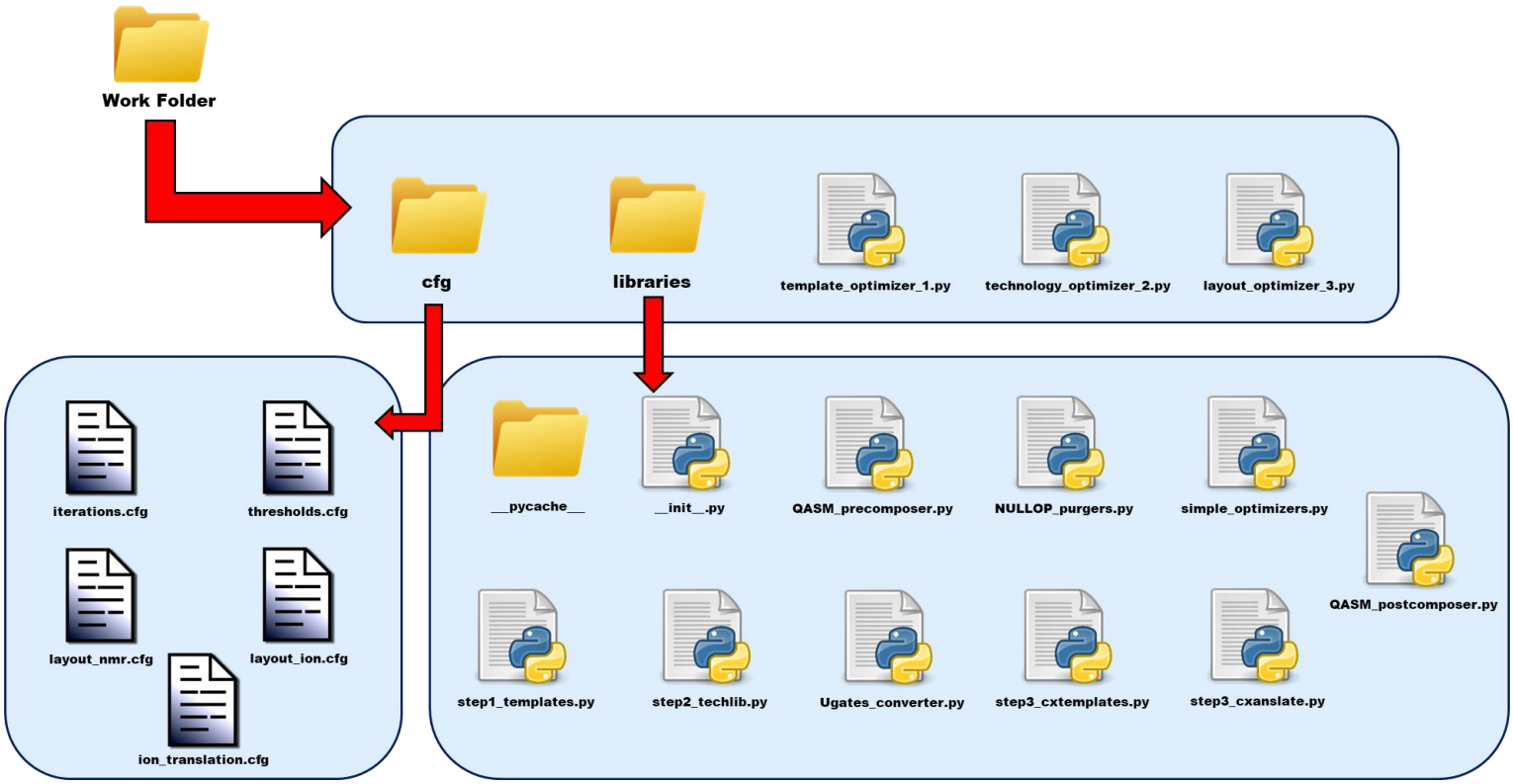
QUANTUM TOOLCHAIN: A COOKBOOK

Manfredi A.

INDICE CARTELLE DI LAVORO:

STRUTTURA DELLA TOOLCHAIN:



- ***Libraries:*** In questa cartella sono contenute tutte le librerie Python che vengono utilizzate dagli script principali della Toolchain per effettuare le ottimizzazioni. Per un'analisi in dettaglio delle varie funzioni delle librerie, rifarsi alla tesi "*Proposal for a multi-technology, template-based quantum circuits compilation Toolchain*" o alla documentazione a riguardo. La cartella **__pycache__** e il file **__init__.py** hanno il solo scopo di identificare la cartella *Libraries* come una da cui poter importare delle librerie in uno script Python, e non c'è bisogno di modificarle.

- ***CFG:*** In questa cartella sono contenuti i vari file .cfg utilizzati per definire alcuni parametri utilizzati dalla Toolchain nelle varie fasi dell'ottimizzazione da essa eseguita. Segue un elenco completo dei file e dei vari parametri da essi contenuti. Quando si editano i file è molto importante mantenere le indicazioni sulla forma del parametro specificate nei file stessi, evitando di utilizzare un format errato o di cambiare il nome dei parametri stessi (a meno

che, ovviamente, a questa seconda azione non corrisponda anche una apposita modifica negli script della Toolchain).

- **Thresholds.cfg:** Contiene il parametro **Threshold 1 (thr1)**, che definisce l'approssimazione decimale di pi greco e dei suoi sotto multipli utilizzata nelle varie funzioni (massima approssimazione supportata attualmente: **15** cifre decimali), e il parametro **Threshold 2 (thr2)**, che definisce il parametro rotazionale soglia sotto a cui una porta con suddetto parametro viene considerata come "nulla" (rotazione approssimabile a zero). In entrambi i casi, il parametro va scritto nella forma "**10^-x**".

- **Iterations.cfg:** Contiene i parametri di iterazione **Iter 1 (IT1)**, **Iter 2 (IT2)** e **Iter 3 (IT3)** utilizzati nello **Step 1 (IT1 e IT2)** e nello **Step 3 (IT3)** per definire il numero di iterazioni del loop interno. Per quanto non ci siano vincoli effettivi al valore che si può scrivere nel file .cfg, attualmente i vari script della Toolchain sono progettati per imporre un valore minimo di "**1"** per questi parametri.

- **Ion_Translation.cfg:** Contiene il parametro di traduzione **Iontranslate (iontran)** utilizzato nello **Step 2** e nello **Step 3** per definire se le porte **RX e RY** devono essere tradotte nella porta di rotazione generica **R** tipica della tecnologia. Può assumere un valore uguale a "**T**" (vero) o a "**F**" (falso).

- **Layout_Ion.cfg, Layout_NMR.cfg:** Entrambi i file sono analoghi, e contengono dei parametri utilizzati per gestire le porte a **2-qubit** nel caso (rispettivamente) della tecnologia a **ioni intrappolati** e la **NMR**. Entrambi contengono un parametro di **traduzione** (**CX Translation (ioncxtrans)** per gli **ioni**, **CZ Translation (cztranslat)** per la **NMR**) che può assumere un valore uguale a "**T**" (vero) o a "**F**" (falso) e che definisce se le porte a 2-qubit nello **Step 3** devono essere decomposte usando le porte base della tecnologia (**RXX o RZZ**). Contengono poi un parametro di **Layout** (**ion_layout** e **NMR_layout**) scritto in forma di una lista di liste che va a definire il **segno dell'interazione** di ciascun qubit con ogni altro singolo qubit. I valori ammessi sono "**1**" (**+**), "**-1**" (**-**) e "**0**" (segno dell'interazione di ogni qubit con sé stesso. In caso il

parametro venga scritto erroneamente e ci siano dei qubit che non hanno un'interazione nulla con sé stessi, lo **Step 3** è stato programmato in modo da controllare e riportare la cosa tramite un messaggio di errore).

- **Layouts.txt:** Non un file .cfg, questo file non è necessario per il funzionamento della Toolchain. All'interno di questo file di testo sono semplicemente riportati alcuni parametri di **Layout** utilizzati durante i vari test con la Toolchain, ed è utile come repository per evitare di riscrivere ogni volta una lista di liste potenzialmente anche molto lunga nei file .cfg.

- ***Cartella di lavoro principale (di default: optimizer):*** La cartella di lavoro è dove tutti i file .QASM da ottimizzare vanno spostati e dove tutti i file .QASM ottimizzati verranno prodotti. Essa contiene due serie di script: gli **script principali della Toolchain** e gli **script di debug**. Questi script, così come le funzioni, sono stati progettati per funzionare con **PYTHON 3.X.**

  - **Template_optimizer_1.py:** Lo script che implementa lo **Step 1 della Toolchain**. Accetta come parametri dalla shell **1) il nome del file .QASM da ottimizzare** e **2) il parametro Subcircuit** che definisce se il QC descritto è un sottocircuito, e può avere un valore uguale a "**T**"/"**true**" o "**F**"/"**false**". Se il parametro **Subcircuit** non viene specificato, viene automaticamente considerato come **falso**. Il file .QASM inserito deve essere innanzitutto un file **.QASM** (non sono ammessi altri tipi di file) e deve **trovarsi nella cartella di lavoro** (questa è un requisito per com'è attualmente implementata la Toolchain che può essere tranquillamente cambiato). Il file da ottimizzare deve avere come prime 4 righe "***OPENQASM 2.0; include "qelib1.inc"; qreg q[..]; creg c[..];"*** e deve contenere solo porte supportate dallo Step 1 (attualmente **RX, RY, RZ, X, Y, Z, T, Tdg, S, Sdg, H, CX, CZ, CCX.** Le porte IBM **U1, U2, U3** sono in realtà supportate, ma nessuna ottimizzazione viene eseguita su di esse in questo step). Il file di output avrà come **nome** il nome originale + un suffisso "**_optimized**".

- **Technology_optimizer_2.py:** Lo script che implementa lo **Step 2 della Toolchain**. Accetta come parametri dalla shell **1) il nome del file .QASM da ottimizzare** e **2) il parametro Subcircuit** che definisce se il QC descritto è un sottocircuito, e può avere un valore uguale a "**T**"/"**true**" o "**F**"/"**false**". Se il parametro **Subcircuit** non viene specificato, viene automaticamente considerato come **falso**. Il file .QASM inserito deve essere innanzitutto un file **.QASM** (non sono ammessi altri tipi di file) e deve **trovarsi nella cartella di lavoro** (questa è un requisito per com'è attualmente implementata la Toolchain che può essere tranquillamente cambiato). Idealmente, il file da ottimizzare dev'essere stato prodotto dallo **Step 1** (e avere dunque un suffisso "**_optimized**"), ma questo non è strettamente necessario. Il file da ottimizzare deve avere come prime 4 righe "***OPENQASM 2.0; include "qelib1.inc"; qreg q[..]; creg c[..];"*** e deve contenere solo porte supportate dallo Step 2 (attualmente **RX, RY, RZ, CX, CZ**). Ciascuna porta, inoltre, **non deve essere adiacente a una porta dello stesso tipo** (es. due porte RX adiacenti sulla stessa qubit line potrebbero portare a mancate ottimizzazioni). Questi ultimi due requisiti sono automaticamente soddisfatti se il file da ottimizzare è un output dello **Step 1**. Una volta lanciato lo script, verrà richiesto di inserire come input la **tecnologia quantistica da utilizzare**. Il file di output avrà come **nome** il nome originale + un suffisso "**_X_techoptimized**", in cui **X** è il parametro che indica la tecnologia utilizzata (**M** per **NMR**, **I** per **ioni intrappolati**, **S** per **superconduttive**). Nel caso in cui il file di input fosse un output dello Step 1, questo suffisso si va a sostituire al precedente suffisso "**_optimized**".

- **Layout_optimizer_3.py:** Lo script che implementa lo **Step 3 della Toolchain**. Accetta come parametri dalla shell **1) il nome del file .QASM da ottimizzare** e **2) il parametro Subcircuit** che definisce se il QC descritto è un sottocircuito, e può avere un valore uguale a "**T**"/"**true**" o "**F**"/"**false**". Se il parametro **Subcircuit** non viene specificato, viene automaticamente considerato come **falso**. Il file .QASM inserito deve essere innanzitutto un file **.QASM** (non sono ammessi altri tipi di file) e deve **trovarsi nella cartella di lavoro** (questa è un requisito per com'è attualmente implementata la Toolchain che può essere tranquillamente cambiato). Idealmente, il file da ottimizzare dev'essere stato prodotto dallo **Step 2** (e avere dunque un suffisso "**_X_techoptimized**"), ma questo non è strettamente necessario. Il file da ottimizzare deve avere

come prime 4 righe "***OPENQASM 2.0; include "qelib1.inc"; qreg q[..]; creg c[..];"*** e deve contenere solo porte supportate dallo Step 3 a seconda del tipo di tecnologia (attualmente **RX, RY, RZ, CX, CZ** per la **NMR**, **RX, RY, R, RZ, CX** per gli **ioni** e **U1, U2, U3, CX** per le **superconduttive**). Ciascuna porta, inoltre, **non deve essere adiacente a una porta dello stesso tipo** (es. due porte RX adiacenti sulla stessa qubit line potrebbero portare a mancate ottimizzazioni). Questi ultimi due requisiti sono automaticamente soddisfatti se il file da ottimizzare è un output dello **Step 2**. Una volta lanciato lo script, se il file di input è stato prodotto dallo **Step 2** il nome di questo verrà letto e il tipo di tecnologia da utilizzare nell'ottimizzazione verrà automaticamente identificato; in caso contrario, verrà richiesto di inserire come input la **tecnologia quantistica da utilizzare**. Il file di output avrà come **nome** il nome originale + un suffisso "**_X_finaloptimized**", in cui **X** è il parametro che indica la tecnologia utilizzata (**M** per **NMR**, **I** per **ioni intrappolati**, **S** per **superconduttive**). Nel caso in cui il file di input fosse un output dello Step 2, questo suffisso si va a sostituire al precedente suffisso "**_X_techoptimized**".

Per quanto riguarda gli script di **debug**, essi non sono altro che degli script "di comodo" per velocizzare le elaborazioni di più file permettendo delle ottimizzazioni "in massa".

- **Debugger.py:** Una volta lanciato, applica l'ottimizzazione dello **Step 1** a **tutti** i file **.QASM** attualmente nella cartella di lavoro.

- **Debugger2.py:** Una volta lanciato, applica l'ottimizzazione dello **Step 2** a **tutti** i file **.QASM che terminano con "_optimized.qasm"** (e che dunque sono stati prodotti dallo **Step 1**) attualmente nella cartella di lavoro.

- **Debugger3.py:** Una volta lanciato, applica l'ottimizzazione dello **Step 3** a **tutti** i file **.QASM che terminano con "_techoptimized.qasm"** (e che dunque sono stati prodotti dallo **Step 2**) attualmente nella cartella di lavoro.

GUIDA PRATICA ALL'UTILIZZO DELLA TOOLCHAIN:

- **Requisiti:** Come già specificato, i vari script e le librerie della Toolchain sono state progettate per funzionare con **PYTHON 3.X.** Per garantire il funzionamento della Toolchain è inoltre necessaria l'installazione delle librerie **python-math (default), Configparser, SciPy e NumPy.**

Per utilizzare i vari **script principali** descritti nella sezione precedente basta entrare nella **cartella di lavoro** e immettere nella shell il comando ***"python3 *nome script* *input script* ".***

EX: *Ottimizzazione con Step 1 di un file chiamato "example.qasm" che descrive un sottocircuito*

python3 template_optimizer_1.py example.qasm T

Per ottimizzare dunque un file .QASM tramite **tutti gli step della Toolchain** è dunque necessario usarlo come input per lo Step 1 e poi successivamente usare i file generati come input prima dello Step 2 e poi dello Step 3.

Nel caso si vogliano fare delle ottimizzazioni in massa tramite **gli script di debug**, basterà lanciare in sequenza i comandi "***python3 debugger.py***", "***python3 debugger2.py***", "***python3 debugger3.py***" all'interno della cartella di lavoro.

Tuttavia, durante **l'utilizzo dello script debugger.py** (il primo), è consigliabile spostare/eliminare della cartella di lavoro tutti i file .QASM che sono già stati ottimizzati dallo **Step 1** o che non devono essere ottimizzati da esso, in quanto si rischierebbe di fare delle elaborazioni non richieste (visto che lo script **debugger.py** usa automaticamente come input tutti i file .QASM presenti nella cartella).

(In caso di warning su *"Gimbal Locks detected"*, non allarmarsi: i tempi di computazione si allungheranno, ma la Toolchain è in grado di gestirli)

FILES DI BENCHMARK:

Nella cartella ***benchmark*** sono contenuti tre tipi di file: gli ***script Python di Benchmark***, gli ***script Bash per Benchmarks automatizzati*** e ***la repository di file da testare*****.**

- ***Repository di file da testare*:** Semplici cartelle ordinate a seconda dello **step di ottimizzazione** e della **tecnologia utilizzata** pensate per contenere i vari file .QASM da testare. Utilizzate dagli **script bash** come origine dei file da usare come input.

- ***Script Python di Benchmark*:** Sono tre distinti script usati per intavolare il benchmark vero e proprio, e simulano rispettivamente gli output di ogni **step della Toolchain** comparandoli ai risultati di **Qiskit** e **T-KET**. A livello di struttura complessiva, ognuno di essi è composto da una **prima parte di gestione degli input** seguita da una sezione in cui **i circuiti originali o prodotti dalla Toolchain** sono costruiti utilizzando **Qiskit**, in cui le loro varie **porte** vengono **contate** e in cui infine **vengono testati per verificare l'equivalenza logica tra circuiti**. Successivamente, si passa alla compilazione prima dei **circuiti ottimizzati da Qiskit** e poi dei **circuiti ottimizzati da T-KET**. Questi script sono stati progettati per funzionare con **PYTHON 3.X**, e si appoggiano estensivamente sulle librerie di **QISKIT** e **PYTKET**, che è dunque necessario installare**.**

    - **Benchmark.py:** Lo script necessario per testare i risultati dello **Step 1 della Toolchain**. Accetta come parametri dalla shell **1) il path del file .QASM di riferimento (target dell'ottimizzazione)**, **2) il path del file .QASM ottimizzato dallo Step 1**, **3) il parametro Unitary Simulator** che definisce se il test sul QC deve essere effettuato usando lo **Unitary Simulator** dell'**Aer backend di Qiskit** anziché il **QASM Simulator** standard, e può avere un valore uguale a "**T**"/"**true**" o "**F**"/"**false**", e **4) il parametro Latency Calculator** che definisce se durante la simulazione deve essere effettuato il **calcolo della latenza pesata** per le porte **a singolo qubit**, e può avere un valore uguale a "**T**"/"**true**" o "**F**"/"**false**".

Se i parametri **Unitary Simulator** e **Latency Calculator** non vengono specificati, vengono automaticamente considerati come **falsi**. I file .QASM inseriti devono essere innanzitutto file **.QASM** (non sono ammessi altri tipi di file). Lo script è pensato per testare QC prodotti dallo Step 1, ma di fatto **può essere utilizzato per testare qualsiasi coppia di QC descritti da file .QASM** (da qui **il generico nome "benchmark"**, a differenza degli altri due script). Nelle simulazioni, il parametro **π** è approssimato con una precisione di **10^-15**. Lo script si divide in più parti. Nella **prima parte**, i circuiti descritti **dai due file di input** vengono costruiti usando **Qiskit**. Questa costruzione, che avviene anche nelle fasi successive, si basa sul metodo **QuantumCircuit.from_qasm_file** di **Qiskit**. A esso viene in seguito accompagnato il metodo **count_ops()** che effettua un conteggio per tipo di **tutte le porte presenti nel circuito descritto**. L'output di questa prima parte è dunque una **lista contenente tutti i tipi di porta presenti nel circuito di riferimento e in quello ottimizzato dallo Step 1 con associato il relativo conteggio**, più un conteggio programmato ad-hoc delle **porte a singolo qubit**, **delle porte a singolo qubit di tipo non-RZ** e **delle porte CX/CZ**. Nel caso il parametro **Latency Calculator sia vero**, a partire dal conteggio effettuato viene anche valutata **la latenza pesata sulle porte a singolo qubit**. Questo processo si basa sull'identificare il parametro rotazionale di ogni singola porta nel circuito (eventualmente utilizzando l'approssimazione di **π**) e calcolare per ogni porta **la latenza** utilizzando la formula $\boldsymbol{Latenza} = \frac{\mathbf{2} * \boldsymbol{Parametro\ rotazionale}}{\boldsymbol{\pi}}$. Nel caso delle porte con parametro **π (X, Y)** questa latenza vale automaticamente **2**. Nel caso delle **porte H** la latenza è tarata per essere valutata nel "best case scenario" descritto nella tesi in cui viene utilizzata la scomposizione **S – RX(π/2) – S**, con relativa latenza uguale a **1**. Infine, le **porte di tipo RZ** sono considerate con **latenza nulla** in quanto **implementabili virtualmente**. Nella **seconda parte,** i due circuiti appena costruiti vengono **simulati**. Nel caso dello **Unitary Simulator**, si cerca di costruire la **matrice unitaria** relativa ai due circuiti e di plottarla sulla shell. Per farlo si generano due file temporanei identici a quelli di input ma "depurati" da eventuali **measure** (che non sono compatibili con questo tipo di simulazione), e da lì si calcolano le matrici. Successivamente, viene calcolata la **deviazione massima sulla diagonale tra il prodotto tra una matrice e la trasposta coniugata dell'altra** e **la matrice identità** (a cui, idealmente, tale prodotto

dovrebbe essere uguale). Questa deviazione è calcolata come **1 – min(x)**, in cui **x sono la norma due dei valori presenti sulla diagonale della matrice generata dal prodotto**. Nel caso questa deviazione sia **maggiore di 0.001**, viene generato un messaggio di errore per indicare che le matrici sono troppo differenti e che dunque **l'equivalenza logica tra circuito e circuito ottimizzato non è rispettata**. Questo simulatore è stato inizialmente utilizzato come metodo di test per circuiti che non prevedono un singolo risultato di output. **Tuttavia, si sconsiglia il suo utilizzo in quanto il calcolo delle matrici ha un carico computazionale molto intenso, e per circuiti medio-grandi Qiskit tende direttamente ad "arrendersi" e crashare quando si tenta di eseguirlo**. Non a caso, negli script di benchmark successivi il caso dello Unitary Simulator è stato del tutto **abbandonato**. Nel caso standard, quello del **QASM Simulator**, i circuiti costruiti vengono eseguiti dal backend di Qiskit e viene **plottato un istogramma relativo alle probabilità degli output da essi prodotti** utilizzando la libreria **matplotlib**. In questo caso, per validare l'equivalenza logica basta **osservare le differenze tra gli istogrammi** che, in un caso ideale, devono essere **nulle**. Nella **terza parte**, vengono costruiti e simulati i **circuiti ottimizzati da Qiskit**. Partendo dal **circuito di riferimento**, si va a definire una **base di porte da utilizzare nell'ottimizzazione (basis_gates)** e si fa un **transpiling ottimizzato** del circuito prima con **livello di ottimizzazione 1** e poi **3 (optimization_level)**. Una volta costruiti i circuiti, si effettua su di essi **il conteggio delle porte** e, eventualmente, **il calcolo della latenza pesata**. Va notato che **la base di porte utilizzate da Qiskit** può essere definita un'unica volta per entrambe le simulazione con diversi livelli di ottimizzazione, e che di default per testare lo **Step 1** è **RX, RY, RZ, CX**. Infine nella **quarta parte** vengono costruiti e simulati i **circuiti ottimizzati da T-KET**. In questa sezione bisogna innanzitutto definire **la struttura circuitale con cui verranno tradotte le porte CNOT**. In questo caso, verranno tradotte appunto utilizzando una **CX**. In secondo luogo, bisogna definire **la struttura circuitale con cui deve essere tradotta una generica tripletta RX(a) – RY(b) – RZ(c)** dal compilatore. In questo caso, le triplette verranno tradotte appunto in forma **RX – RY – RZ**, con l'eliminazione di una delle tre porte nel caso in cui **il suo parametro rotazionale sia nullo**. Fatto ciò, si va a costruire il circuito di riferimento con **Qiskit** e lo si interfaccia usando **pytket** con **T-KET**, lo si compila utilizzando **il backend Aer** e lo si ottimizza effettuando un **rebase**

utilizzando **la base di porte legali che si preferisce**. In questo caso la base è **identica a quella usata per Qiskit**. Il **rebase** viene effettuato utilizzando la funzione **RebaseCustom** (che richiede in ingresso alcuni parametri tra cui **le strutture circuitali** precedentemente descritte) e applicandolo poi al circuito ottimizzato. Anziché un **rebase personalizzato**, è possibile anche usare uno tra i vari **rebase automatici forniti da pytket** (più informazioni nella documentazione di **pytket.passes**). Il circuito viene compilato sia **con livello standard di** ottimizzazione sia **con livello massimo di ottimizzazione (compile_circuit(tket, optimisation_level = 2))**. Una volta costruiti i circuiti, si effettua su di essi **il conteggio delle porte** e, eventualmente, **il calcolo della latenza pesata**. Va notato che a differenza del caso di Qiskit, per **T-KET** sia il **rebase** che le varie **strutture circuitali** devono essere definiti nuovamente per ogni compilazione.

- **Benchmark_step2.py:** Lo script necessario per testare i risultati dello **Step 2 della Toolchain**. Accetta come parametri dalla shell **1) il path del file .QASM di riferimento (target dell'ottimizzazione)**, **2) il path del file .QASM ottimizzato dallo Step 1**, **3) il path del file .QASM ottimizzato dallo Step 2**, **4) il parametro che descrive la tecnologia usata nell'ottimizzazione ('M'** per la **NMR, 'I'** per gli **Ioni Intrappolati, 'S'** per le **Superconduttive)** e **5) il parametro Latency Calculator** che definisce se durante la simulazione deve essere effettuato il **calcolo della latenza pesata** per le porte **a singolo qubit**, e può avere un valore uguale a "**T**"/"**true**" o "**F**"/"**false**". Se il parametro **Latency Calculator** non viene specificato, viene automaticamente considerato come **falso**. I file .QASM inseriti devono essere innanzitutto file **.QASM** (non sono ammessi altri tipi di file). Nelle simulazioni, il parametro **π** è approssimato con una precisione di **10^-15**. Lo script si divide in più parti. Nella **prima parte**, i circuiti ottimizzati descritti **da due file di input** vengono costruiti usando **Qiskit**. Questa costruzione, che avviene anche nelle fasi successive, si basa sul metodo **QuantumCircuit.from_qasm_file** di **Qiskit**. A esso viene in seguito accompagnato il metodo **count_ops()** che effettua un conteggio per tipo di **tutte le porte presenti nel circuito descritto**. L'output di questa prima parte è dunque una **lista contenente tutti i tipi di porta presenti nel circuito ottimizzato dallo Step 1 e in quello ottimizzato dallo Step 2 con associato il relativo conteggio**, più un conteggio programmato ad-

hoc delle **porte a singolo qubit**, **delle porte a singolo qubit di tipo non-RZ** e **delle porte CX/CZ**. Nel caso il parametro **Latency Calculator sia vero**, a partire dal conteggio effettuato viene anche valutata **la latenza pesata sulle porte a singolo qubit**. Questo processo si basa sull'identificare il parametro rotazionale di ogni singola porta nel circuito (eventualmente utilizzando l'approssimazione di **π**) e calcolare per ogni porta **la latenza** utilizzando la formula $\boldsymbol{Latenza} = \frac{2 * \boldsymbol{Parametro\ rotazionale}}{\pi}$. Il benchmark, esattamente come lo **Step** 2, prevede che **nei circuiti** ottimizzati siano presenti **solo porte a singolo qubit di tipo RX, RY, RZ o U1, U2, U3**. Le **porte di tipo RZ** sono considerate con **latenza nulla** in quanto **implementabili virtualmente**. Nel caso delle **porte U**, le porte **U1** vengono considerate a **latenza** nulla (in quanto analoghe a delle **RZ**), le porte **U2** vengono considerate con **latenza = 1** e le porte **U3** vengono considerate con **latenza = 2**. Questa classificazione, per quanto meno esatta, permette di evidenziare come le porte **U3** abbiano una **latenza doppia** rispetto alle **U2**. Nella **seconda parte,** i due circuiti appena costruiti vengono **simulati utilizzando il QASM Simulator**. I circuiti costruiti vengono eseguiti dal backend di Qiskit e viene **plottato un istogramma relativo alle probabilità degli output da essi prodotti** utilizzando la libreria **matplotlib**. In questo caso, per validare l'equivalenza logica basta **osservare le differenze tra gli istogrammi** che, in un caso ideale, devono essere **nulle**. Nella **terza parte**, vengono costruiti e simulati i **circuiti ottimizzati da Qiskit**. Partendo dal **circuito di riferimento**, si va a definire una **base di porte da utilizzare nell'ottimizzazione (basis_gates) dipendente dalla tecnologia usata nell'ottimizzazione** e si fa un **transpiling ottimizzato** del circuito prima con **livello di ottimizzazione 1** e poi **3 (optimization_level)**. Una volta costruiti i circuiti, si effettua su di essi **il conteggio delle porte** e, eventualmente, **il calcolo della latenza pesata**. Va notato che **la base di porte utilizzate da Qiskit** può essere definita un'unica volta per entrambe le simulazione con diversi livelli di ottimizzazione, e che di default per testare lo **Step 2** è **RX, RY, RZ, CX, CZ** per la **NMR**, **RX, RY, RZ, CX** per gli **Ioni Intrappolati**, **U1, U2, U3, CX** per le **Superconduttive**. Infine nella **quarta parte** vengono costruiti e simulati i **circuiti ottimizzati da T-KET**. In questa sezione bisogna innanzitutto definire **la struttura circuitale con cui verranno tradotte le porte CNOT**. In questo caso, verranno tradotte appunto utilizzando una **CX**. In secondo luogo, bisogna definire **la struttura circuitale con cui**

deve essere tradotta una generica tripletta RX(a) – RY(b) – RZ(c) dal compilatore. In questo caso, le triplette verranno tradotte appunto in forma **RX – RY – RZ**, con l'eliminazione di una delle tre porte nel caso in cui **il suo parametro rotazionale sia nullo**. Fatto ciò, si va a costruire il circuito di riferimento con **Qiskit** e lo si interfaccia usando **pytket** con **T-KET**, lo si compila utilizzando **il backend Aer** e lo si ottimizza effettuando un **rebase** utilizzando **la base di porte legali che si preferisce**. In questo caso la base è **identica a quella usata per Qiskit**. Il **rebase** viene effettuato utilizzando la funzione **RebaseCustom** (che richiede in ingresso alcuni parametri tra cui **le strutture circuitali** precedentemente descritte) o, nel caso delle **Superconduttive**, utilizzando l'apposito **RebaseIBM** fornito da **pytket**, e applicandolo poi al circuito ottimizzato. Anziché un **rebase personalizzato**, è possibile anche usare uno tra i vari **rebase automatici forniti da pytket** (più informazioni nella documentazione di **pytket.passes**). Il circuito viene compilato sia **con livello standard di** ottimizzazione sia **con livello massimo di ottimizzazione (compile_circuit(tket, optimisation_level = 2))**. Una volta costruiti i circuiti, si effettua su di essi **il conteggio delle porte** e, eventualmente, **il calcolo della latenza pesata**. Va notato che a differenza del caso di Qiskit, per **T-KET** sia il **rebase** che le varie **strutture circuitali** devono essere definiti nuovamente per ogni compilazione.

- **Benchmark_step3.py:** Lo script necessario per testare i risultati dello **Step 3 della Toolchain**. Accetta come parametri dalla shell **1) il path del file .QASM di riferimento (target dell'ottimizzazione)**, **2) il path del file .QASM ottimizzato dallo Step 3**, **3) il parametro che descrive la tecnologia usata nell'ottimizzazione ('M'** per la **NMR, 'I'** per gli **Ioni Intrappolati, 'S'** per le **Superconduttive)** e **4) il parametro Latency Calculator** che definisce se durante la simulazione deve essere effettuato il **calcolo della latenza pesata** per le porte **a singolo qubit**, e può avere un valore uguale a "**T**"/"**true**" o "**F**"/"**false**". Se il parametro **Latency Calculator** non viene specificato, viene automaticamente considerato come **falso**. I file .QASM inseriti devono essere innanzitutto file **.QASM** (non sono ammessi altri tipi di file). Nelle simulazioni, il parametro **π** è approssimato con una precisione di **10^-15**. Lo script si divide in più parti. Nella **prima parte**, i circuiti ottimizzati descritti **dai due file di input** vengono costruiti usando **Qiskit**. Questa costruzione,

che avviene anche nelle fasi successive, si basa sul metodo **QuantumCircuit.from_qasm_file** di **Qiskit**. A esso viene in seguito accompagnato il metodo **count_ops()** che effettua un conteggio per tipo di **tutte le porte presenti nel circuito descritto**. L'output di questa prima parte è dunque una **lista contenente tutti i tipi di porta presenti nel circuito di riferimento e in quello finale ottimizzato dallo Step 3 con associato il relativo conteggio**, più un conteggio programmato ad-hoc delle **porte a singolo qubit**, **delle porte a singolo qubit di tipo non-RZ** e **delle porte CX/CZ**. Nel caso il parametro **Latency Calculator sia vero**, a partire dal conteggio effettuato viene anche valutata **la latenza pesata sulle porte a singolo qubit**. Questo processo si basa sull'identificare il parametro rotazionale di ogni singola porta nel circuito (eventualmente utilizzando l'approssimazione di **π**) e calcolare per ogni porta **la latenza** utilizzando la formula $\boldsymbol{Latenza} = \frac{\boldsymbol{2 * Parametro\ rotazionale}}{\boldsymbol{\pi}}$. Il benchmark, esattamente come lo **Step 3**, prevede che **nei circuiti** ottimizzati siano presenti **solo porte a singolo qubit di tipo RX, RY, RZ o U1, U2, U3**. Le **porte di tipo RZ** sono considerate con **latenza nulla** in quanto **implementabili virtualmente**. Nel caso delle **porte U**, le porte **U1** vengono considerate a **latenza** nulla (in quanto analoghe a delle **RZ**), le porte **U2** vengono considerate con **latenza = 1** e le porte **U3** vengono considerate con **latenza = 2**. Questa classificazione, per quanto meno esatta, permette di evidenziare come le porte **U3** abbiano una **latenza doppia** rispetto alle **U2**. Nella **seconda parte,** i due circuiti appena costruiti vengono **simulati utilizzando il QASM Simulator**. I circuiti costruiti vengono eseguiti dal backend di Qiskit e viene **plottato un istogramma relativo alle probabilità degli output da essi prodotti** utilizzando la libreria **matplotlib**. In questo caso, per validare l'equivalenza logica basta **osservare le differenze tra gli istogrammi** che, in un caso ideale, devono essere **nulle**. Nella **terza parte**, vengono costruiti e simulati i **circuiti ottimizzati da Qiskit**. Partendo dal **circuito di riferimento**, si va a definire una **base di porte da utilizzare nell'ottimizzazione (basis_gates) dipendente dalla tecnologia usata nell'ottimizzazione** e si fa un **transpiling ottimizzato** del circuito prima con **livello di ottimizzazione 1** e poi **3 (optimization_level)**. Una volta costruiti i circuiti, si effettua su di essi **il conteggio delle porte** e, eventualmente, **il calcolo della latenza pesata**. Va notato che **la base di porte utilizzate da Qiskit** può essere definita un'unica volta per entrambe le simulazione con diversi livelli di

ottimizzazione, e che di default per testare lo **Step 2** è **RX, RY, RZ, CZ** per la **NMR** (in quanto **il transpiling usando le RZZ non funziona in Qiskit)**, **RX, RY, RZ, RXX** per gli **Ioni Intrappolati**, **U1, U2, U3, CX** per le **Superconduttive**. Infine nella **quarta parte** vengono costruiti e simulati i **circuiti ottimizzati da T-KET**. In questa sezione bisogna innanzitutto definire **la struttura circuitale con cui verranno tradotte le porte CNOT**. In questo caso, verranno tradotte utilizzando **lo schema circuitale con le RZZ**, **lo schema circuitale con le RXX** o con semplici **CX** nel caso rispettivamente della **NMR**, degli **Ioni Intrappolati** e delle **Superconduttive**. In secondo luogo, bisogna definire **la struttura circuitale con cui deve essere tradotta una generica tripletta RX(a) – RY(b) – RZ(c)** dal compilatore. In questo caso, le triplette verranno tradotte appunto in forma **RX – RY – RZ**, con l'eliminazione di una delle tre porte nel caso in cui **il suo parametro rotazionale sia nullo**. Fatto ciò, si va a costruire il circuito di riferimento con **Qiskit** e lo si interfaccia usando **pytket** con **T-KET**, lo si compila utilizzando **il backend Aer** e lo si ottimizza effettuando un **rebase** utilizzando **la base di porte legali che si preferisce**. In questo caso la base è **identica a quella usata per Qiskit**, tranne nel caso della **NMR** in cui è effettivamente possibile usare le porte **RZZ** (chiamate **ZZPhase**) anziché le **CZ**. Il **rebase** viene effettuato utilizzando la funzione **RebaseCustom** (che richiede in ingresso alcuni parametri tra cui **le strutture circuitali** precedentemente descritte) o, nel caso delle **Superconduttive**, utilizzando l'apposito **RebaseIBM** fornito da **pytket**, e applicandolo poi al circuito ottimizzato. Anziché un **rebase personalizzato**, è possibile anche usare uno tra i vari **rebase automatici forniti da pytket** (più informazioni nella documentazione di **pytket.passes**). Il circuito viene compilato sia **con livello standard di** ottimizzazione sia **con livello massimo di ottimizzazione (compile_circuit(tket, optimisation_level = 2))**. Una volta costruiti i circuiti, si effettua su di essi **il conteggio delle porte** e, eventualmente, **il calcolo della latenza pesata**. Va notato che a differenza del caso di Qiskit, per **T-KET** sia il **rebase** che le varie **strutture circuitali** devono essere definiti nuovamente per ogni compilazione.

- ***Script Bash per Benchmarks automatizzati*:** Una serie di script che permettono di effettuare dei **benchmarks di massa** utilizzando **la modalità desiderata di test**. Essi si basano sul prendere la lista di **tutti i file** contenuti in

ogni cartella della **repository di file da testare** e di usarli come **input** per una chiamata reiterata di un certo **script Python di benchmark** con determinati **parametri tecnologici in ingresso**. In altre parole, basta sistemare i vari file .QASM di riferimento e i corrispettivi file ottimizzati nelle varie cartelle della **repository** e invocare questi script per effettuare in un colpo solo **più benchmarks**. Ovviamente, ciò diventa possibile solo se per **ogni file di riferimento** è presente nella cartella apposita un **corrispettivo file ottimizzato**, e viceversa.

- **Mass_qasmsim_benchmark.sh:** Utilizza i file contenuti nelle cartelle **Original_QCs (file di riferimento) e Optimized_QCs (file ottimizzati dallo Step 1)** per effettuare un benchmark di massa utilizzando lo script **Benchmark.py** in modalità **QASM Simulator** con **calcolo della latenza**.

- **Mass_unitary_benchmark.sh:** Utilizza i file contenuti nelle cartelle **Original_QCs (file di riferimento) e Optimized_QCs (file ottimizzati dallo Step 1)** per effettuare un benchmark di massa utilizzando lo script **Benchmark.py** in modalità **Unitary Simulator** con **calcolo della latenza**.

- **Mass_M_benchmark.sh:** Utilizza i file contenuti nelle cartelle **Original_QCs (file di riferimento)**, **Optimized_QCs (file ottimizzati dallo Step 1)** e **Techoptimized_M (file ottimizzati dallo Step 2 per NMR)** per effettuare un benchmark di massa utilizzando lo script **Benchmark_step2.py** per **NMR** con **calcolo della latenza**.

- **Mass_I_benchmark.sh:** Utilizza i file contenuti nelle cartelle **Original_QCs (file di riferimento)**, **Optimized_QCs (file ottimizzati dallo Step 1)** e **Techoptimized_I (file ottimizzati dallo Step 2 per Ioni Intrappolati)** per effettuare un benchmark di massa utilizzando lo script **Benchmark_step2.py** per **Ioni Intrappolati** con **calcolo della latenza**.

- **Mass_S_benchmark.sh:** Utilizza i file contenuti nelle cartelle **Original_QCs (file di riferimento)**, **Optimized_QCs (file ottimizzati dallo Step 1)** e **Techoptimized_S (file ottimizzati dallo Step 2 per Superconduttive)** per effettuare un benchmark di massa utilizzando lo script **Benchmark_step2.py** per **Superconduttive** con **calcolo della latenza**.

- **Mass_M_finalbenchmark.sh:** Utilizza i file contenuti nelle cartelle **Original_QCs (file di riferimento)** e **Finaloptimized_M (file ottimizzati dallo Step 3 per NMR)** per effettuare un benchmark di massa utilizzando lo script **Benchmark_step3.py** per **NMR** con **calcolo della latenza**.
- **Mass_I_finalbenchmark.sh:** Utilizza i file contenuti nelle cartelle **Original_QCs (file di riferimento)** e **Finaloptimized_I (file ottimizzati dallo Step 3 per Ioni Intrappolati)** per effettuare un benchmark di massa utilizzando lo script **Benchmark_step3.py** per **Ioni Intrappolati** con **calcolo della latenza**.

- **Mass_S_finalbenchmark.sh:** Utilizza i file contenuti nelle cartelle **Original_QCs (file di riferimento)** e **Finaloptimized_S (file ottimizzati dallo Step 3 per Superconduttive)** per effettuare un benchmark di massa utilizzando lo script **Benchmark_step3.py** per **Superconduttive** con **calcolo della latenza**.

GUIDA PRATICA ALL'UTILIZZO DEI BENCHMARK:

Requisiti: Come già specificato, i vari script di benchmark sono stati progettati per funzionare con **PYTHON 3.X.** Inoltre per poter effettuare i test comparati con **Qiskit** e **T-KET** è imprescindibile installare le librerie di **Qiskit** e di **pytket**. Si consiglia di installare queste librerie in un **apposito environment Conda** da attivare prima di effettuare i benchmarks. Per determinate versioni di **Qiskit**, si consiglia di installare una versione di **Python** nell'environment non superiore alla **3.7**, in quanto la **3.8** può in alcuni casi dare alcuni problemi.

Per utilizzare i vari **script principali** descritti nella sezione precedente basta entrare nella **cartella di lavoro** e immettere nella shell il comando ***"python3 *nome script* *input script* ".***

EX: *Benchmark di un circuito ottenuto con lo Step 2 della Toolchain utilizzando la tecnologia Superconduttiva ricavato partendo da un file di riferimento di nome "example.qasm", con calcolo della latenza abilitato (ipotizzando che i file si trovino nella disposizione di default della repository)*

python3 benchmark_step2.py ./Original_QCs/example.qasm ./techoptimized_S/example_S_techoptimized.qasm S T

In linea di massima, i benchmark "di interesse" sono quelli che riguardano i risultati finali prodotti dalla Toolchain, e dunque i file "_finaloptimized" testati usando lo script ***benchmark_step3.py***. Gli altri benchmark hanno (e hanno avuto) rilevanza solo in fase di sviluppo delle varie funzionalità della Toolchain.

Nel caso si vogliano fare delle ottimizzazioni in massa tramite **gli script Bash**, basterà lanciare lo script desiderato. Ovviamente, si consiglia di assicurarsi di aver disposto i file appositamente nelle varie cartelle della repository.

EX: *Benchmarks di massa di tutti i file ottimizzati dalla Toolchain usando la tecnologia a Ioni Intrappolati*

Bash mass_I_finalbenchmark.sh

QASMs:

La repository di file .QASM contiene tre cartelle: **github repository**, **mine** e **mine_randomized**.

La cartella **github repository** è l'archivio principale dei vari file .QASM usati nelle fasi di benchmarking per la Toolchain, ed è composta da svariati circuiti estratti dalle repository di **QASMBench** e dell'**Università di Linz**. Visto che alcuni circuiti nelle repository originali hanno più versioni, è stata salvata nell'archivio solo la versione "finale" (o comunque l'ultima al tempo del download).

La cartella **mine** contiene i vari file .QASM creati appositamente tramite IBM Quantum Experience per testare certe funzionalità della Toolchain e la corretta applicazione dei template. Essi non hanno nessuna rilevanza all'atto pratico, ma sono stati conservati per scrupolo.

La cartella **mine_randomized** contiene alcuni file .QASM che descrivono un circuito quantistico con una singola linea di qubit che contiene **10000 porte a singolo qubit** casuali, di tipo differente e adiacenti. Questi circuiti sono stati utilizzati per testare la "potenza" dei meccanismi **Eulercombo** e **Supercondmerge** nello **Step 2**. Sono anche presenti le forme di tali circuiti ottimizzate dallo **Step 1** e dallo **Step 2** per varie tecnologie. Anche questi circuiti non hanno rilevanza, ma se se ne volessero produrre degli altri si può utilizzare l'apposita funzione contenuta nelle **tool_functions** (descritte nella sezione successiva).

TOOL FUNCTIONS:

Queste tre funzioni sono degli "accessori" utilizzati in fase di sviluppo e conservati. Non svolgono nessuna funzione necessaria per le operazioni di ottimizzazione o per i benchmark.

- **Qiskitdrawer.py:** Una funzione che richiede come input il path di un file .QASM e che serve a disegnare il circuito quantistico da esso descritto. Al momento la funzione è impostata per scriverlo in forma testuale sulla shell (**output = 'text'**), ma può anche essere cambiata per disegnare una figura usando matplotlib. Ovviamente questa funzione utilizza i tool di **Qiskit**, ed è dunque necessario soddisfare i relativi requisiti elencati nella sezione relativa ai **Benchmarks.**

- **Zerocleaner.py:** Una funzione che richiede come input un file .QASM nella stessa cartella e che serve ad eliminare in esso tutte le porte RX, RY e RZ con un parametro rotazionale **nullo ('0.0')**. Questa funzione di pulizia è oramai automaticamente implementata nel meccanismo **Eulercombo**, che anche nel caso di **Gimbal Locks** non produce più queste porte ridondanti.

- **Randomstreak_generator.py:** Una funzione che genera tre file .QASM che descrivono altrettanti circuiti quantistici composti da una **singola linea di qubit** con **10000 porte a singolo qubit** casuali, di tipo differente e adiacenti. I file .QASM vengono salvati nella cartella dove si trova la funzione. Per cambiare i circuiti pseudo-casuali, occorre modificare arbitrariamente il parametro **seed** in funzione del parametro di iterazione **j**, e se si vogliono produrre più circuiti di questo tipo basta dare un range di valori diverso a **j** stesso nel ciclo *for* della funzione.