Architectures for Hybrid Quantum/Classical Computing

Will Zeng
ORNL 2017 05.25.2017
Outline

> Near-term quantum computers & What they can and can’t do

> An architectural outline for hybrid classical/quantum computing

  > The Quantum Instruction Language (Quil) for hybrid computing

  > Intro to Higher-level programming with pyQuil

> A worked example: QAOA for MAXCUT compiling from pyQuil down to the metal

> Example open problems for collaboration:

  > Routing, generic unitary compilation, high-performance noisy simulation, and classical integration
Near-term Quantum Computers

> Tens to low hundreds of physical qubits

- Nearest-neighbor lattices: superconducting qubits
- Finite fidelities
- Measurable cross-talk
- Typically capable of approximately parameterized gates
- Fast-feedback limitations
- Limited Error-correction
What we can’t do near-term

> Shor’s algorithm (of order $10^8$ qubits c.f. Fowler et al. 1208.0928)

> Anything with a qRAM

> Grover’s search

> Exact Hamiltonian Simulation

> Fault-tolerant quantum computation
What we can do: hybrid pre-threshold algorithms

- Variational Quantum Eigensolver
- Quantum Approximate Optimization Algorithm
What we can do: hybrid pre-threshold algorithms

> Variational Quantum Eigensolver

Peruzzo et al. 1304.3061

Kandala et al. 1704.05018

> Quantum Approximate Optimization Algorithm

O’Malley et al. 1512.06860
Variational Quantum Eigensolver

1. MOLECULAR DESCRIPTION
e.g. Electronic Structure Hamiltonian

2. MAP TO QUBIT REPRESENTATION
e.g. Bravyi-Kitaev or Jordan-Wigner Transform

3. PARAMETERIZED ANSATZ
e.g. Unitary Coupled Cluster
Variational Adiabatic Ansatz

\[
H = \sum_{i,j<i}^{N_n} |Z_i Z_j| + \sum_{i=1}^{N_n} -\frac{\nabla^2_{r_i}}{2} - \sum_{i,j}^{N_n,N_n} |Z_i Z_j| + \sum_{i,j<i}^{N_n} \frac{1}{|r_i - r_j|}.
\]

4. RUN Q.V.E. QUANTUM-CLASSICAL HYBRID ALGORITHM

\[
\frac{\langle \varphi(\theta) | H | \varphi(\theta) \rangle}{\langle \varphi(\theta) | \varphi(\theta) \rangle} \geq E_0
\]
Variational Quantum Eigensolver

Kandala et al. 1704.05018
Challenge in near-term quantum algorithms

Up to low hundreds of noisy physical qubits

Quantum Simulation
Variational Quantum Eigensolvers

Quantum Optimization
QAOA

QCVV & QEC
Topological Codes, GST, RB, CSS Codes, etc.

Catalysts

Complex Materials

\[ \text{Max-Cut cost operator } \hat{C}_4 = \frac{1}{2} \left( 1 - \sigma_1^z \sigma_2^z \right) \]

\( \text{Score } +1 \)

\( \text{Score } 0 \)

\( \text{N}_2\text{-activation, } \text{H}_2\text{O}_2\text{-etc} \)

\( \text{High-}\text{T}_c\text{, dichalcogenides, etc} \)

What are the gate counts / resource reqs?

How do they behave under noise?

How do we optimize them?

...and more!
Challenge in near-term quantum algorithms

Up to low hundreds of noisy physical qubits

**Quantum Simulation**
- Variational Quantum Eigensolvers $^{1,2,3,4}$
- Catalysts
- Complex Materials $^5$

**Quantum Optimization**
- QAOA $^6,7$
- $\text{MAX-CUT}$

Max-Cut cost operator $\tilde{\mathcal{C}}_U = \frac{1}{2}(I - \sigma_i^x \sigma_j^x)$

**QCVV & QEC**
- Topological Codes, GST, RB, CSS Codes, etc. $^{8,9,10,11}$

What are the gate counts / resource reqs?
How do they behave under noise?
How do we optimize them?

---

...and more!
How do we program near-term systems for near-term algorithms?
FOREST: A stack for classical/quantum hybrid programming

> Write applications...

> using tools...

> that build quantum programs...

> that compile onto quantum hardware...

> that execute on a real or virtual quantum processor.

APPLICATIONS: Grove

DEVELOPMENT: pyQuil

QUANTUM INSTRUCTION LANGUAGE: Quil

Allocation & Routing

Unitary Compilation

Scheduling

QUANTUM VIRTUAL MACHINE / QUANTUM COMPUTER

Open-sourced on github under Apache v2.0 license

github.com/rigetticomputing/pyquil
github.com/rigetticomputing/grove

Open for private-beta signups

forest.rigetti.com

forest.rigetti.com
Quil and the Quantum Abstract Machine

A hybrid classical/quantum programming model.
Quil is **portable**.
Quil is **portable, foundational**.
Quil is portable, foundational, hybrid.
Quil is **portable, foundational, hybrid.**
The Quil Programming Model

Targets a Quantum Abstract Machine (QAM)

- **Quil** is the instruction language and is how you interact with the machine
- It is a syntax for representing state transitions.
The Quil Programming Model

Targets a **Quantum Abstract Machine (QAM)**

- **Quil** is the instruction language and is how you interact with the machine
- It is a syntax for representing state transitions.

\[
\begin{align*}
\Psi &: \text{Quantum state (qubits)} \rightarrow \text{quantum instructions} \\
C &: \text{Classical state (bits)} \rightarrow \text{classical and measurement instructions} \\
\kappa &: \text{Execution state (program)} \rightarrow \text{control instructions (e.g., jumps)}
\end{align*}
\]

# Quil Example

```
H 3
MEASURE 3 [4]
JUMP-WHEN @END [5]
```

The Quil Programming Model

Targets a Quantum Abstract Machine (QAM)

- **Quil** is the instruction language and is how you interact with the machine
- It is a syntax for representing state transitions.

\[ \Psi: \text{Quantum state (qubits)} \rightarrow \text{quantum instructions} \]
\[ C: \text{Classical state (bits)} \rightarrow \text{classical and measurement instructions} \]
\[ \kappa: \text{Execution state (program)} \rightarrow \text{control instructions (e.g., jumps)} \]

0. Initialize into zero states

QAM: \( \Psi_0, C_0, \kappa_0 \)

1. Hadamard on qubit 3

\( \Psi_1, C_0, \kappa_1 \)

# Quil Example

```
H 3
MEASURE 3 [4]
JUMP-WHEN @END [5]
```

.
The Quil Programming Model

Targets a Quantum Abstract Machine (QAM)

> Quil is the instruction language and is how you interact with the machine
> It is a syntax for representing state transitions.

ψ: Quantum state (qubits) → quantum instructions

C: Classical state (bits) → classical and measurement instructions

κ: Execution state (program) → control instructions (e.g., jumps)

0. Initialize into zero states

QAM: ψ₀, C₀, κ₀

1. Hadamard on qubit 3

Outcome 0

ψ₁, C₀, κ₁

2. Measure qubit 3 into bit #4

Outcome 0

ψ₁, C₀, κ₁

Outcome 1

ψ₂, C₀, κ₁

# Quil Example

H 3

MEASURE 3 [4]

JUMP-WHEN @END [5]
The Quil Programming Model

Targets a **Quantum Abstract Machine (QAM)**

- **Quil** is the instruction language and is how you interact with the machine.
- It is a syntax for representing state transitions.

Ψ: Quantum state (qubits) → quantum instructions

C: Classical state (bits) → classical and measurement instructions

κ: Execution state (program) → control instructions (e.g., jumps)

---

# Quil Example

```
H 3  
MEASURE 3 [4]  
JUMP-WHEN @END [5]  
```

---

0. Initialize into zero states

QAM: $\Psi_0$, $C_0$, $\kappa_0$

1. Hadamard on qubit 3

2. Measure qubit 3 into bit #4

3. Jump to end of program if bit #5 is TRUE

---

Outcome 0

$\Psi_1$, $C_0$, $\kappa_1$

Outcome 1

$\Psi_3$, $C_1$, $\kappa_2$

---

$\Psi_2$, $C_0$, $\kappa_3$

---

...
Interacting with a Classical Computer

> The Quantum Abstract Machine has a shared classical state.
> The QAM becomes a practical device with this shared state.
> Classical computers can take over with classical/quantum synchronization.
A Practical Quantum Instruction Set Architecture

Robert S. Smith, Michael J. Curtis, William J. Zeng
Rigetti Computing
775 Heinz Ave.
Berkeley, California 94710
Email: {robert, spike, will}@rigetti.com

Abstract—Quantum computing technology has advanced rapidly in the last few years. Physical systems—superconducting qubits in particular—promise scalable gate-based hardware. Alongside these advances, new algorithms have been discovered that are adapted to the relatively smaller, noisier hardware that will become available in the next few years. These tend to be hybrid classical/quantum algorithms, where the quantum hardware is used in a co-processor model. Here, we introduce an abstract machine architecture for describing these algorithms, along with a language for representing computations on this machine, and discuss a classically simulable implementation architecture. Keywords—quantum computing, software architecture
Quantum Teleportation in Quil

DEFCIRCUIT TELEPORT A q B:
# Bell pair
H A
CNOT A B

# Teleport
CNOT q A
H q
MEASURE q [0]
MEASURE A [1]

# Classically communicate measurements
JUMP-UNLESS @SKIP [1]
X B
LABEL @SKIP
JUMP-UNLESS @END [0]
Z B
LABEL @END

# If Alice’s qubits are 0 and 1
# and Bob’s is 5
TELEPORT 0 1 5
Higher level programming with pyQuil
A Python library for hybrid programming
MAXCUT on a near-term quantum computer

Compiling the QAOA hybrid algorithm down to the metal
FOREST: Tools for experimental quantum programming


Quantum Approximate Optimization Algorithm (QAOA)

Overview

pyQAOA is a Python module for running the Quantum Approximate Optimization Algorithm on an instance of a quantum abstract machine.

The pyQAOA package contains separate modules for each type of problem instance: MAX-CUT, graph partitioning, etc. For each problem instance the user specifies the driver Hamiltonian, cost Hamiltonian, and the approximation order of the algorithm.

$qaoa.py$ contains the base QAOA class and routines for finding optimal rotation angles via Grove's variational-quantum-eigensolver method.
The Quantum Approximate Optimization Algorithm

QAOA: kwaah-waah

> A Hybrid-Quantum Classical Algorithm: Farhi et al. 2014 [1411.4028]
  - Quantum co-processor algorithm (like VQE)
  - Noise tolerant

> Can demonstrate quantum supremacy: Farhi & Harrow 2016 [1602.07674]

> Similar to Digitized Quantum Annealing: Barends et al. 2015 [1511.03316]
The Quantum Approximate Optimization Algorithm

QAOA: kwaah-waah

> A Hybrid-Quantum Classical Algorithm: Farhi et al. 2014 [1411.4028]
  - Quantum co-processor algorithm (like VQE)
  - Noise tolerant

> Can demonstrate quantum supremacy: Farhi & Harrow 2016 [1602.07674]

> Similar to Digitized Quantum Annealing: Barends et al. 2015 [1511.03316]

THE PROBLEM

Constraint Satisfaction Problems:

\[ z \in \{0,1\}^n \]

\[ C_a(z) = \begin{cases} 
1 & \text{if } z \text{ satisfies the constraint } a \\
0 & \text{if } z \text{ does not}.
\end{cases} \]

\[ C'(z) = \sum_{a=1}^{m} C_a(z) \]
The Quantum Approximate Optimization Algorithm

QAOA: kwaah-waah

Constraint Satisfaction Problems:

\[ z \in \{0, 1\}^n. \]

\[ C_a(z) = \begin{cases} 1 & \text{if } z \text{ satisfies the constraint } a \\ 0 & \text{if } z \text{ does not.} \end{cases} \]

\[ C(z) = \sum_{a=1}^{m} C_a(z) \]

MAXIMIZE
The Quantum Approximate Optimization Algorithm

QAOA: kwaah-waah

> Constraint Satisfaction Problems:

\[z \in \{0, 1\}^n,\]

\[C_a(z) = \begin{cases} 1 & \text{if } z \text{ satisfies the constraint } a \\ 0 & \text{if } z \text{ does not} \end{cases}\]

\[C(z) = \sum_{a=1}^{m} C_a(z)\]

\[\hat{H}_s = (1 - s)\hat{H}_{\text{Driver}} + s\hat{H}_{\text{Cost}}\]

\[e^{-i\hat{H}_s dt} = e^{-i(1-s)\hat{H}_X dt} e^{-is\hat{H}_Z dt}\]

\[(1-s)dt = \beta_s\]

\[(s)dt = \gamma_s\]

\[U(\hat{H}_{\text{Driver}}, \beta_s) U(\hat{H}_{\text{Cost}}, \gamma_s)\]
The Quantum Approximate Optimization Algorithm

QAOA: kwaah-waah

> Constraint Satisfaction Problems:

\[ z \in \{0, 1\}^n. \]

\[ C_a(z) = \begin{cases} 1 & \text{if } z \text{ satisfies the constraint } a \\ 0 & \text{if } z \text{ does not} \end{cases} \]

\[ C(z) = \sum_{a=1}^{m} C_a(z) \]

\[ H_s = (1-s)\hat{H}_{\text{Driver}} + s\hat{H}_{\text{Cost}} \]

\[ e^{-i\hat{H}_s dt} = e^{-i(1-s)H_X dt} e^{-isH_Z dt} \]

\[ (1-s)dt = \beta_s \]

\[ (s)dt = \gamma_s \]

\[ U(\hat{H}_{\text{Driver}}, \beta_s) U(\hat{H}_{\text{Cost}}, \gamma_s) \]

\[ |0\rangle^\otimes n \]

s = 0

\[ U(\hat{H}_{\text{Cost}}, \gamma_s) \]

s = 1

\[ U(\hat{H}_{\text{Driver}}, \beta_s) \]

\[ \ldots \]

s = (p - 1)

\[ U(\hat{H}_{\text{Driver}}, \beta_s) \]

\[ |\psi_F(\vec{\beta}, \vec{\gamma})\rangle \]
The Quantum Approximate Optimization Algorithm

> Constraint Satisfaction Problems:
\[ z \in \{0, 1\}^n. \]
\[ C_a(z) = \begin{cases} 1 & \text{if } z \text{ satisfies the constraint } a \\ 0 & \text{if } z \text{ does not} \end{cases} \]
\[ C(z) = \sum_{a=1}^{m} C_a(z) \]

\[ \hat{H}_s = (1-s)\hat{H}_{\text{Driver}} + s\hat{H}_{\text{Cost}} \]
\[ e^{-i\hat{H}_s dt} = e^{-i(1-s)H_X dt} e^{-isH_Z dt} \]
\[ (1-s)dt = \beta_s \]
\[ (s)dt = \gamma_s \]
\[ U(\hat{H}_{\text{Driver}}, \beta_s) U(\hat{H}_{\text{Cost}}, \gamma_s) \]

\[ \langle \psi_F(\bar{\beta}, \bar{\gamma}) | C | \psi_F(\bar{\beta}, \bar{\gamma}) \rangle \]

\[ s = 0 \]
\[ s = 1 \]
\[ \ldots \]
\[ s = (p-1) \]

\[ |0\rangle^\otimes n \]
The Quantum Approximate Optimization Algorithm

\[ \text{QAOA: kwaah-waah} \]

Constraint Satisfaction Problems:
\[ z \in \{0, 1\}^n. \]
\[ C_a(z) = \begin{cases} 1 & \text{if } z \text{ satisfies the constraint } a \\ 0 & \text{if } z \text{ does not} \end{cases} \]
\[ C(z) = \sum_{a=1}^{m} C_a(z) \]

\[ \hat{H}_s = (1 - s)\hat{H}_\text{Driver} + s\hat{H}_\text{Cost} \]
\[ e^{-i\hat{H}_s dt} = e^{-i(1-s)H_X dt} e^{-isH_Z dt} \]
\[ (1-s)dt = \beta_s \quad (s)dt = \gamma_s \]
\[ U(\hat{H}_\text{Driver}, \beta_s) \quad U(\hat{H}_\text{Cost}, \gamma_s) \]

\[ \langle \psi_F(\vec{\beta}, \vec{\gamma}) | C | \psi_F(\vec{\beta}, \vec{\gamma}) \rangle \]

\[ |0\rangle^n \rightarrow U(\vec{\beta}, \vec{\gamma}) |0\rangle^n \]
Running QAOA

Determine direction for minimization

CPU

Update Beta, Gamma

$|\psi_F(\vec{\beta}, \vec{\gamma})\rangle$

Noisy Probability Density Minimization

Objective Function

Probability vs. Optimization param
Running QAOA

Determine direction for minimization

CPU
Update Beta, Gamma

Alternative Approach: analytically calculate optimal coefficients and run once.
WIP by Rieffel & NASA QuAIL
QAOA for MAX-CUT

> Define constraints with an arbitrary graph

\[ C_{ij} = \frac{1}{2} (1 - Z_i Z_j) \]

where \( Z_i \in \{+1, -1\} \)

> Hamiltonian Cost Function:

\[ \hat{C}_{ij} = \frac{1}{2} \left( \mathbf{I} - \sigma_i^z \sigma_j^z \right) \]

> Ring Example:

Score = 0  \quad \rightarrow \quad \text{Score} = +4
QAOA for MAX-CUT

> Define constraints with an arbitrary graph

\[ C_{ij} = \frac{1}{2} (1 - Z_i Z_j) \]
\[ Z_i \in \{+1, -1\} \]

> Hamiltonian Cost Function:

\[ \hat{C}_{ij} = \frac{1}{2} (I - \sigma_i^Z \sigma_j^Z) \]

> Ring Example:

```
import itertools
from pyquil.quil import Program
from pyquil.paulis import sZ, sX, sI, exponential_map
from pyquil.compiler import rpqc

graph = [(0, 1), (1, 2), (2, 3), (3, 4)]
nodes = {node for edge in graph for node in edge}

cost_ham = sum(0.5 * sZ(i) * sZ(j) - 0.5 * sI(0) for i, j in graph)
driver_ham = sum(-1.0 * sX(i) for i in nodes)
```
QAOA for MAX-CUT

> Define constraints with an arbitrary graph

\[ C_{ij} = \frac{1}{2} (1 - Z_i Z_j) \]

\[ Z_i \in \{+1, -1\} \]

> Hamiltonian Cost Function:

\[ \hat{C}_{ij} = \frac{1}{2} (I - \sigma_i^Z \sigma_j^Z) \]

> Ring Example:

\[
\begin{array}{c}
\text{Score} = 0 \\
\text{Score} = +4
\end{array}
\]

```python
import itertools
from pyquil.quil import Program
from pyquil.paulis import sZ, sX, sI, exponential_map
from pyquil.compiler import rpqc

graph = [(0, 1), (1, 2), (2, 3), (3, 4)]
nodes = {node for edge in graph for node in edge}

cost_ham = sum(0.5 * sZ(i) * sZ(j) - 0.5 * sI(0) for i, j in graph)
driver_ham = sum(-1. * sX(i) for i in nodes)

def cost_step(gamma):
    return merge_program([exponential_map(term)(gamma) for term in cost_ham])

def driver_step(beta):
    return merge_program([exponential_map(term)(beta) for term in driver_ham])
```
QAOA for MAX-CUT

> Define constraints with an arbitrary graph

\[
    C_{ij} = \frac{1}{2} (1 - Z_i Z_j)
\]

> Hamiltonian Cost Function:

\[
    \hat{C}_{ij} = \frac{1}{2} (I - \sigma^Z_i \sigma^Z_j)
\]

> Ring Example:

```
import itertools
from pyquil.paulis import sZ, sX, sI, exponential_map
from pyquil.compiler import rpqc

graph = [(0, 1), (1, 2), (2, 3), (3, 4)]

nodes = {node for edge in graph for node in edge}

cost_ham = sum(0.5 * sZ(i) * sZ(j) - 0.5 * sI(0) for i, j in graph)
driver_ham = sum(-1. * sX(i) for i in nodes)

def cost_step(gamma):
    return merge_program([exponential_map(term)(gamma) for term in cost_ham])

def driver_step(beta):
    return merge_program([exponential_map(term)(beta) for term in driver_ham])

def qaoa_circuit_maker(gammas, betas):
    cost_steps = map(cost_step, gammas)
driver_steps = map(driver_step, betas)

    interleaved_steps = zip(cost_steps, driver_steps)

    qaoa_circuit = merge_program([step for step_pair in interleaved_steps for step in step_pair])

    return rpqc(qaoa_circuit)

gammas = [2.0, 1.0]
betas = [0.0, 3.0]

qaoa_circuit_maker(gammas, betas)

# then execute the circuit and optimize over gammas and betas
```
Compilation

DEVELOPMENT: pyQuil

QUANTUM INSTRUCTION LANGUAGE: Quil

Allocation & Routing

Unitary Compilation

Scheduling

QUANTUM VIRTUAL MACHINE / QUANTUM COMPUTER

> Quil qubit labels to physical qubits
> Routing to deal with two-qubit gates
> Optimization over noise and errors

> Compilation into the natural gate set
> Rotation Decomposition

> Scheduling into microcode
### Why do we need to schedule?

- Quil has **no** notion of time or synchronization.
- But time and synchronization are very important.
- What are our options?

<table>
<thead>
<tr>
<th>Option</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Give up: Admit the physicists are better</td>
<td>“Program” with buttons and wires.</td>
<td>Difficult to reason about</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nixes the idea of an abstraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Difficult to automate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Have to think about hardware</td>
</tr>
<tr>
<td>Include ad hoc synchronization instructions</td>
<td>Extend Quil to “know” about time.</td>
<td>Extremely complicated!</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Difficult to reason about</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not easily extensible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hard to implement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Loses the “essence”</td>
</tr>
<tr>
<td>Compile Quil into some temporal representation</td>
<td>Add machine-specific directives.</td>
<td>Compilation is more difficult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Performance characterization is machine-specific</td>
</tr>
</tbody>
</table>

**Pros:**
- Maximal control
- Remains abstract
- Adds control as necessary
- Extensible!
- Keeps Quil “clean”
- Remains abstract
- Adds control as necessary
- Extensible!
- Keeps Quil “clean”

**Cons:**
- Difficult to reason about
- Nixes the idea of an abstraction
- Difficult to automate
- Have to think about hardware
- Extremely complicated!
- Not easily extensible
- Hard to implement
- Loses the “essence”
Events
(target, name, start_time, duration, param_dict)

Schedules
A set of events (and some transformations on them)

Compilation to Schedule

QPU Microcode is given by supported event types, e.g.

(target, “X-HALF”, start_time, 40.e-9, {“z_shift”: theta})
(target, “+F”, start_time, 250.e-9, { })
(target, “-F”, start_time, 250.e-9, { })
(target, “READOUT”, start_time, 1.e-6, { })
## Open Problems:

<table>
<thead>
<tr>
<th>Allocation &amp; Routing</th>
<th>Generic Unitary Decomposition</th>
<th>High performance simulation</th>
<th>Integration with Classical HPC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Optimal implementation includes optimization over:</strong></td>
<td><strong>Single-qubit case is well understood $O(\log(1/e))$</strong> (<a href="https://arxiv.org/abs/1510.03888">Kliuchnikov et al. 1510.03888</a>)</td>
<td><strong>qHIPSTER. Smelyanski et al. 1601.07195.</strong></td>
<td><strong>Post-processing to reduce impact of sampling error in VQE &amp; QAOA</strong></td>
</tr>
<tr>
<td>&gt; Gate sets that vary across the chip</td>
<td><strong>Martinez et al. Compiling quantum algorithms for architectures with multi-qubit gates. 1601.06819</strong></td>
<td><strong>High Performance Emulation of Quantum Circuits. Haener et al. 1604.06460</strong></td>
<td><strong>Computationally intensive decoders in QEC</strong></td>
</tr>
<tr>
<td>&gt; Noise in gates</td>
<td><strong>Maslov. Basic circuit compilation techniques for an ion-trap quantum machine. 1603.07678</strong></td>
<td><strong>0.5 Petabyte Simulation of a 45-Qubit Quantum Circuit. Haener &amp; Steiger. 1704.01127.</strong></td>
<td><strong>Integrations of quantum co-processors in larger workflows, e.g. DMET w/ VQE. Rubin 1610.06910</strong></td>
</tr>
<tr>
<td>&gt; Noise in qubits</td>
<td><strong>ScaffCC <a href="https://arxiv.org/abs/1507.01902">1507.01902</a></strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Acknowledgements

Rigetti Quantum Programming Team

Robert Smith  Nick Rubin

Our hardware & control software colleagues, esp. Spike Curtis, Matt Reagor and Rodney Sinclair