solo *qubit* “interessanti”
 - no slack variables

QMASM

Esempio: Macro

```
# Y = A OR B
!begin_macro OR
  $A 0.5
  $B 0.5
  $Y -1

  $A $B 0.5
  $A $Y -1
  $B $Y -1
!end_macro OR
```

Figure: or gate

```
# Y = NOT A
!begin_macro NOT
  $A $Y 1.0
!end_macro NOT
```

Figure: not gate

```
# Y = A AND B
!begin_macro AND
  $A -0.5
  $B -0.5
  $Y 1

  $A $B 0.5
  $A $Y -1
  $B $Y -1
!end_macro AND
```

Figure: and gate

possiamo racchiudere queste macro nel file gates.qasm

QMASM

Esempio: $y = x_1 \wedge \neg(x_2 \vee x_3)$

```
!include <gates>

!use_macro OR x2_or_x3
x2_or_x3.$A = x2
x2_or_x3.$B = x3
x2_or_x3.$Y = $x4

!use_macro NOT not_x4
not_x4.$A = $x4
not_x4.$Y = $x5

!use_macro AND x1_and_x5
x1_and_x5.$A = x1
x1_and_x5.$B = $x5
x1_and_x5.$Y = y
```

Figure: CircSat problem

“Pinniamo” il valore di y per ottenere l’assegnamento delle x_i che verificano la formula logica:

```
qmasm --run --pin="y := true" circsat.qmasm
```

Solution #1 (energy = -20.0000, tally = 647):

Variable	Value
-----	-----
x1	True
x2	False
x3	False
y	True

Figure: CircSat solution

Yosys - edif2qasm

Verilog \rightarrow Circuito digitale $\rightarrow \mathcal{H}$ simbolica

Yosys

- Framework per la sintesi del Verilog
- Free and open software sotto licenza ISC
- Output: RTL Netlist in formato EDIF

Immagine

edif2qasm

- Converte dal formato EDIF a QMASM
- Attinge a una libreria di gate

Immagine

Yosys - edif2qasm

Esempio: moltiplicazione tra interi

```
module mult (multiplicand, multiplier, product);  
    input [1:0] multiplicand;  
    input [1:0] multiplier;  
    output [2:0] product;  
  
    assign product = multiplicand * multiplier;  
endmodule
```

Figure: Factorization problem

Tradotto in EDIF con:

```
yosys myfile.v synth.ys -b edif -o myfile.edif
```

Tradotto in QMASM con:

```
edif2qasm -o="myfile.qasm" myfile.edif
```

Eseguito con:

```
qasm --run --pin="mult.product[2:0] := 110"  
--solver="sim_anneal" mult.qasm
```

Solution #1 (energy = -57.5000, tally = 68):

Variable	Value
mult.multiplicand[0]	False
mult.multiplicand[1]	True
mult.multiplier[0]	True
mult.multiplier[1]	True
mult.product[0]	False
mult.product[1]	True
mult.product[2]	True

Figure: Factorization Solution

QA-Prolog

- Traduzione da Prolog a Verilog
- Wrapper per tutta la Pipeline
- Risultati in formato Human Readable
- Decide dimensione delle variabili

immagine

QA-Prolog

Esempio

QAOA

Quantum approximate optimization algorithm

- 1 Definire matrici: \mathcal{H}_c e \mathcal{H}_m
- 2 Definire oracoli parametrici: $\mathcal{U}_c(\gamma) = e^{-i\gamma\mathcal{H}_c}$ e $\mathcal{U}_m(\beta) = e^{-i\beta\mathcal{H}_m}$
- 3 Applicazione ripetuta di $\mathcal{U}_c(\gamma)$ e $\mathcal{U}_m(\gamma)$
- 4 Ottimizzazione (classica) dei parametri γ_i e β_i

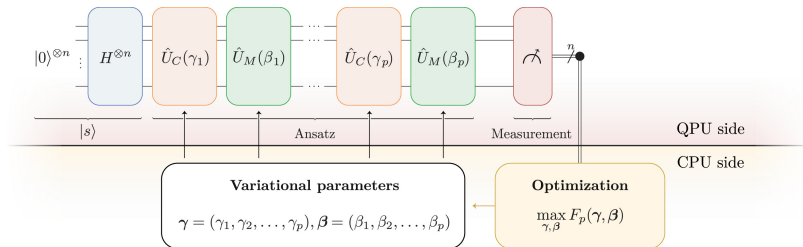


Figure: BLEKOS, Kostas, et al. A review on quantum approximate optimization algorithm and its variants

QUBO → ISING → Operatori di Pauli

```
qubo = qubovert.utils.matrix_to_qubo(qubo_mat)
ising = qubovert.utils.qubo_to_quso(qubo)
qubo_dict = dict(qubo)
ising_dict = dict(ising)
ising_dict.popitem()
ising_mat = qubovert.utils.qubo_to_matrix(ising_dict)
```

Figure: QUBO → ISING (libreria qubovert)

```
def build_paulis(matrix):
    pauli_list = []
    for i in range(len(matrix)):
        pauli_list.append(("Z", [i], matrix[i][i]))
        for j in range(i+1, len(matrix)):
            pauli_list.append(("ZZ", [i, j], matrix[i][j]))
    return pauli_list

sat_paulis = build_paulis(ising_mat)
cost_hamiltonian = SparsePauliOp.from_sparse_list(sat_paulis, n_qubits)
```

Figure: ISING → Operatori di Pauli

Esempio

Esempio

Risultato 1

Esempio

Risultato 2

Ontologie

OWL

Inferenze in OWL

Complessità dell'Inferenza

Da OWL-rdf a Prolog