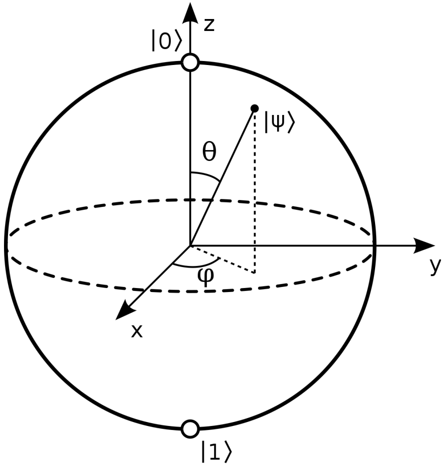
Implementing NChooseK on IBM Q Quantum Computer Systems

Harsh Khetawat, Ashlesha Atrey, George Li,
Frank Mueller, Scott Pakin

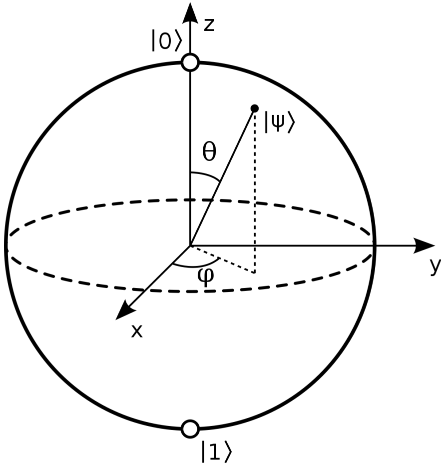
Reversible Computing 2019
05/02/2019

Quantum Computing Fundamentals



Basic unit of computation – qubit

Quantum Computing Fundamentals

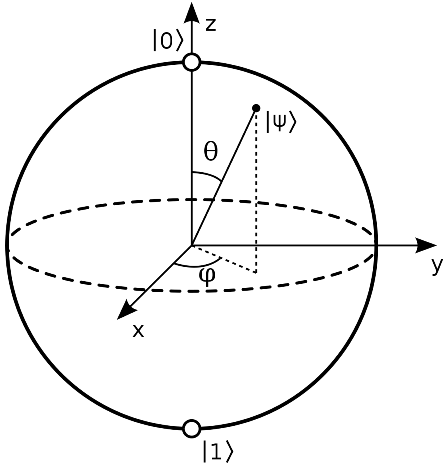


Basic unit of computation – qubit

Exists in superposition –

ex. $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$ where α & β are probability amplitudes – complex numbers

Quantum Computing Fundamentals



Basic unit of computation – qubit

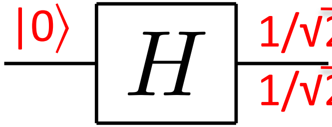
Exists in superposition –

ex. $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$ where α & β are probability amplitudes – complex numbers

Probability of measuring $|0\rangle$ and $|1\rangle$ are α^2 and β^2 respectively.

Basic Quantum Gates

Basic Quantum Gates

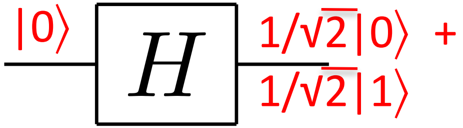


$|0\rangle \to 1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle$

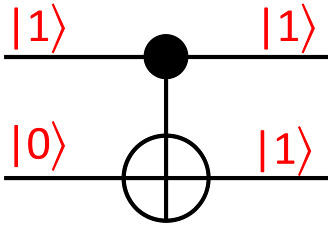
Hadamard Gate – Used to create superposition

Equal probabilities of measuring $|0\rangle$ or $|1\rangle$

Basic Quantum Gates

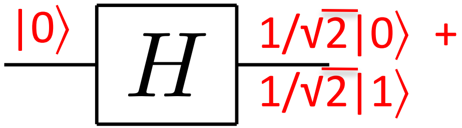


Hadamard Gate – Used to create superposition
Equal probabilities of measuring |0⟩ or |1⟩

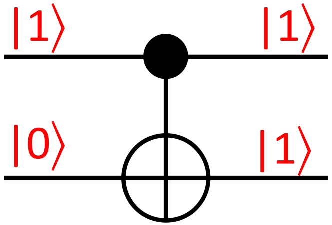


Controlled NOT Gate – 2 qubit gate
Flips target (2nd) qubit if control (1st) qubit is 1

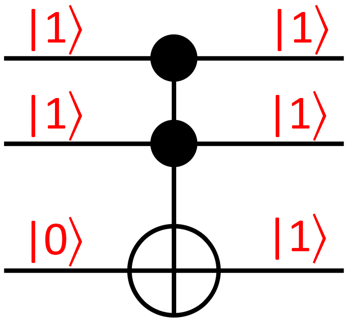
Basic Quantum Gates



Hadamard Gate – Used to create superposition
Equal probabilities of measuring $|0\rangle$ or $|1\rangle$



Controlled NOT Gate – 2 qubit gate
Flips target (2^{nd}) qubit if control (1^{st}) qubit is 1



Toffoli Gate – 3 qubit gate
Flips target (3^{rd}) qubit if both control (1^{st} & 2^{nd}) qubits is 1
Along with the Hadamard gate, Toffoli gate is universal

Grover's Algorithm

Grover's Algorithm

Problem: For a black box function, find unique input for a particular output where size of function domain is N

Grover’s Algorithm

Problem: For a black box function, find unique input for a particular output where size of function domain is N

Classical Solution: Requires N iterations in the worst case

Grover's Algorithm

Problem: For a black box function, find unique input for a particular output where size of function domain is N

Classical Solution: Requires N iterations in the worst case

Quantum Solution: Requires $\sqrt{N}$ iterations

Grover's Algorithm

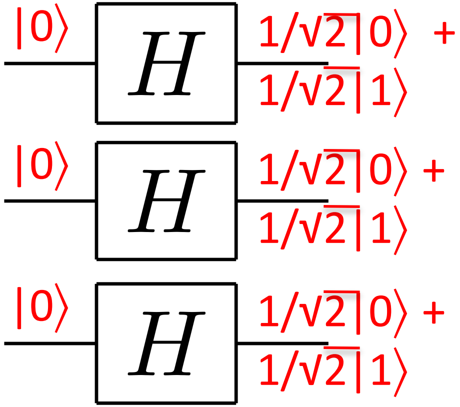
Problem: For a black box function, find unique input for a particular output where size of function domain is N

Classical Solution: Requires N iterations in the worst case

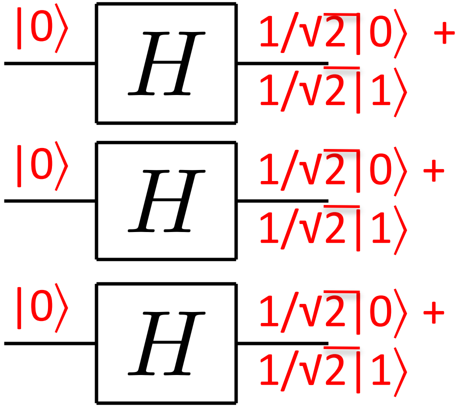
Quantum Solution: Requires $\sqrt{N}$ iterations

Applications: DB search, breaking cryptography

Quantum Advantage

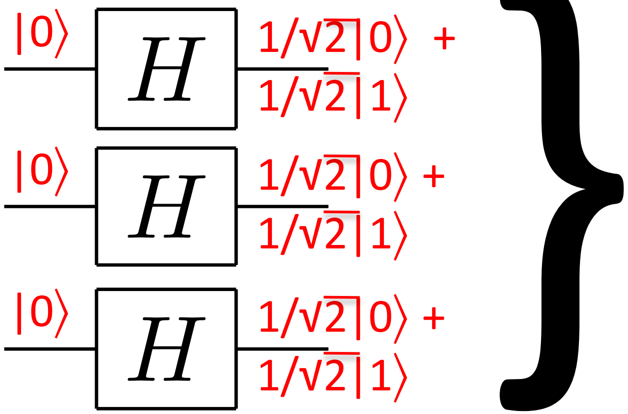


Quantum Advantage

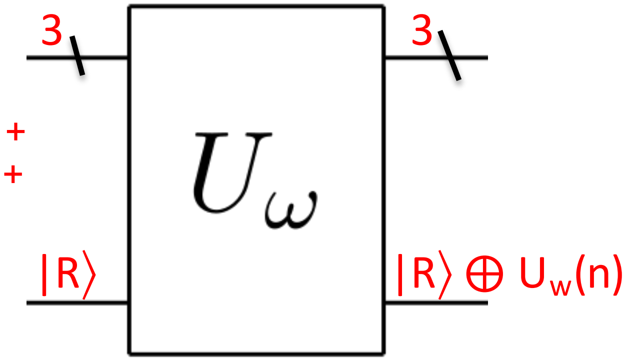


$1/\sqrt{8}|000\rangle + 1/\sqrt{8}|001\rangle + 1/\sqrt{8}|010\rangle +$
$1/\sqrt{8}|011\rangle + 1/\sqrt{8}|100\rangle + 1/\sqrt{8}|101\rangle +$
$1/\sqrt{8}|110\rangle + 1/\sqrt{8}|111\rangle$

Quantum Advantage



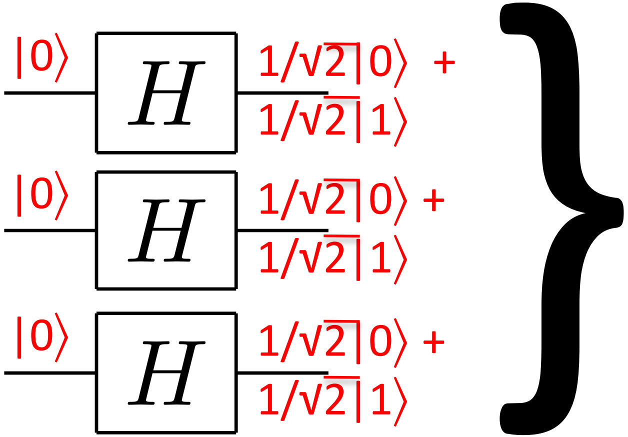
$1/\sqrt{8}|000\rangle + 1/\sqrt{8}|001\rangle + 1/\sqrt{8}|010\rangle +$
$1/\sqrt{8}|011\rangle + 1/\sqrt{8}|100\rangle + 1/\sqrt{8}|101\rangle +$
$1/\sqrt{8}|110\rangle + 1/\sqrt{8}|111\rangle$



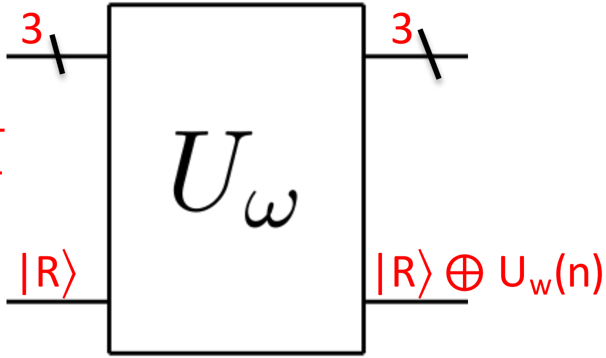
Process all input combinations simultaneously

- Measurement yields 1 of 8 input states and corresponding R

Quantum Advantage



$1/\sqrt{8}|000\rangle + 1/\sqrt{8}|001\rangle + 1/\sqrt{8}|010\rangle +$
$1/\sqrt{8}|011\rangle + 1/\sqrt{8}|100\rangle + 1/\sqrt{8}|101\rangle +$
$1/\sqrt{8}|110\rangle + 1/\sqrt{8}|111\rangle$



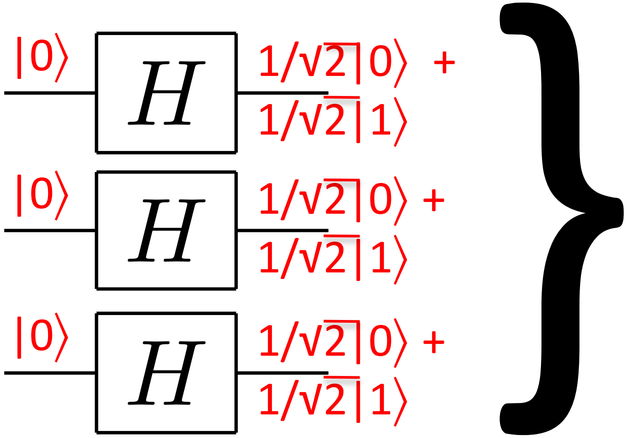
Process all input combinations simultaneously

- Measurement yields 1 of 8 input states and corresponding R

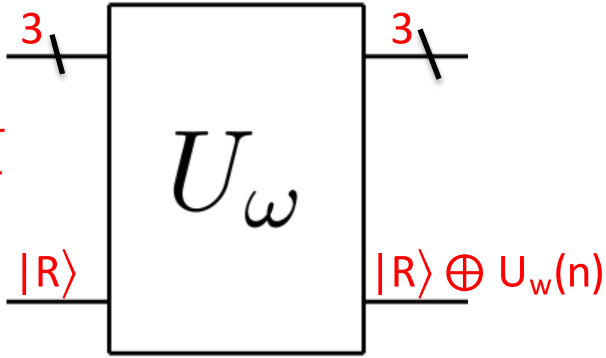
R flips ($|0\rangle \rightarrow |1\rangle$ or $|1\rangle \rightarrow |0\rangle$) for values where U_w evaluates to 1

- Funny things happen when R is not a pure state ($|0\rangle$ or $|1\rangle$)

Quantum Advantage



$1/\sqrt{8}|000\rangle + 1/\sqrt{8}|001\rangle + 1/\sqrt{8}|010\rangle + 1/\sqrt{8}|011\rangle + 1/\sqrt{8}|100\rangle + 1/\sqrt{8}|101\rangle + 1/\sqrt{8}|110\rangle + 1/\sqrt{8}|111\rangle$



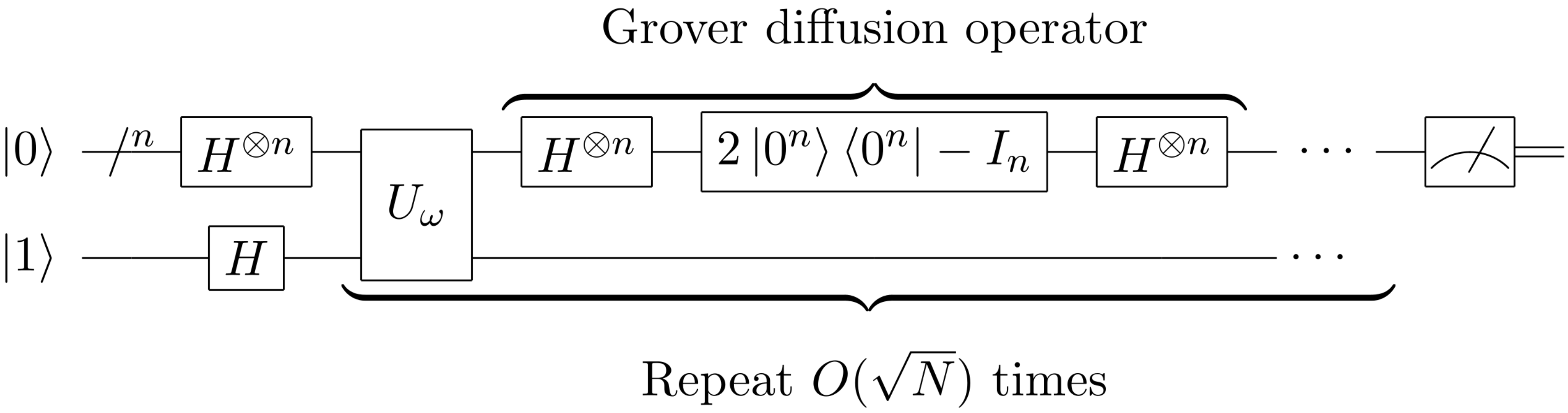
Process all input combinations simultaneously

- Measurement yields 1 of 8 input states and corresponding R

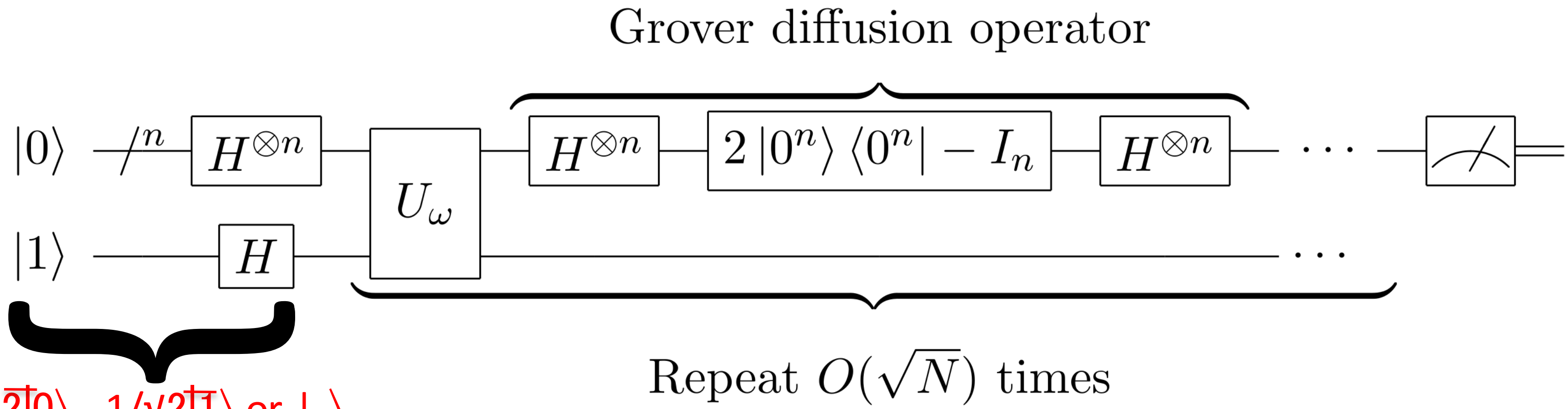
R flips ($|0\rangle \rightarrow |1\rangle$ or $|1\rangle \rightarrow |0\rangle$) for values where U_w evaluates to 1

- Funny things happen when R is not a pure state ($|0\rangle$ or $|1\rangle$)

Non-trivial to extract desired input value



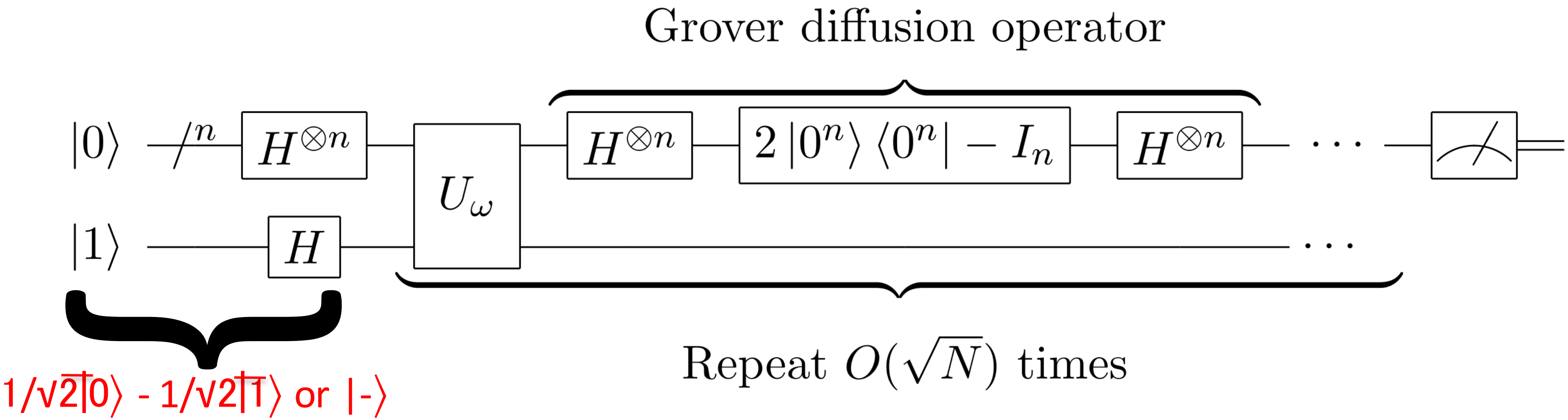
n input qubits in equal superposition of all *N* states



$1/\sqrt{2}|0\rangle - 1/\sqrt{2}|1\rangle$ or $|-\rangle$

n input qubits in equal superposition of all *N* states

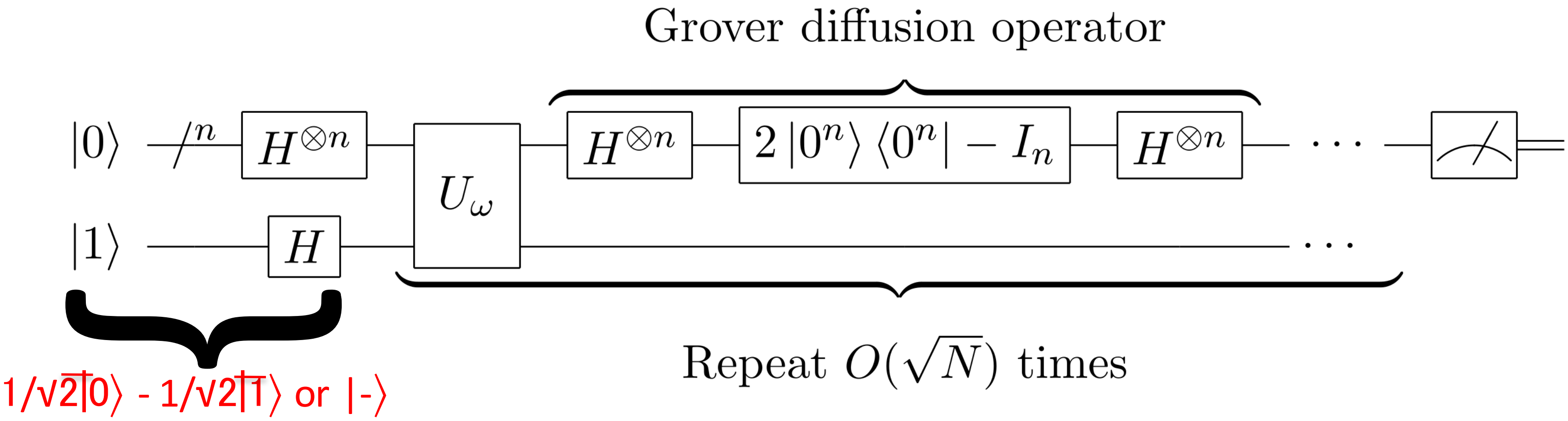
R is set to to $|-\rangle$ state



n input qubits in equal superposition of all N states

R is set to to |-⟩ state

U_w is the function encoded as a black box



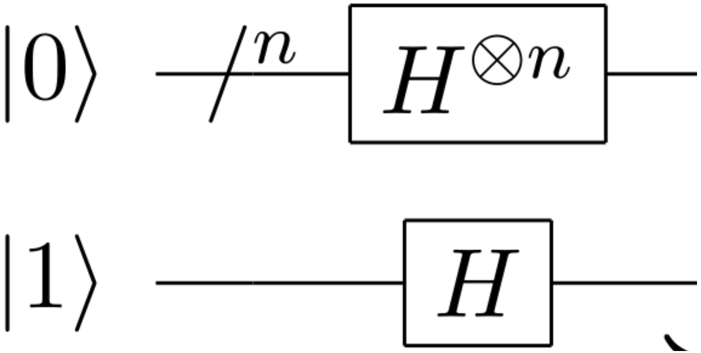
n input qubits in equal superposition of all *N* states

R is set to to |-⟩ state

U_w is the function encoded as a black box

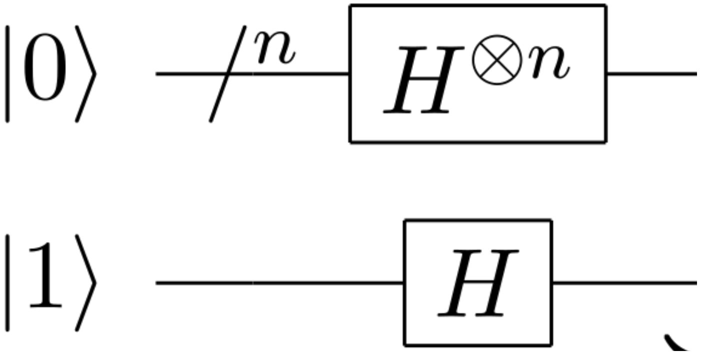
Grover diffusion operator reflects probability amplitudes around the average

Grover’s Algorithm - Setup



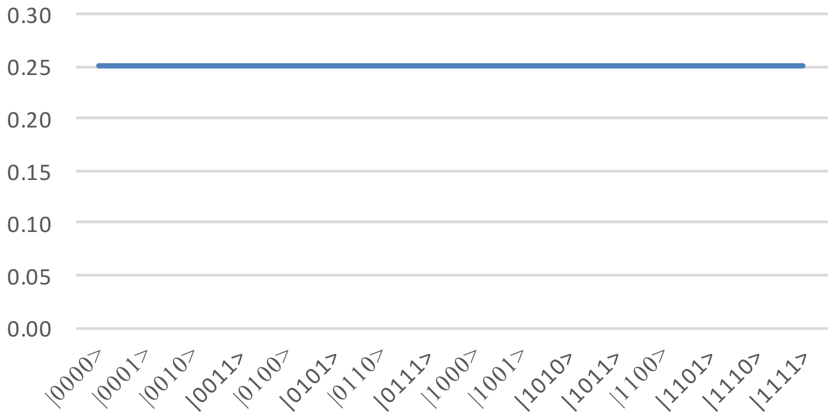
Apply Hadamard on *n (4)* input qubits for 2^n (16) input states

Grover's Algorithm - Setup

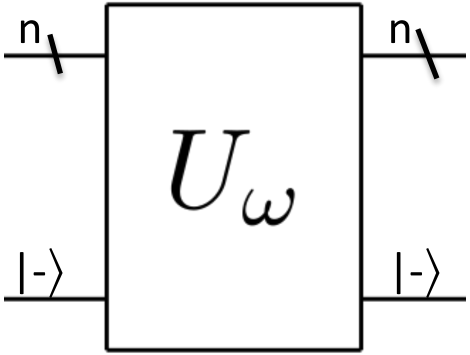


Apply Hadamard on *n (4)* input qubits for 2^n (16) input states

Creates equal superposition of all $N = 2^n$ input states, each with probability amplitude of $1/\sqrt{N}$ (1/4)



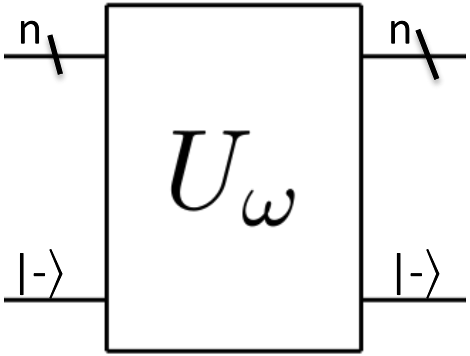
1st Iteration – Phase Inversion



Inputs go into black box along with ancilla qubit in |-⟩ state

Funny thing: R (|-⟩) remains the same for all states

1^{st} Iteration – Phase Inversion



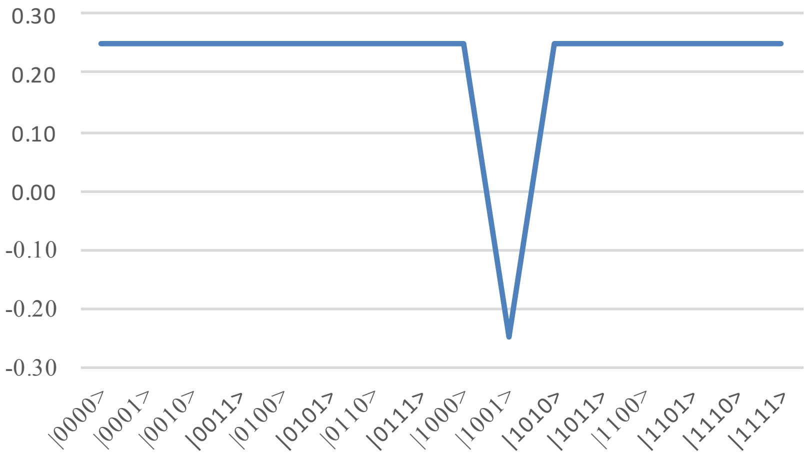
Inverts phase of desired input from $1/\sqrt{N}$ (1/4) to $-1/\sqrt{N}$ (-1/4)

$1/\sqrt{16}|0000\rangle + 1/\sqrt{16}|0001\rangle + \dots$
$- 1/\sqrt{16}|1001\rangle + \dots + 1/\sqrt{16}|1111\rangle$

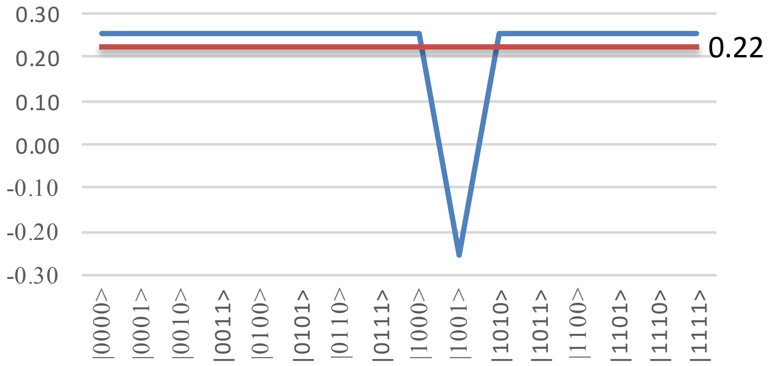
Measurement at this stage would yield completely random n & R

Inputs go into black box along with ancilla qubit in |-⟩ state

Funny thing: R (|-⟩) remains the same for all states

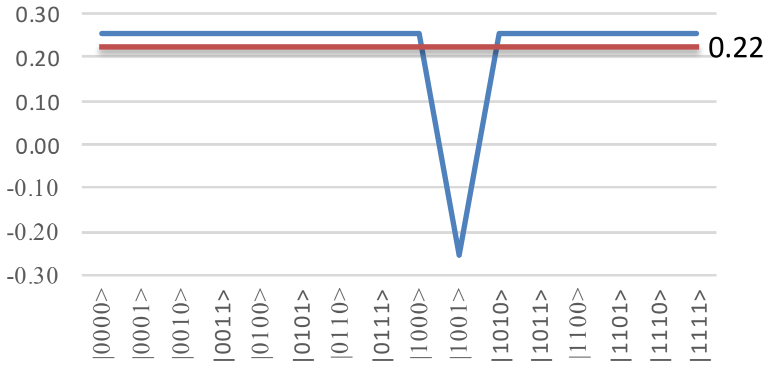


1st Iteration - Diffusion



In this stage the probability amplitudes of all states reflects around the average

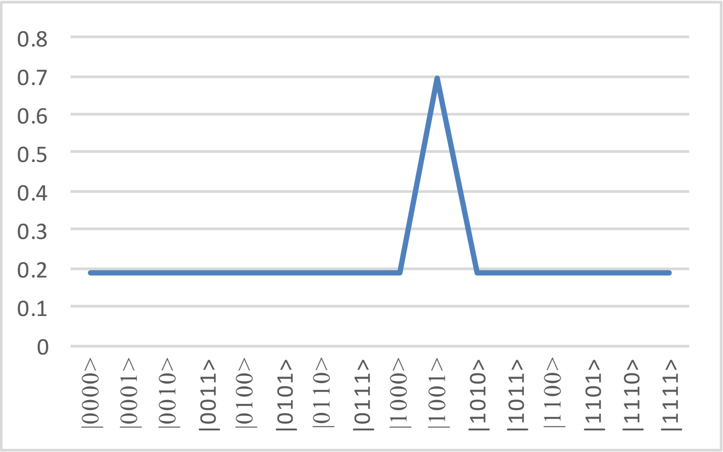
1st Iteration - Diffusion



In this stage the probability amplitudes of all states reflects around the average

Measurement at this stage would give us the desired input with a higher probability

But is that good enough?



$\sqrt{N}$ Iterations

We repeat the phase inversion and diffusion steps for $\sqrt{N}$ iterations

Desired result with high probability

Can be extended to work for multiple matching inputs

Grover’s Oracle

Grover’s Oracle

The Grover’s black box / Oracle contains the function we need to invert

Grover’s Oracle

The Grover’s black box / Oracle contains the function we need to invert

Need to succinctly and efficiently represent the oracle while preserving rules of quantum computation

Grover’s Oracle

The Grover’s black box / Oracle contains the function we need to invert

Need to succinctly and efficiently represent the oracle while preserving rules of quantum computation

This is where NChooseK comes in

NChooseK

NChooseK

Single parameterized primitive

- Can be used to express wide variety of problems

NChooseK

Single parameterized primitive

Can be used to express wide variety of problems

Constrains *k* of *n* boolean variables to true

NChooseK

Single parameterized primitive

- Can be used to express wide variety of problems

Constrains *k* of *n* boolean variables to true

$$
\boxed{\begin{array}{l}
nck(\{a, b, c\}, \{0, 1\}) \\
nck(\{b, c, d\}, \{2, 3\}) \\
nck(\{c, d, e\}, \{1\})
\end{array}}
$$

Fig. 2. Trivial example of an NChooseK program

NChooseK Example

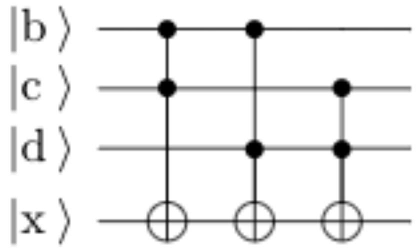


Fig. 3. A quantum black box for $nck(\{b, c, d\}, \{2, 3\})$

Maps quantum state from $|bcd\rangle\ |x\rangle$ to

$|bcd\rangle\ |x\oplus 1\rangle$: when 2 or 3 of $|b\rangle$, $|c\rangle$ and $|d\rangle$ are $|1\rangle$

$|bcd\rangle\ |x\rangle$: otherwise

Generality of NChooseK

Generality of NChooseK

High-level – abstracts away underlying architecture

Generality of NChooseK

High-level – abstracts away underlying architecture

Enables formal specification with unique interpretation across architectures

Generality of NChooseK

High-level – abstracts away underlying architecture

Enables formal specification with unique interpretation across architectures

Can be easily integrated into classical workloads

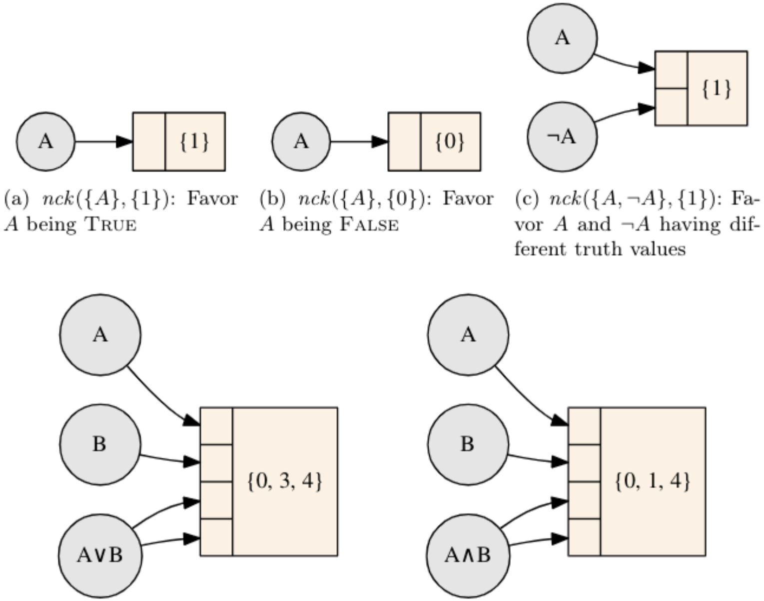
Circuit Satisfiability

Problem: Given boolean expression, find set of inputs to evaluate expression as true

Circuit Satisfiability

Problem: Given boolean expression, find set of inputs to evaluate expression as true

Primitive operations can be used to express circuit satisfiability in NChooseK terms



(a) $nck(\{A\},\{1\})$: Favor A being TRUE

(b) $nck(\{A\},\{0\})$: Favor A being FALSE

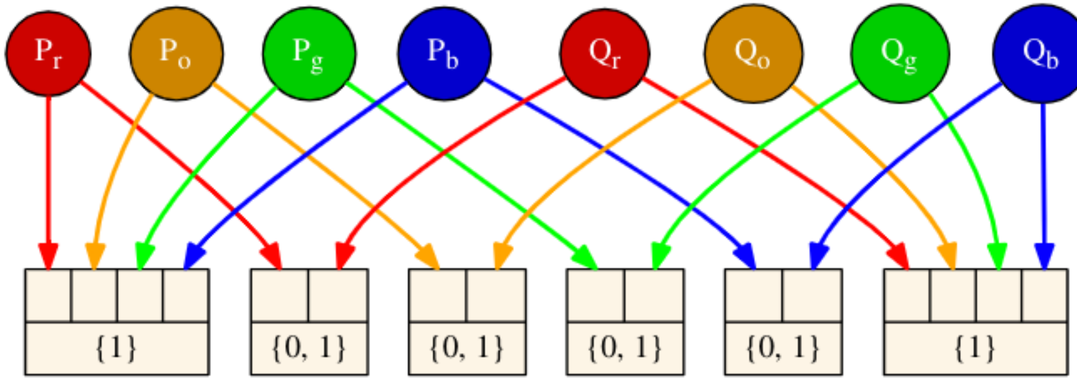
(c) $nck(\{A,\neg A\},\{1\})$: Favor A and $\neg A$ having different truth values

(d) $nck(\{A, B, A \vee B, A \vee B\},\{0,3,4\})$: Favor $A \vee B$ being TRUE if and only if at least one of A or B is TRUE

(e) $nck(\{A, B, A \wedge B, A \wedge B\},\{0,1,4\})$: Favor $A \wedge B$ being TRUE if and only if both A and B are TRUE

Fig. 5. NchooseK building blocks for circuit satisfiability

Map Coloring



Problem: Color map with *c* colors with adjacent regions with different colors

Problem expressed as NChooseK primitives

Code Generator

We implement code generator for IBM Q quantum computers

Given NChooseK primitives, Qiskit code for execution is generated

Code Generator Example

Generated code for XOR as NChooseK({A, B, C}, {0, 2})

Code Generator Example

Generated code for XOR as NChooseK({A, B, C}, {0, 2})

```
q = QuantumRegister(3)
qoutput = QuantumRegister(1)
c = ClassicalRegister(3)
coutput = ClassicalRegister(1)
qc = QuantumCircuit(q, qoutput, c, coutput)

def andInner(t, qx, qz, m, qc):
        if m == 1:
                qc.ccx(t[0], qx[0], qz[0])
        else:
                tmp = QuantumRegister(1)
                qc.add(tmp)
                qc.ccx(t[0], qx[m-1], tmp[0])
                andInner(tmp, qx, qz, m-1, qc)
                qc.ccx(t[0], qx[m-1], tmp[0])
        return qc

def and_nway(qx, qz, n, qc):
        if n == 1:
                qc.cx(qx[0], qz[0])
        else:
                if n == 2:
                        qc.ccx(qx[1], qx[0], qz[0])
                else:
                        t = QuantumRegister(1)
                        qc.add(t)
                        qc.ccx(qx[n-1], qx[n-2], t[0])
                        andInner(t, qx, qz, n-2, qc)
                        qc.ccx(qx[n-1], qx[n-2], t[0])
```

```
        return qc

#Creating equal superposition.
qc.h(q)

qc.x(q)
and_nway(q, qoutput, 3, qc)
qc.x(q)

qc.x(q[0])
and_nway(q, qoutput, 3, qc)
qc.x(q[0])

qc.x(q[1])
and_nway(q, qoutput, 3, qc)
qc.x(q[1])

qc.x(q[2])
and_nway(q, qoutput, 3, qc)
qc.x(q[2])

qc.measure(q, c)
```

Evaluation

We evaluate the code generator on 2 factors

Evaluation

We evaluate the code generator on 2 factors

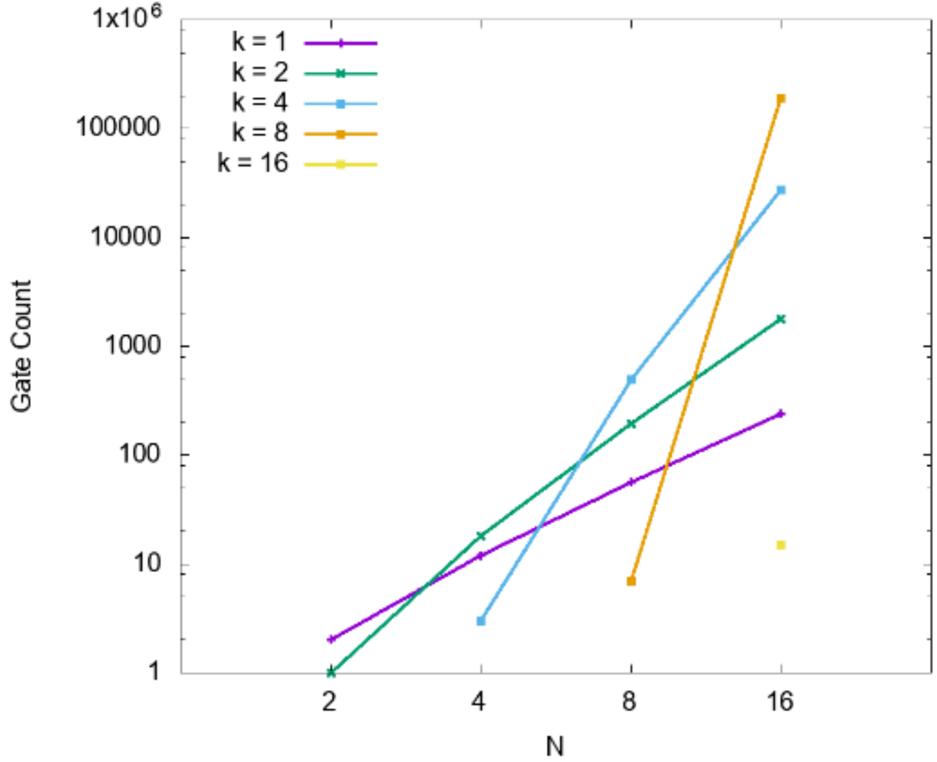
- CCNOT gate count: CCNOT is expensive and cost of the circuit is dominated by it

Evaluation

We evaluate the code generator on 2 factors

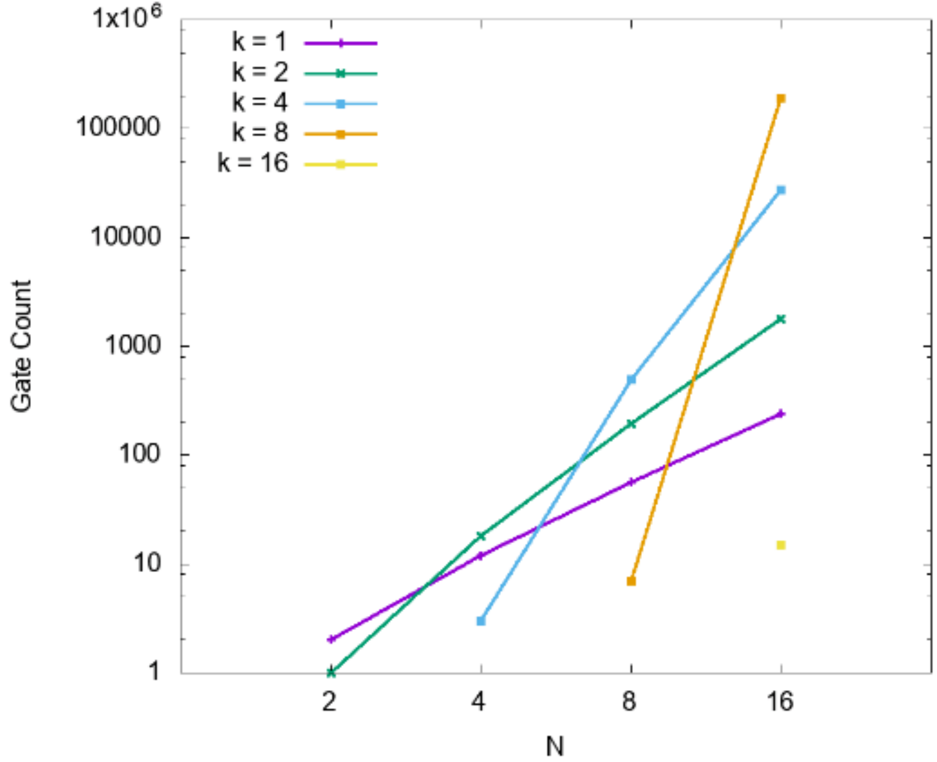
- CCNOT gate count: CCNOT is expensive and cost of the circuit is dominated by it
- Circuit depth: Number of time steps required, important because of qubit decoherence time

CCNOT Gate Count



CCNOT Gate Count

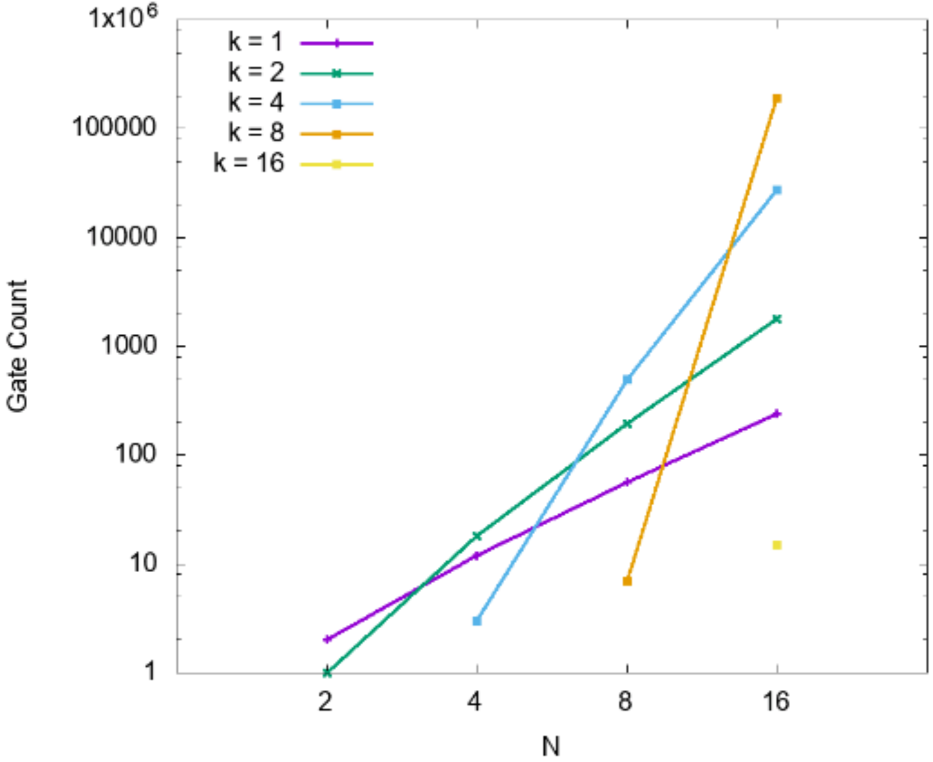
For any *N*, gates required is maximum when *k* = *N/2*



CCNOT Gate Count

For any N, gates required is maximum when $k = N/2$

Gates increases exponentially with N

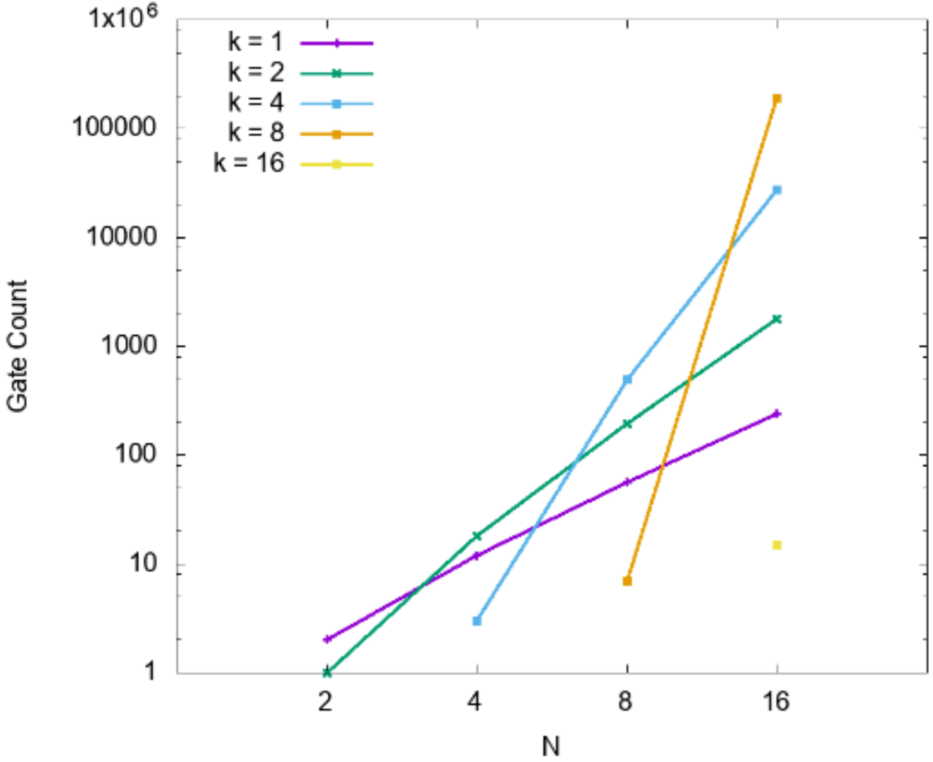


CCNOT Gate Count

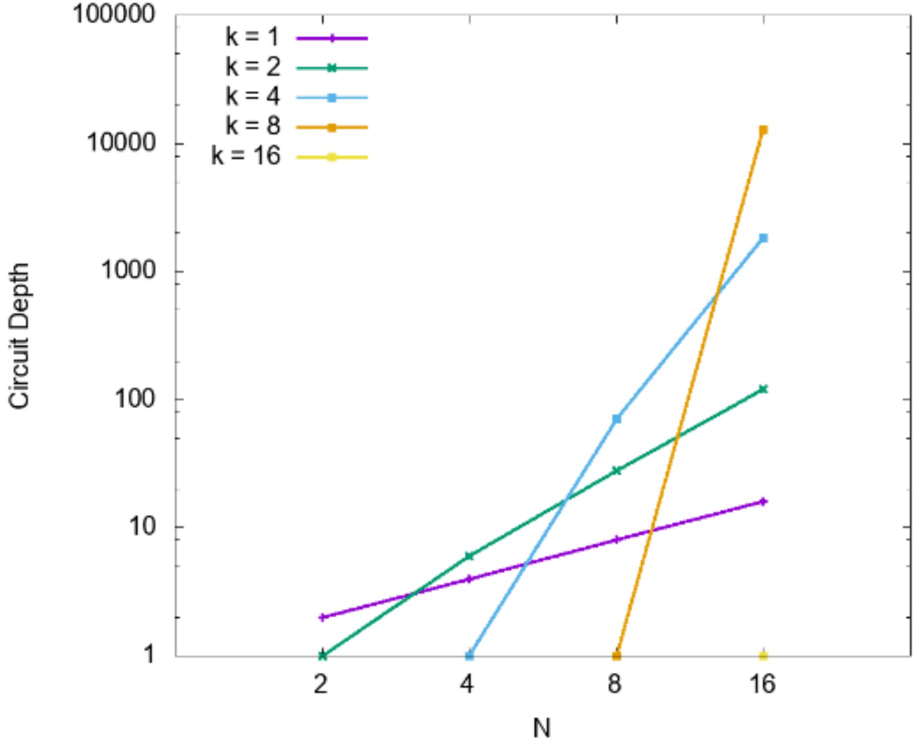
For any N, gates required is maximum when $k = N/2$

Gates increases exponentially with N

Trade-off between using simple circuits and larger NChooseK primitives

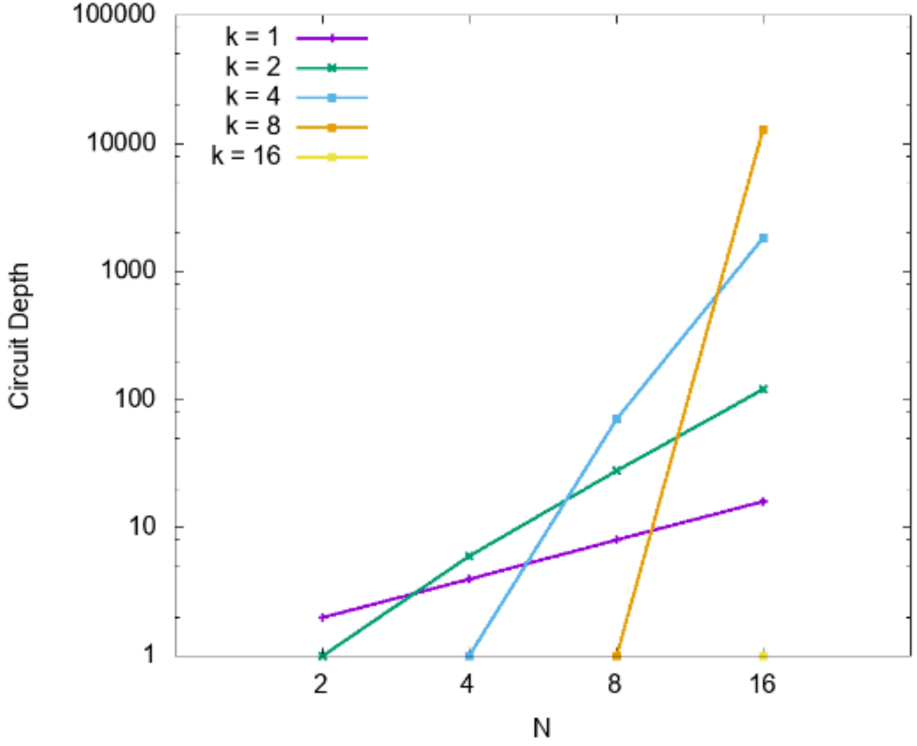


Circuit Depth



Circuit Depth

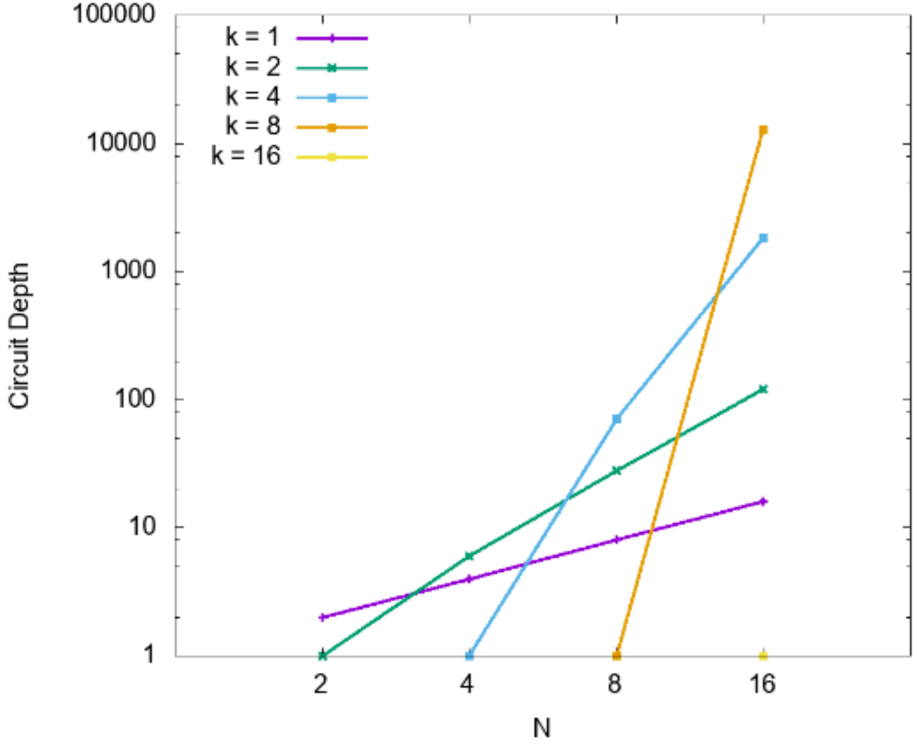
Like gate count, circuit depth is maximum when *k=N/2*



Circuit Depth

Like gate count, circuit depth is maximum when *k=N/2*

Again, depth increases exponentially with increasing *N*

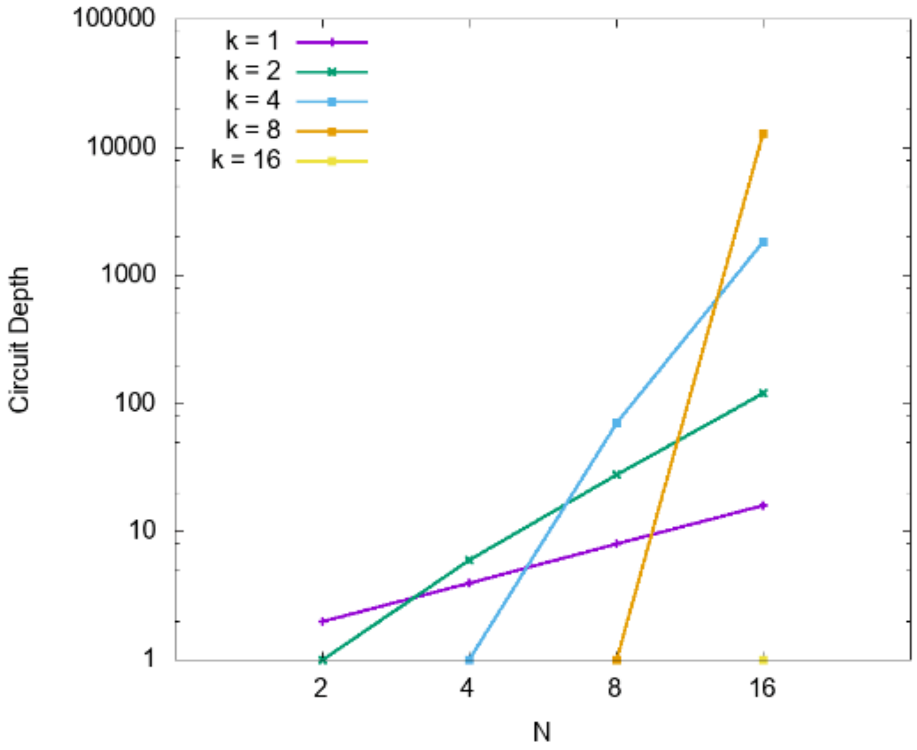


Circuit Depth

Like gate count, circuit depth is maximum when *k=N/2*

Again, depth increases exponentially with increasing *N*

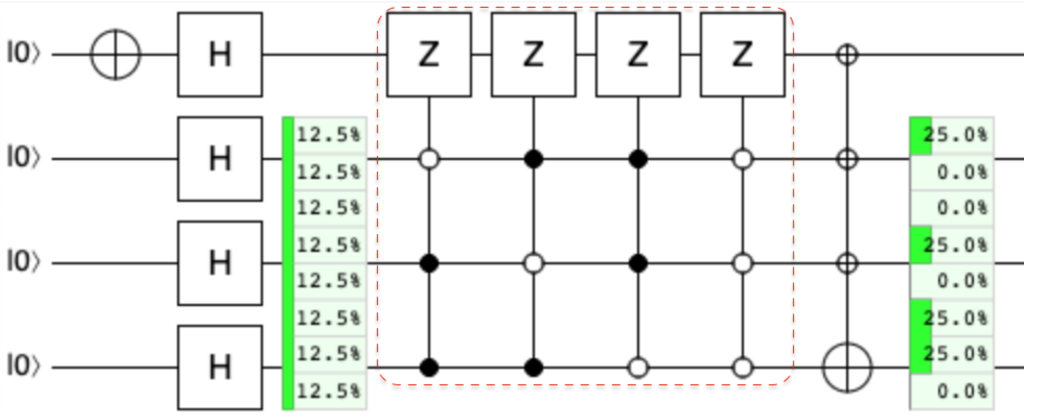
Reaffirms need for establishing trade-off



NChooseK in Grover

Implementation of NChooseK(3,{0,2}) as Quantum Oracle

4 expected outcomes have 25% probability while 0% for others



Conclusion

We present NChooseK for expressing quantum computation

Conclusion

We present NChooseK for expressing quantum computation

We demonstrate generality of the model

Using circuit satisfiability and map coloring

Conclusion

We present NChooseK for expressing quantum computation

We demonstrate generality of the model

Using circuit satisfiability and map coloring

Implement and describe the code generator

Evaluate gate counts and circuit depth for arbitrary N and k parameters

Conclusion

We present NChooseK for expressing quantum computation

We demonstrate generality of the model

Using circuit satisfiability and map coloring

Implement and describe the code generator

Evaluate gate counts and circuit depth for arbitrary N and k parameters

Future work:

Extend generator to combine multiple NChooseK primitives

Also explore trade-off space to automatically break-down into smaller primitives

Acknowledgements

Supported in part by NSF grants1525609 and 1813004 and by the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project numbers 20160069DR and 20190065DR.

Supported by the U.S. Department of Energy through Los Alamos National Laboratory. (contract no.~89233218CNA000001).