Benchmarking GPU Clusters with the Jülich Universal Quantum Computer Simulator

GTC21 | APRIL 14 I DR. DENNIS WILLSCH

JÜLICH Forschungszentrum
JÜLICH SUPERCOMPUTING CENTRE

CONTENTS

JÜLICH SUPERCOMPUTING CENTRE

Dr. Dennis Willsch

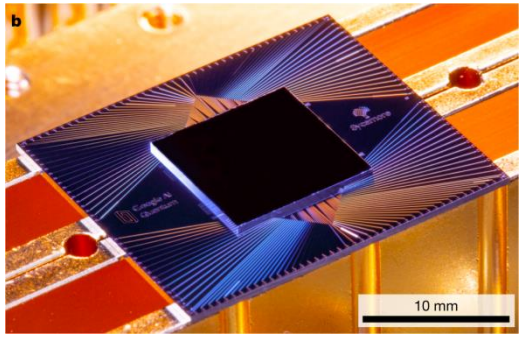
Prof. Dr. Hans De Raedt

Prof. Dr. Kristel Michielsen

QUANTUM COMPUTING

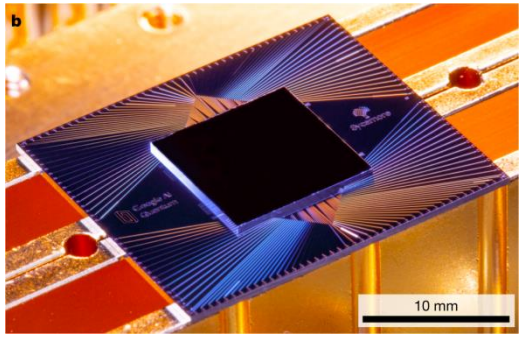
Ideal gate-based quantum computing

- What does a (gate-based) quantum computer do?

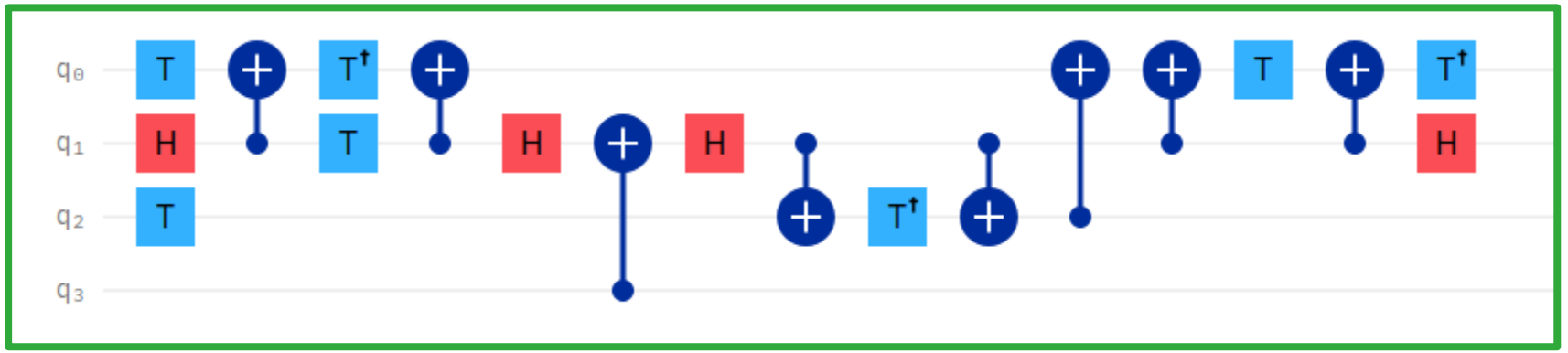


QUANTUM COMPUTING

Ideal gate-based quantum computing

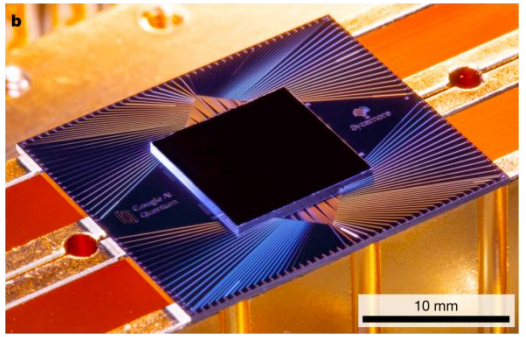


- What does a (gate-based) quantum computer do?
  - It runs a quantum circuit

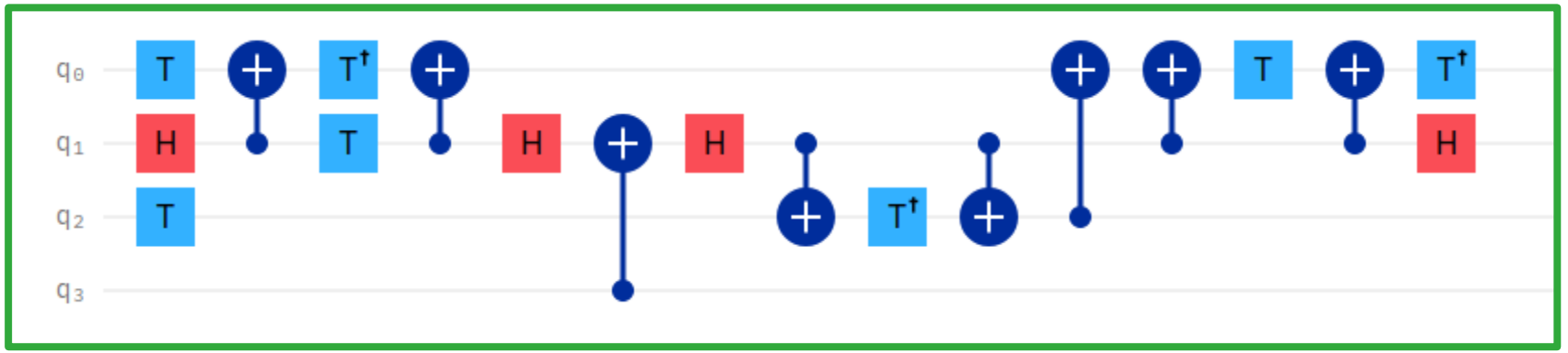


QUANTUM COMPUTING

Ideal gate-based quantum computing



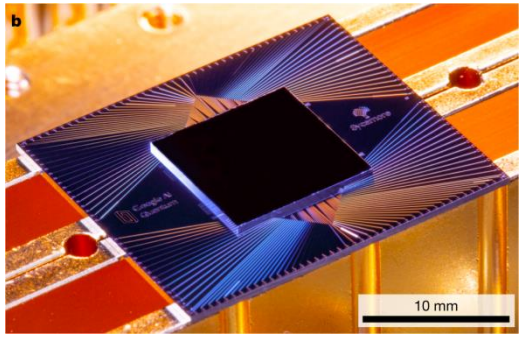
- What does a (gate-based) quantum computer do?
  - It runs a quantum circuit



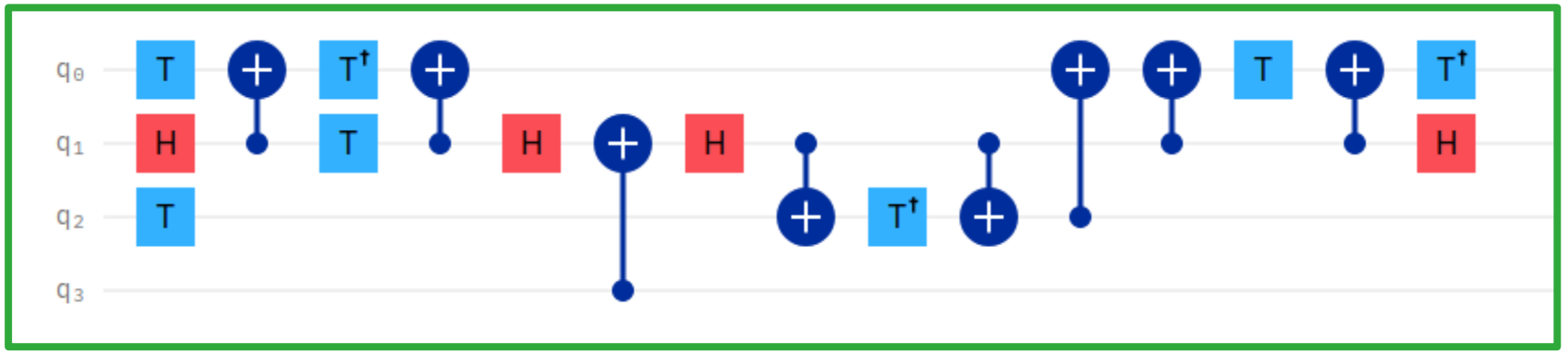
- What does this mean, actually?

QUANTUM COMPUTING

Ideal gate-based quantum computing



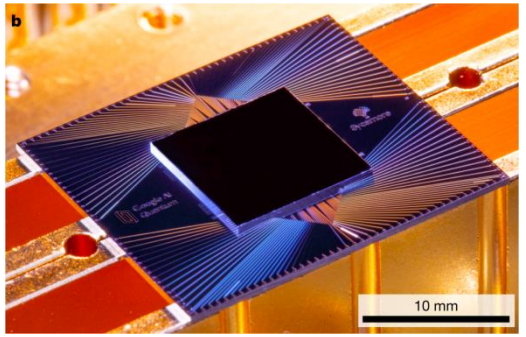
- What does a (gate-based) quantum computer do?
  - It runs a quantum circuit



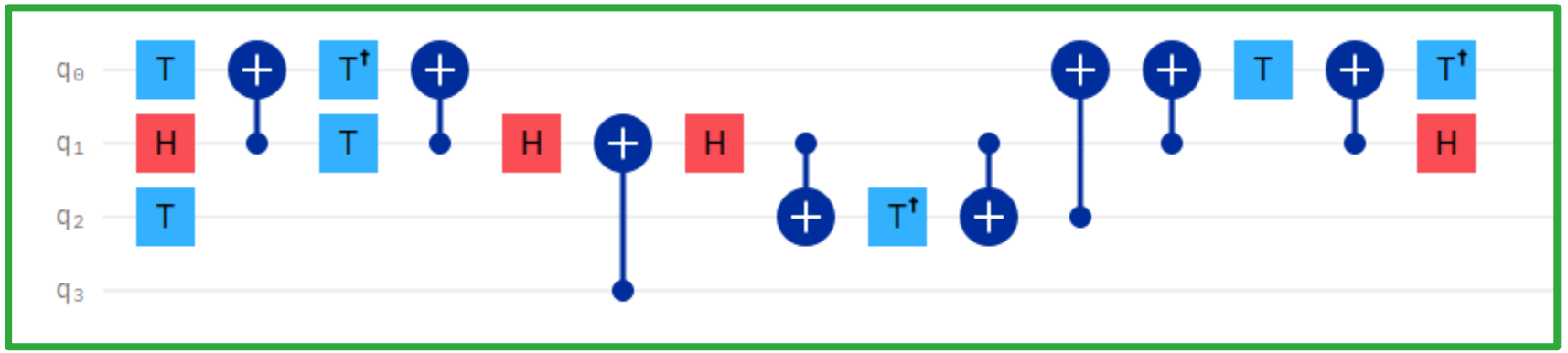
- What does this mean, actually?
  - It performs **matrix-vector multiplications**

QUANTUM COMPUTING

Ideal gate-based quantum computing



- What does a (gate-based) quantum computer do?
  - It runs a quantum circuit

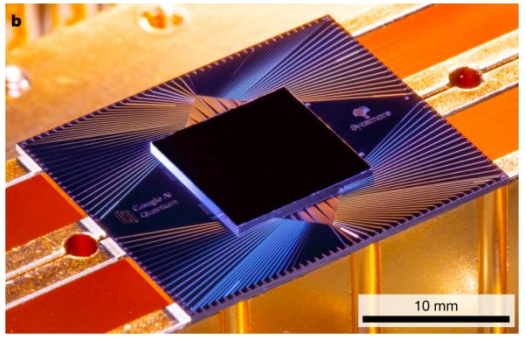


- What does this mean, actually?
  - It performs **matrix-vector multiplications** that are

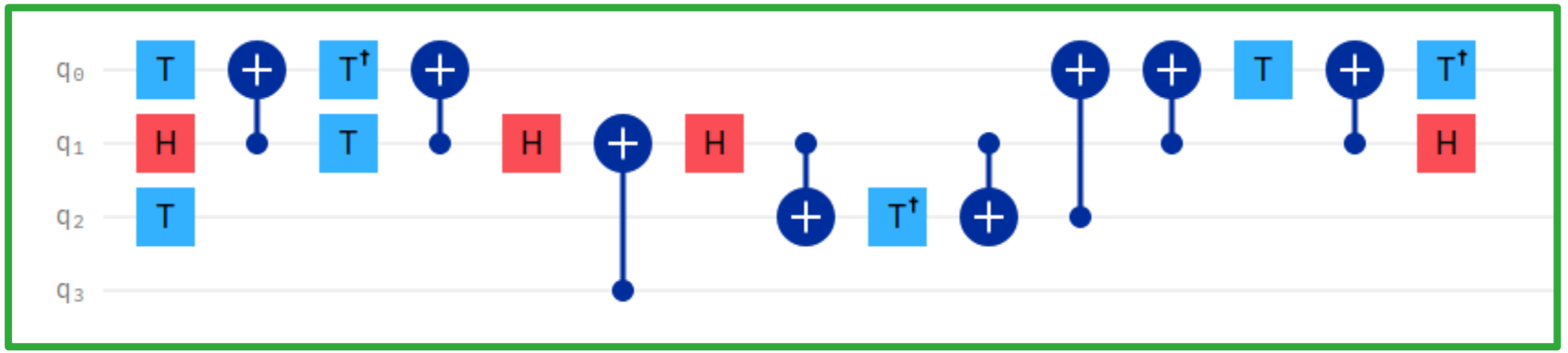
sparse

QUANTUM COMPUTING

Ideal gate-based quantum computing



- What does a (gate-based) quantum computer do?
  - It runs a quantum circuit

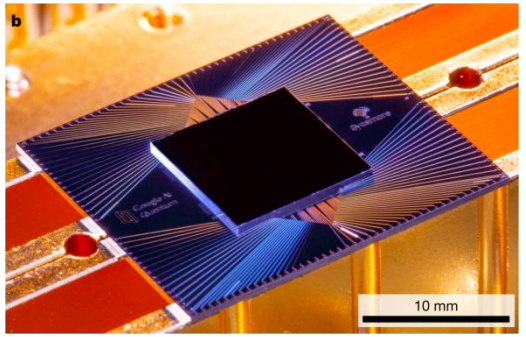


- What does this mean, actually?
  - It performs **matrix-vector multiplications** that are

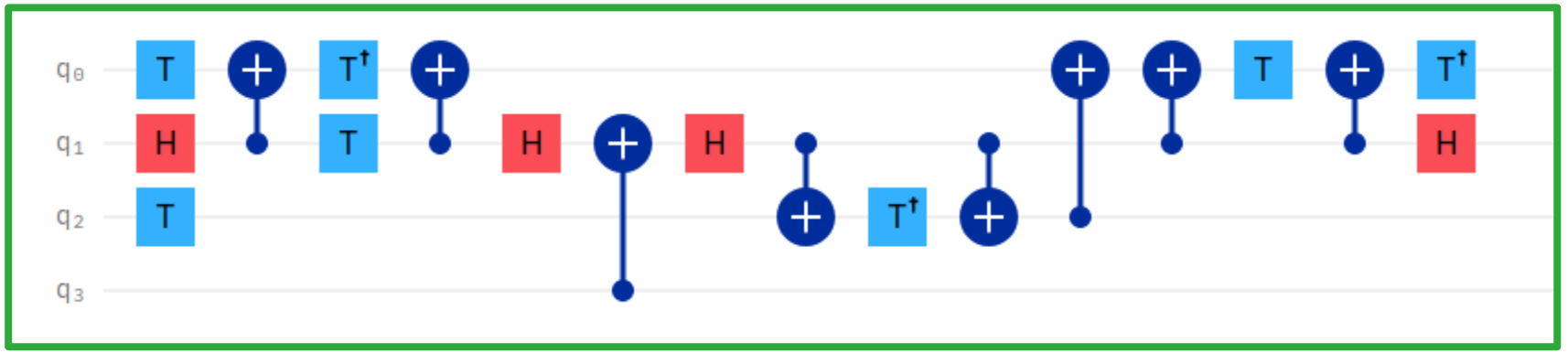
sparse **complex**

QUANTUM COMPUTING

Ideal gate-based quantum computing



- What does a (gate-based) quantum computer do?
  - It runs a quantum circuit

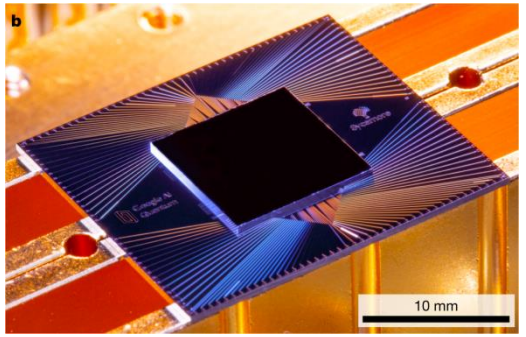


- What does this mean, actually?
  - It performs **matrix-vector multiplications** that are

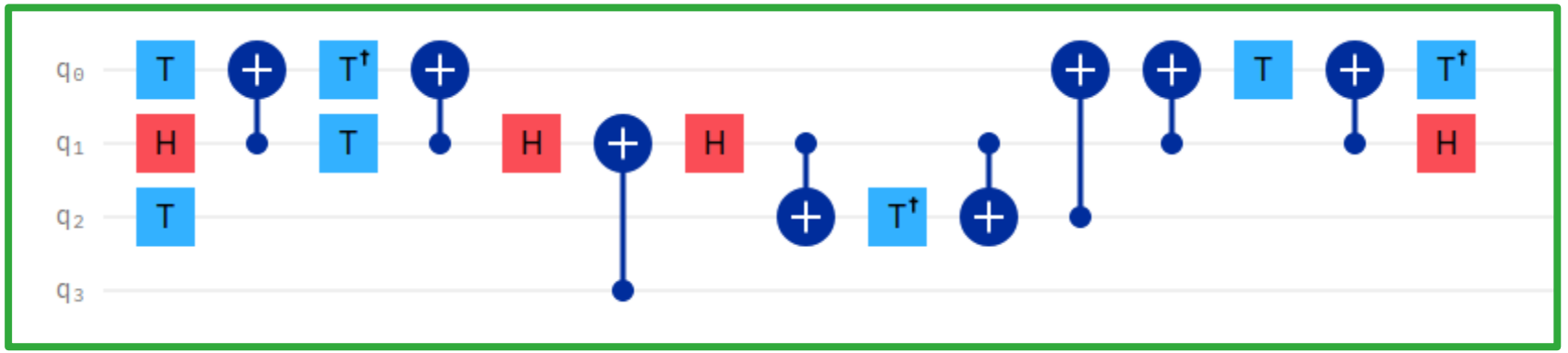
sparse **complex** **unitary**

QUANTUM COMPUTING

Ideal gate-based quantum computing



- What does a (gate-based) quantum computer do?
  - It runs a quantum circuit



- What does this mean, actually?
  - It performs **matrix-vector multiplications** that are

    sparse **complex** **unitary**

  - with **huge** vectors and **huge²** matrices

QUANTUM COMPUTING

Ideal gate-based quantum computing

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

- **Example:**

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

- **Example:**

$n = 4$ qubits
$2^n = 16$ complex numbers

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

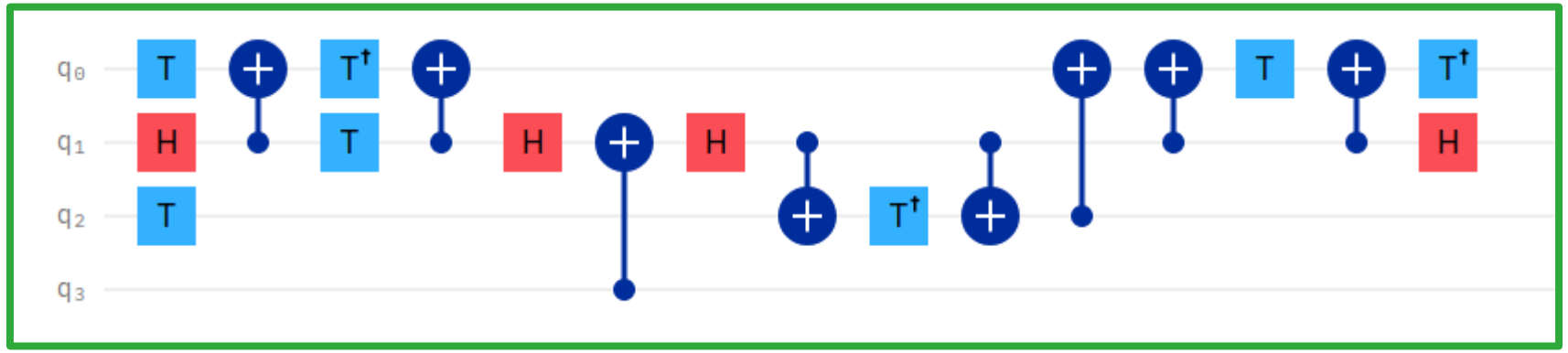
$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

- **Example:**

$n = 4$ qubits
$2^n = 16$ complex numbers



QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

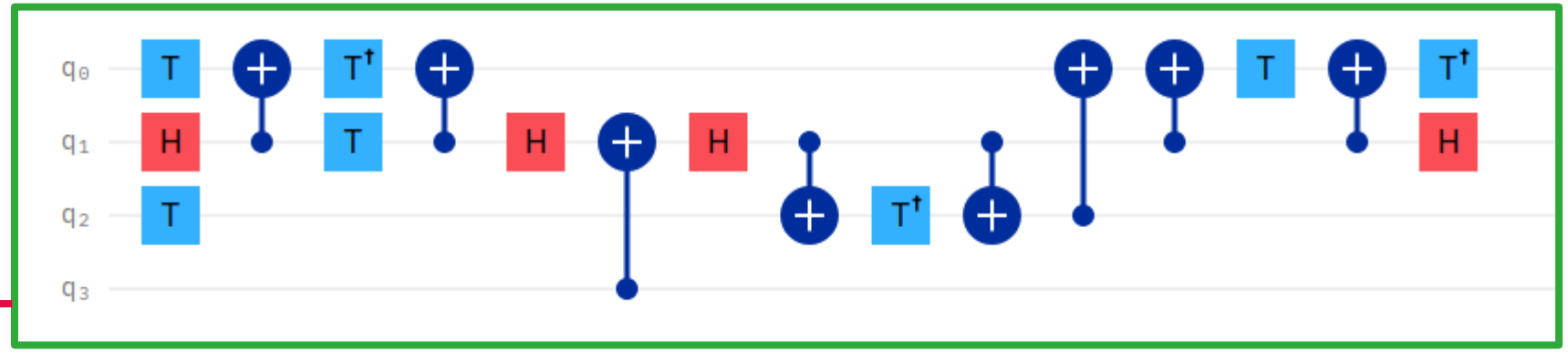
$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

- **Example:**

$n = 4$ qubits
$2^n = 16$ complex numbers



Initial state of QC:

$$|\psi\rangle = |0000\rangle = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

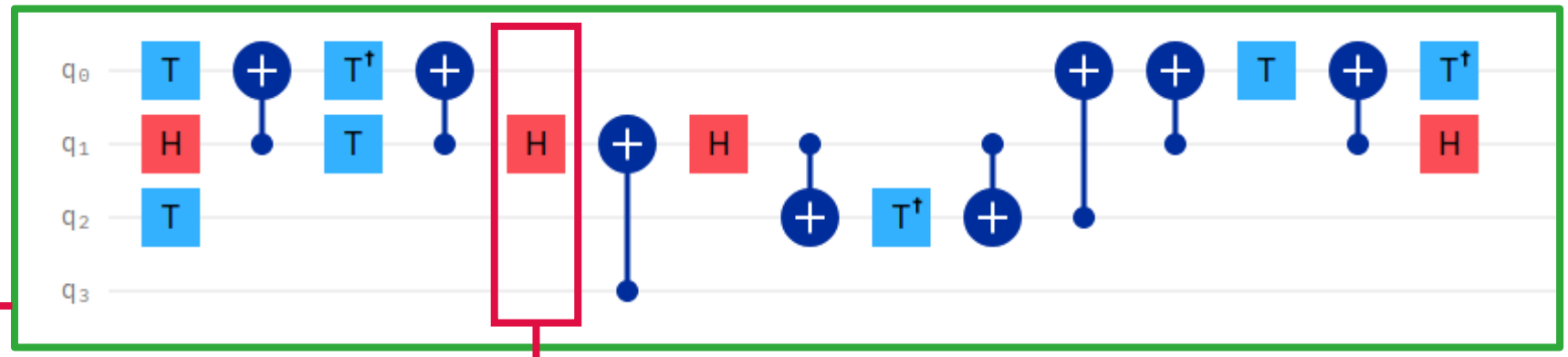
$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

- **Example:**

$n = 4$ qubits
$2^n = 16$ complex numbers



Initial state of QC:

$$|\psi\rangle = |0000\rangle = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Example matrix-vector multiplication: H on qubit q_1

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

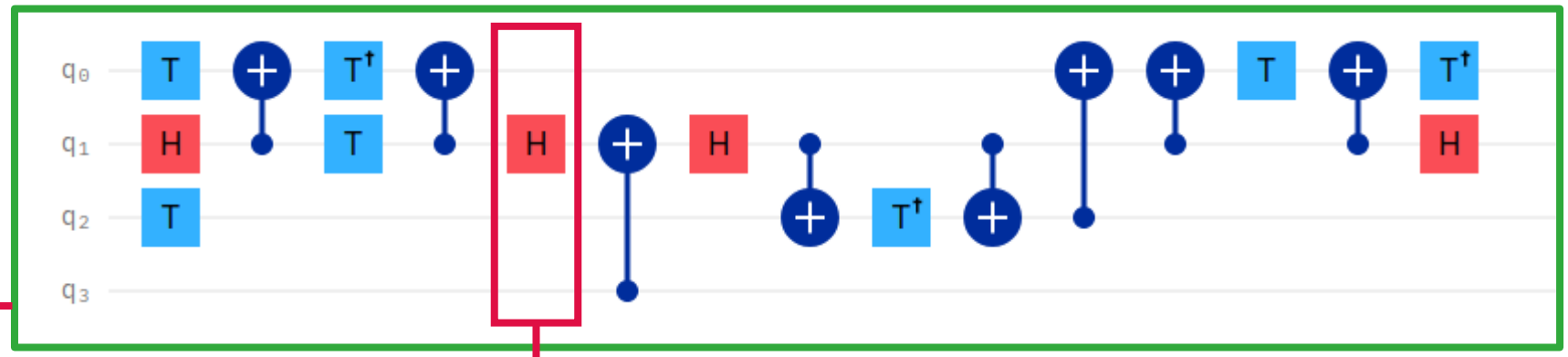
$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

- **Example:**

$n = 4$ qubits
$2^n = 16$ complex numbers



Initial state of QC:

$$|\psi\rangle = |0000\rangle = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Example matrix-vector multiplication: H on qubit q_1

For each q_3, q_2, q_0
perform 2x2 update:

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

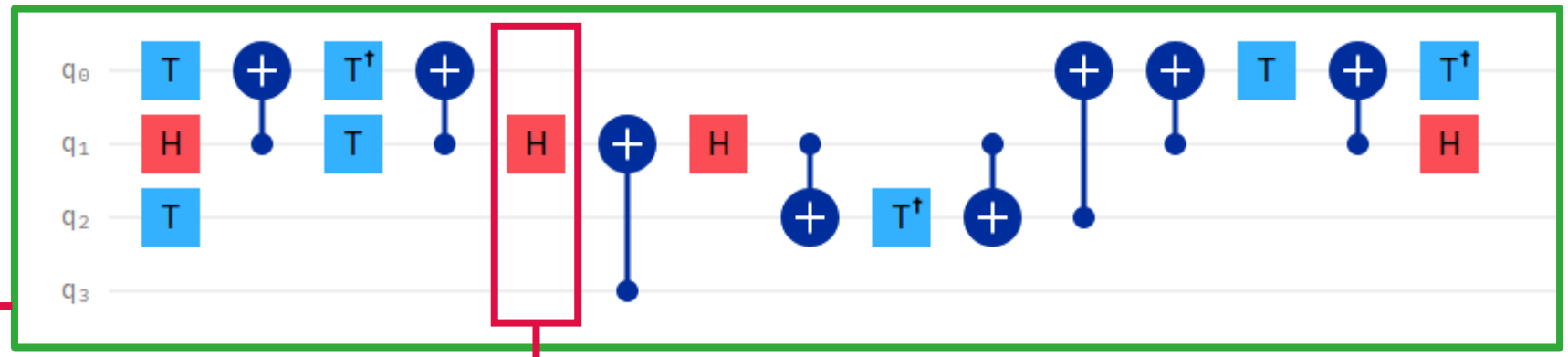
$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

- **Example:**

$n = 4$ qubits
$2^n = 16$ complex numbers



Initial state of QC:

$$|\psi\rangle = |0000\rangle = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Example matrix-vector multiplication: H on qubit q_1

For each q_3, q_2, q_0 perform 2x2 update:

$$\begin{pmatrix} \psi_{q_3 q_2 0 q_0} \\ \psi_{q_3 q_2 1 q_0} \end{pmatrix} \leftarrow \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \psi_{q_3 q_2 0 q_0} \\ \psi_{q_3 q_2 1 q_0} \end{pmatrix}$$

QUANTUM COMPUTING

Ideal gate-based quantum computing

vector = state of the QC = 2^n complex numbers

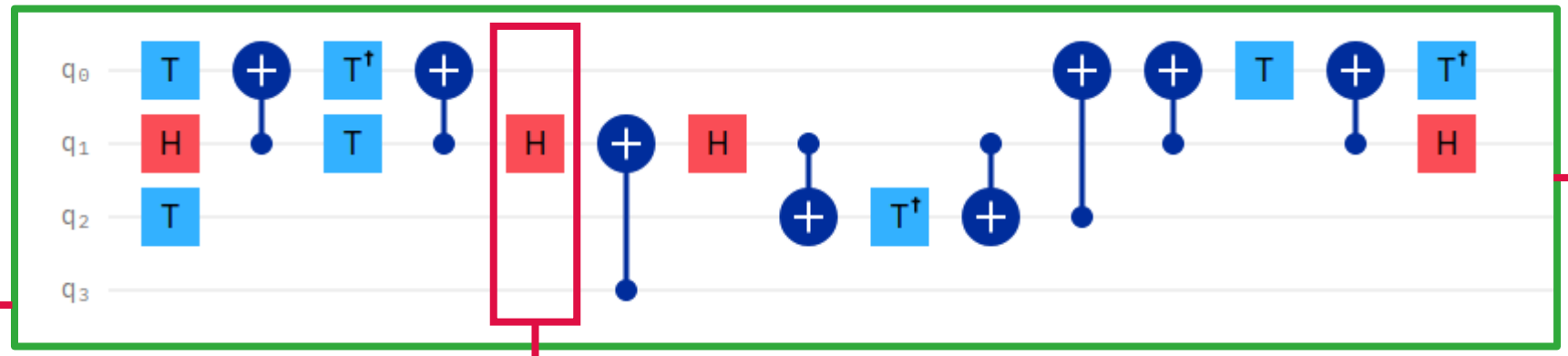
$$|\psi\rangle = \psi_{0\cdots0}|0\cdots0\rangle + \cdots + \psi_{1\cdots1}|1\cdots1\rangle = \begin{pmatrix} \psi_{0\cdots0} \\ \vdots \\ \psi_{1\cdots1} \end{pmatrix}$$

- What kind of sparse, unitary **matrix-vector multiplications**, precisely?

each quantum gate = 1 sparse, unitary **matrix**

- **Example:**

$n = 4$ qubits
$2^n = 16$ complex numbers



Initial state of QC:

$$|\psi\rangle = |0000\rangle = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Example matrix-vector multiplication: H on qubit q_1

For each q_3, q_2, q_0 perform 2x2 update:

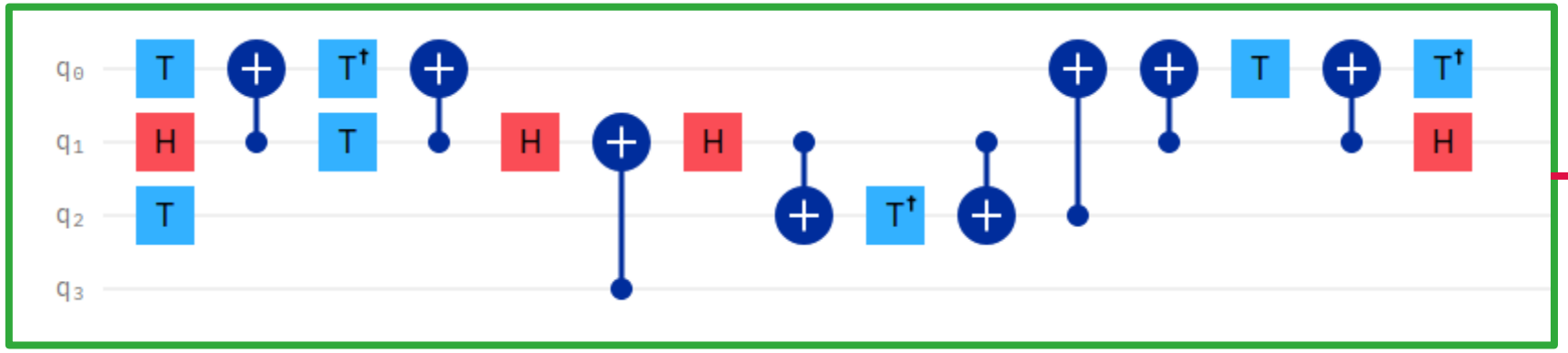
$$\begin{pmatrix} \psi_{q_3 q_2 0 q_0} \\ \psi_{q_3 q_2 1 q_0} \end{pmatrix} \leftarrow \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \psi_{q_3 q_2 0 q_0} \\ \psi_{q_3 q_2 1 q_0} \end{pmatrix}$$

Final state of QC:

$$|\psi\rangle = \begin{pmatrix} \psi_{0000} \\ \vdots \\ \psi_{1111} \end{pmatrix}$$

QUANTUM COMPUTING

Ideal gate-based quantum computing

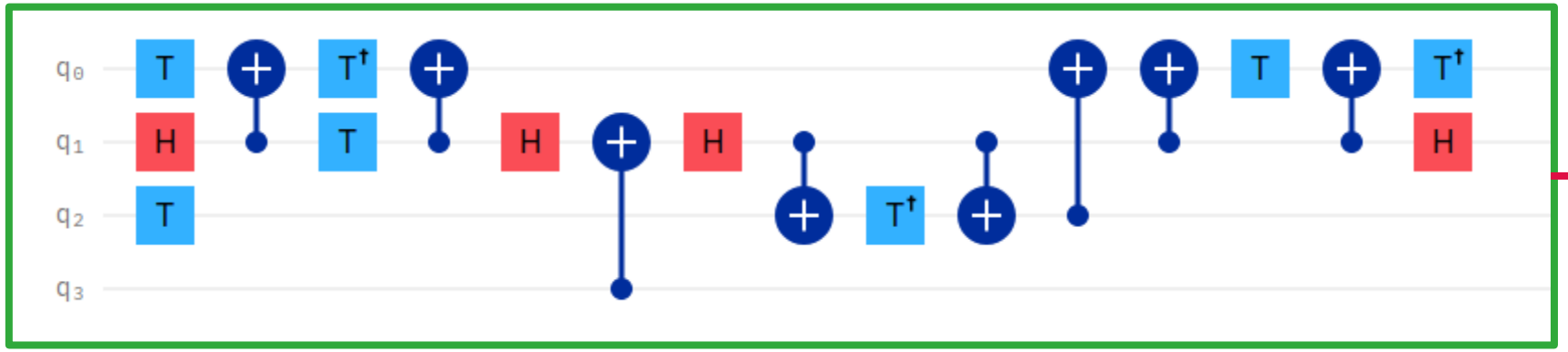


Final state of QC:

$$|\psi\rangle = \begin{pmatrix} \psi_{0000} \\ \vdots \\ \psi_{1111} \end{pmatrix}$$

QUANTUM COMPUTING

Ideal gate-based quantum computing



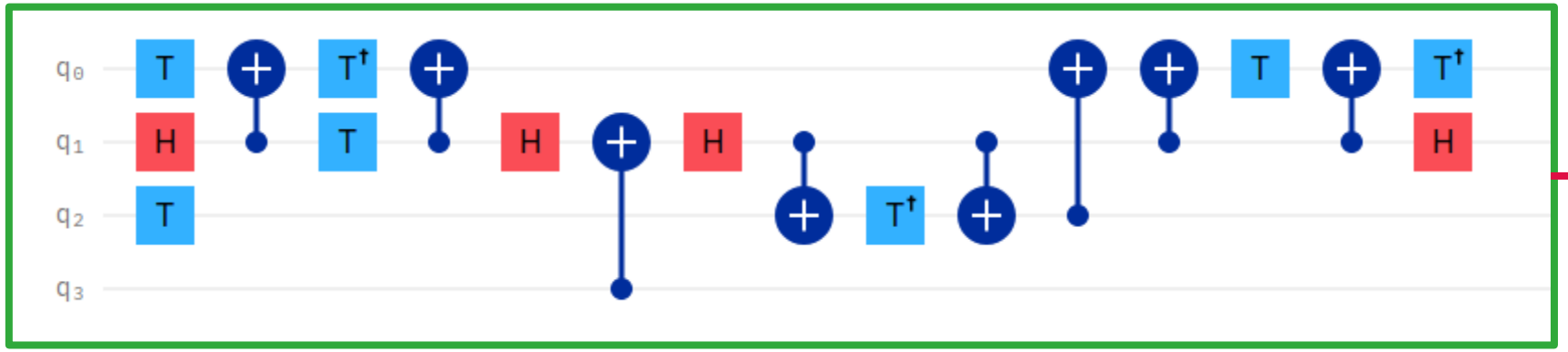
Final state of QC:

$$|\psi\rangle = \begin{pmatrix} \psi_{0000} \\ \vdots \\ \psi_{1111} \end{pmatrix}$$

➢ What does a **hardware realization** of a QC return?

QUANTUM COMPUTING

Ideal gate-based quantum computing



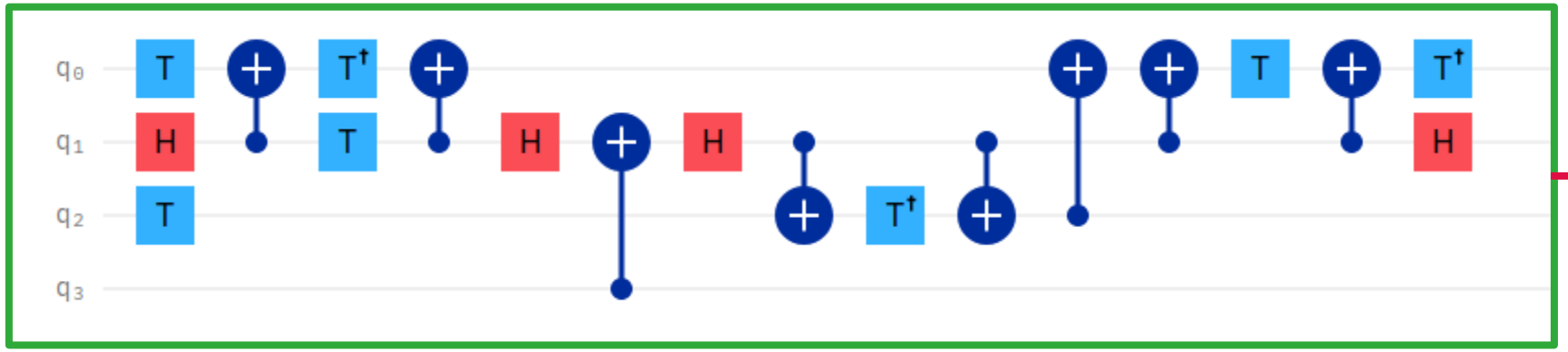
Final state of QC:

$$|\psi\rangle = \begin{pmatrix} \psi_{0000} \\ \vdots \\ \psi_{1111} \end{pmatrix}$$

- What does a **hardware realization** of a QC return?
  - The quantum state after all sparse matrix-vector multiplications?

QUANTUM COMPUTING

Ideal gate-based quantum computing



Final state of QC:

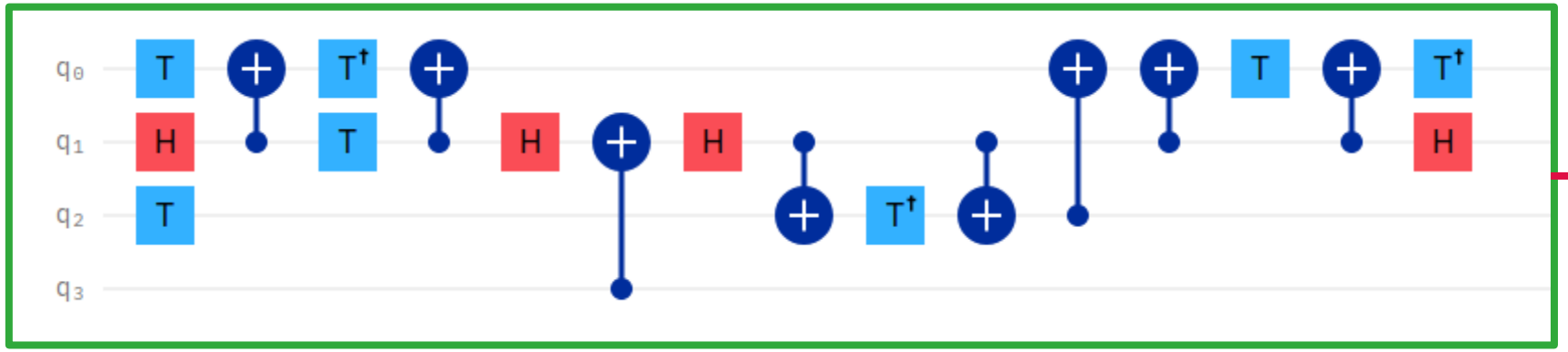
$$|\psi\rangle = \begin{pmatrix} \psi_{0000} \\ \vdots \\ \psi_{1111} \end{pmatrix}$$

- What does a **hardware realization** of a QC return?
  - The quantum state after all sparse matrix-vector multiplications?
  - No! That would be 2^n complex numbers.

For 40 qubits: 2^{40} ψ's = 16 TiB complex numbers

QUANTUM COMPUTING

Ideal gate-based quantum computing



Final state of QC:

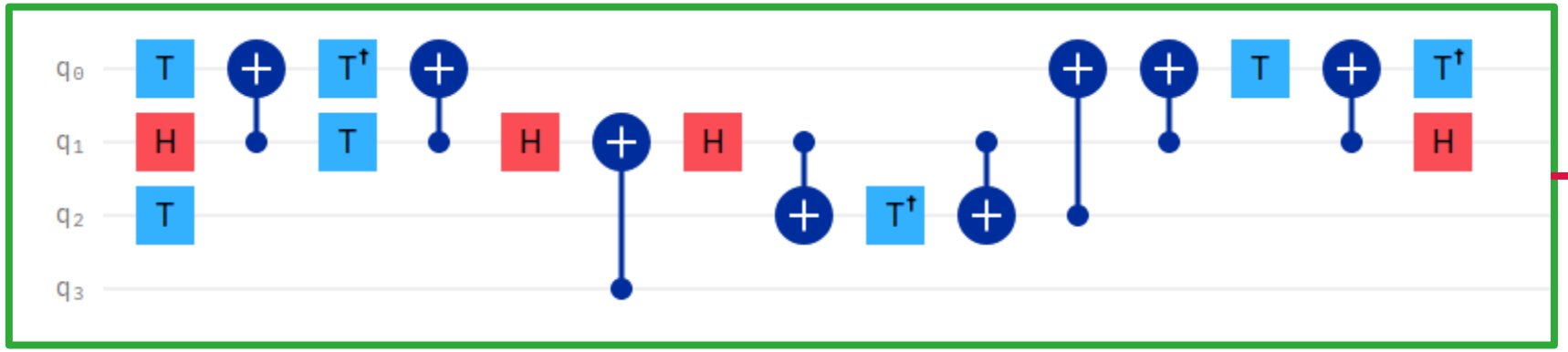
$$|\psi\rangle = \begin{pmatrix} \psi_{0000} \\ \vdots \\ \psi_{1111} \end{pmatrix}$$

- What does a **hardware realization** of a QC return?
  - The quantum state after all sparse matrix-vector multiplications?
  - No! That would be 2^n complex numbers.
  - What then? Only a single bitstring $q_{n-1} \cdots q_1 q_0$ with n bits

For 40 qubits: 2^{40} ψ's = 16 TiB complex numbers

QUANTUM COMPUTING

Ideal gate-based quantum computing



Final state of QC:

$$|\psi\rangle = \begin{pmatrix} \psi_{0000} \\ \vdots \\ \psi_{1111} \end{pmatrix}$$

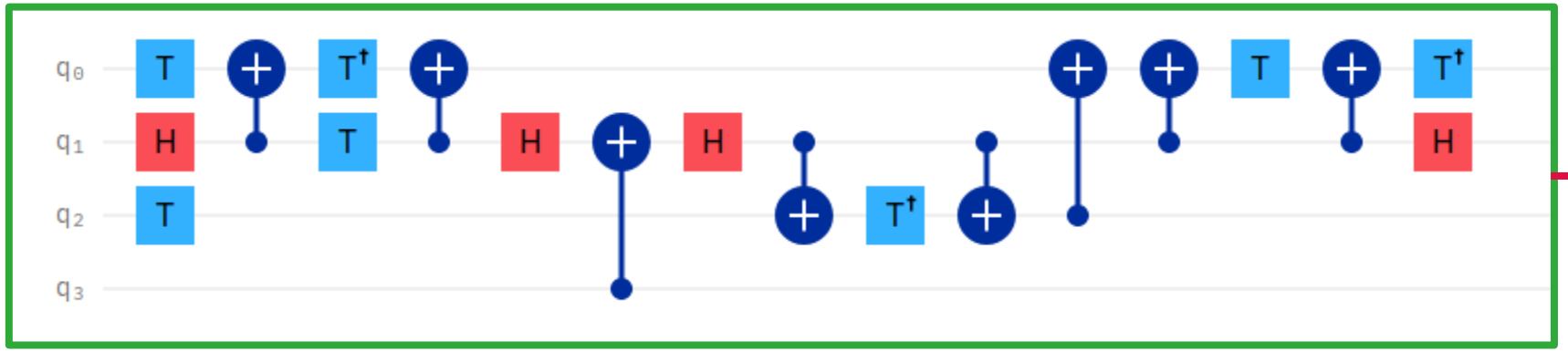
- What does a **hardware realization** of a QC return?
  - The quantum state after all sparse matrix-vector multiplications?
  - No! That would be 2^n complex numbers.
  - What then? Only a single bitstring $q_{n-1} \cdots q_1 q_0$ with n bits
  - The complex numbers only define the **probability**:

For 40 qubits: 2^{40} ψ's = 16 TiB complex numbers

$|\psi_{q_{n-1} \cdots q_1 q_0}|^2$ = probability to return bitstring $q_{n-1} \cdots q_1 q_0$

QUANTUM COMPUTING

Ideal gate-based quantum computing



Final state of QC:

$$|\psi\rangle = \begin{pmatrix} \psi_{0000} \\ \vdots \\ \psi_{1111} \end{pmatrix}$$

- What does a **hardware realization** of a QC return?
  - The quantum state after all sparse matrix-vector multiplications?
  - No! That would be 2^n complex numbers.

For 40 qubits: 2^{40} ψ's = 16 TiB complex numbers

  - What then? Only a single bitstring $q_{n-1} \cdots q_1 q_0$ with n bits
  - The complex numbers only define the **probability**:

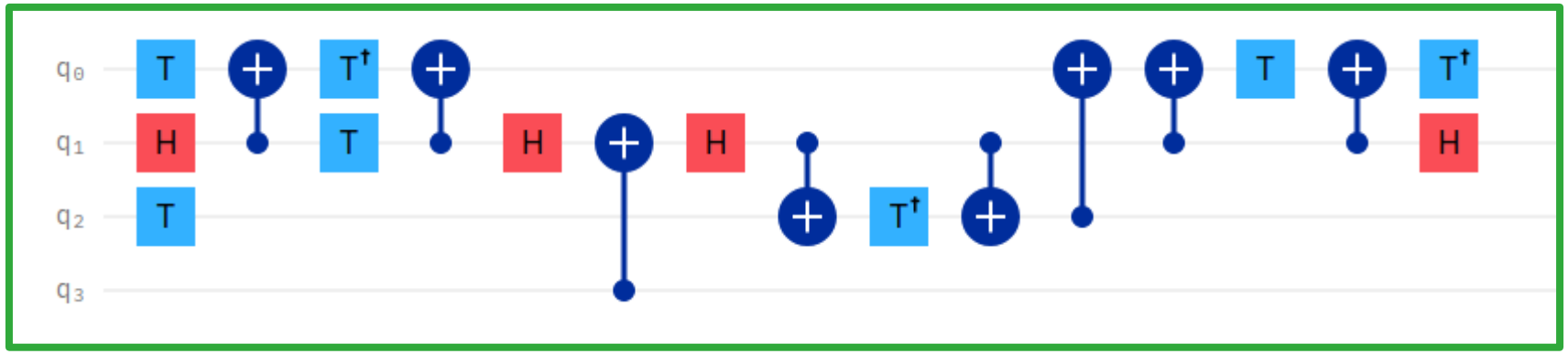
$|\psi_{q_{n-1} \cdots q_1 q_0}|^2$ = probability to return bitstring $q_{n-1} \cdots q_1 q_0$

  - Need to run circuit repeatedly to **sample** from the distribution

QUANTUM COMPUTING

Ideal gate-based quantum computing

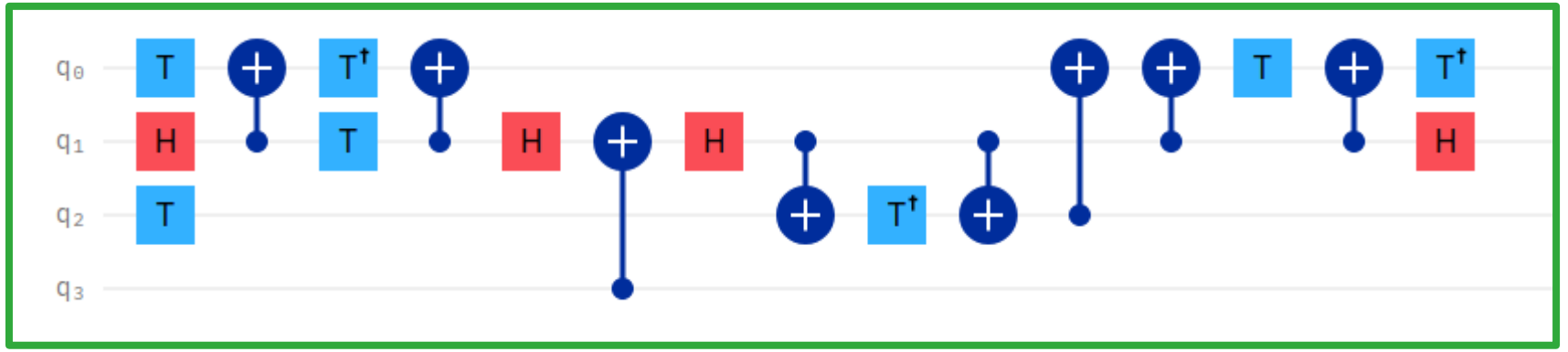
➢ In particular, what does **this quantum circuit** do?



QUANTUM COMPUTING

Ideal gate-based quantum computing

- In particular, what does **this quantum circuit** do?



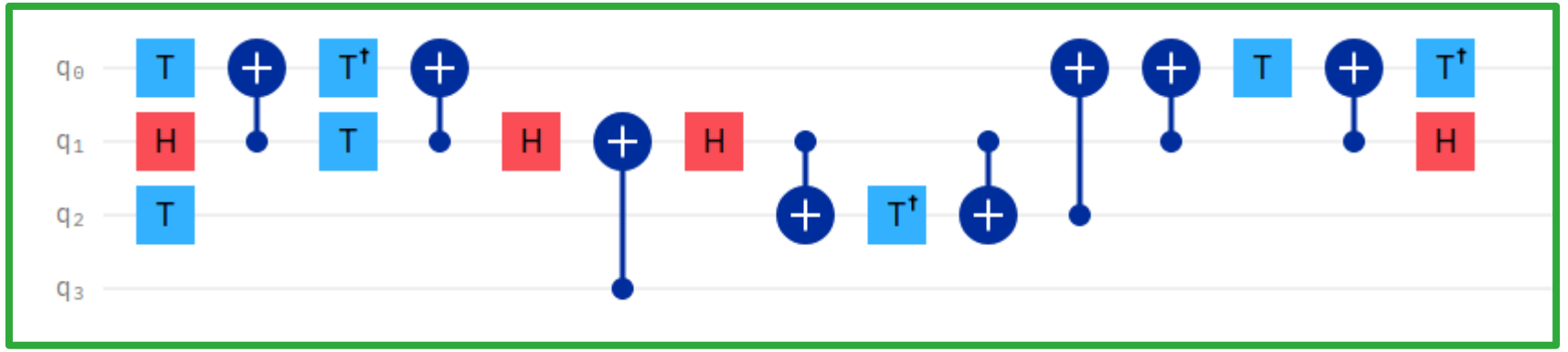
- 2-qubit adder

$$|q_3 q_2\rangle|q_1 q_0\rangle \mapsto |q_3 q_2\rangle|q_3 q_2 + q_1 q_0\rangle$$

QUANTUM COMPUTING

Ideal gate-based quantum computing

- In particular, what does **this quantum circuit** do?



- 2-qubit adder

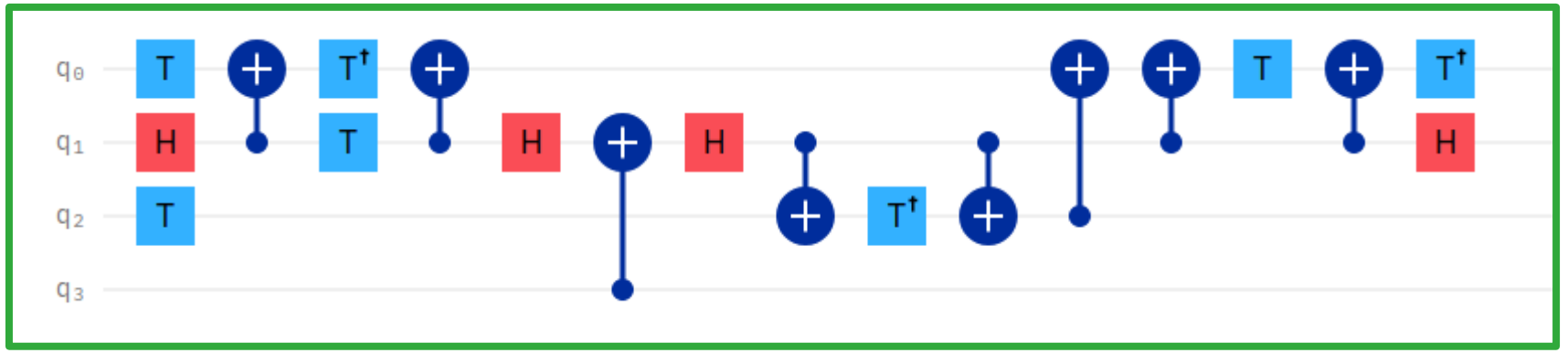
$$|q_3 q_2\rangle|q_1 q_0\rangle \mapsto |q_3 q_2\rangle|q_3 q_2 + q_1 q_0\rangle$$

  - e.g. $|2\rangle|1\rangle \mapsto |2\rangle|3\rangle$

QUANTUM COMPUTING

Ideal gate-based quantum computing

- In particular, what does **this quantum circuit** do?



- 2-qubit adder

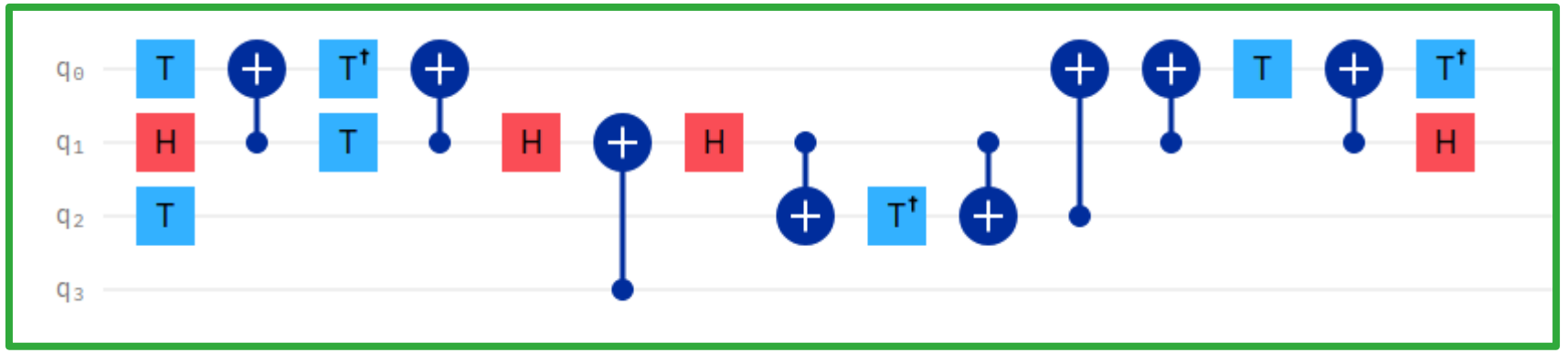
$$|q_3 q_2\rangle|q_1 q_0\rangle \mapsto |q_3 q_2\rangle|q_3 q_2 + q_1 q_0\rangle$$

  - e.g. $|2\rangle|1\rangle \mapsto |2\rangle|3\rangle$
  - but also **superpositions**:

QUANTUM COMPUTING

Ideal gate-based quantum computing

- In particular, what does **this quantum circuit** do?



- 2-qubit adder

$$|q_3 q_2\rangle|q_1 q_0\rangle \mapsto |q_3 q_2\rangle|q_3 q_2 + q_1 q_0\rangle$$

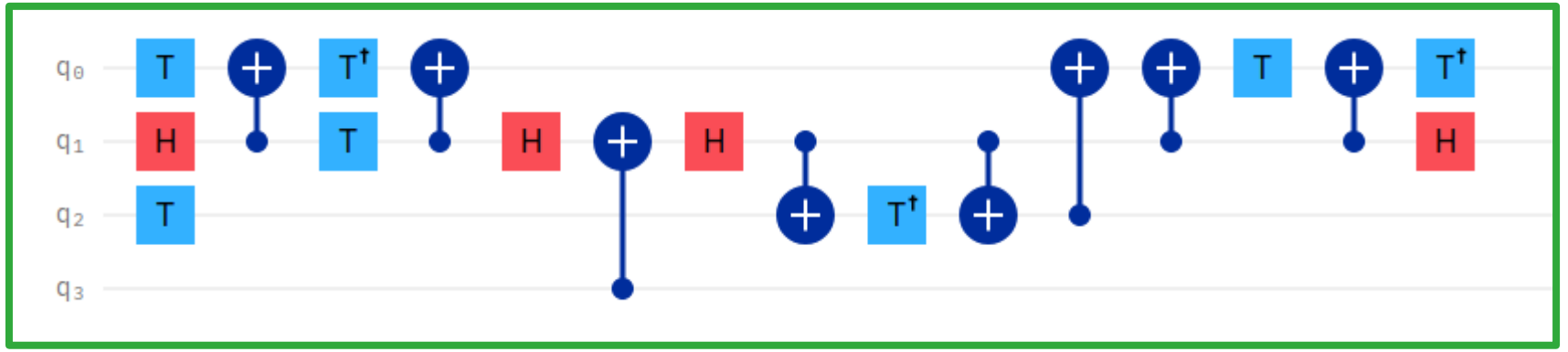
  - e.g. $|2\rangle|1\rangle \mapsto |2\rangle|3\rangle$
  - but also **superpositions**:

$$|2\rangle \frac{|0\rangle + |1\rangle + |2\rangle}{\sqrt{3}} \mapsto |2\rangle \frac{|2\rangle + |3\rangle + |0\rangle}{\sqrt{3}}$$

QUANTUM COMPUTING

Ideal gate-based quantum computing

- In particular, what does **this quantum circuit** do?



- 2-qubit adder

$$|q_3 q_2\rangle|q_1 q_0\rangle \mapsto |q_3 q_2\rangle|q_3 q_2 + q_1 q_0\rangle$$

  - e.g. $|2\rangle|1\rangle \mapsto |2\rangle|3\rangle$
  - but also **superpositions**:

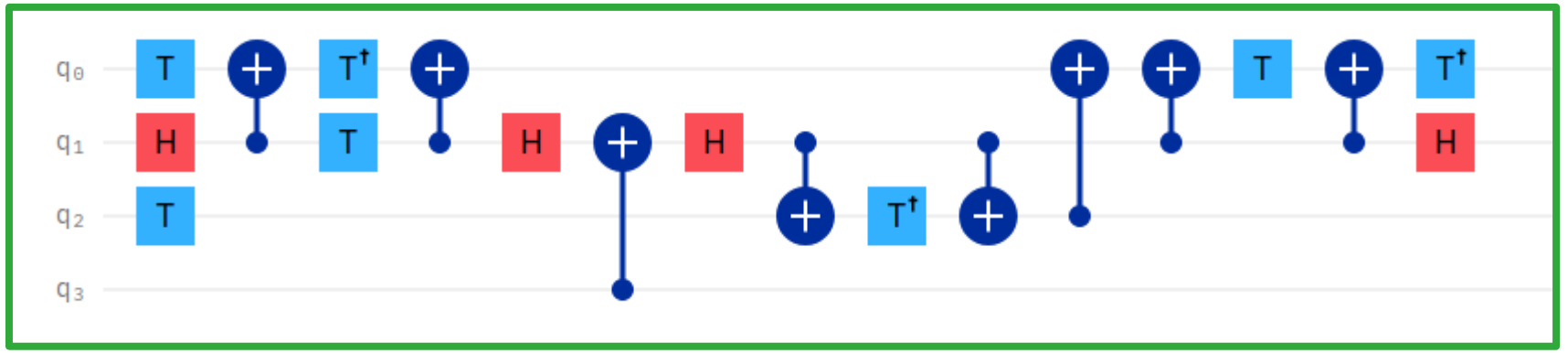
$$|2\rangle \frac{|0\rangle + |1\rangle + |2\rangle}{\sqrt{3}} \mapsto |2\rangle \frac{|2\rangle + |3\rangle + |0\rangle}{\sqrt{3}}$$

$$\begin{pmatrix} \vdots \\ \psi_{1000} = 1/\sqrt{3} \\ \psi_{1001} = 1/\sqrt{3} \\ \psi_{1010} = 1/\sqrt{3} \\ \psi_{1011} = 0 \\ \vdots \end{pmatrix} \mapsto \begin{pmatrix} \vdots \\ \psi_{1000} = 1/\sqrt{3} \\ \psi_{1001} = 0 \\ \psi_{1010} = 1/\sqrt{3} \\ \psi_{1011} = 1/\sqrt{3} \\ \vdots \end{pmatrix}$$

QUANTUM COMPUTING

Ideal gate-based quantum computing

- In particular, what does **this quantum circuit** do?



- 2-qubit adder

$$|q_3 q_2\rangle|q_1 q_0\rangle \mapsto |q_3 q_2\rangle|q_3 q_2 + q_1 q_0\rangle$$

  - e.g. $|2\rangle|1\rangle \mapsto |2\rangle|3\rangle$
  - but also **superpositions**:

$$|2\rangle \frac{|0\rangle + |1\rangle + |2\rangle}{\sqrt{3}} \mapsto |2\rangle \frac{|2\rangle + |3\rangle + |0\rangle}{\sqrt{3}}$$

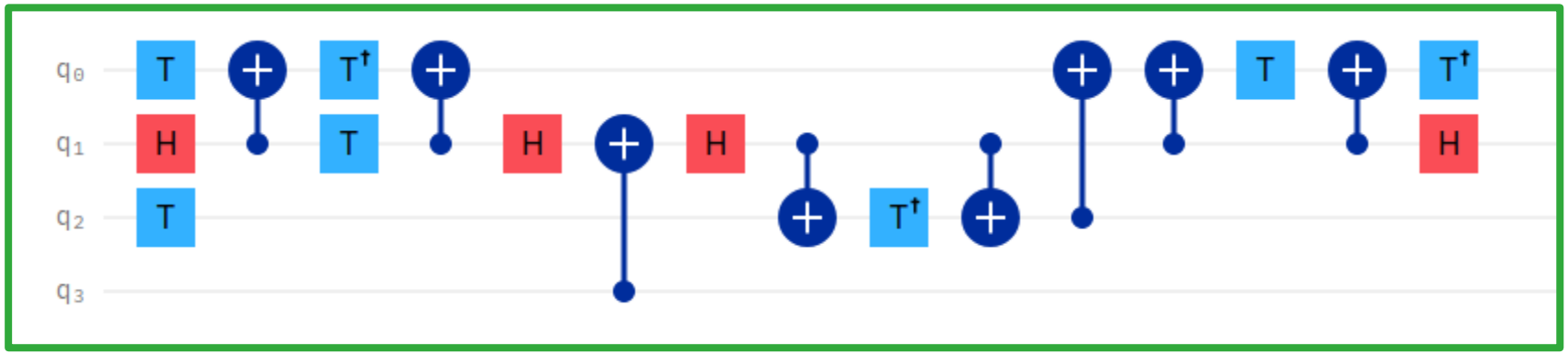
$$\begin{pmatrix} \vdots \\ \psi_{1000} = 1/\sqrt{3} \\ \psi_{1001} = 1/\sqrt{3} \\ \psi_{1010} = 1/\sqrt{3} \\ \psi_{1011} = 0 \\ \vdots \end{pmatrix} \mapsto \begin{pmatrix} \vdots \\ \psi_{1000} = 1/\sqrt{3} \\ \psi_{1001} = 0 \\ \psi_{1010} = 1/\sqrt{3} \\ \psi_{1011} = 1/\sqrt{3} \\ \vdots \end{pmatrix}$$

- Why? → Simulate with JUQCS

JUQCS

Jülich universal quantum computer simulator

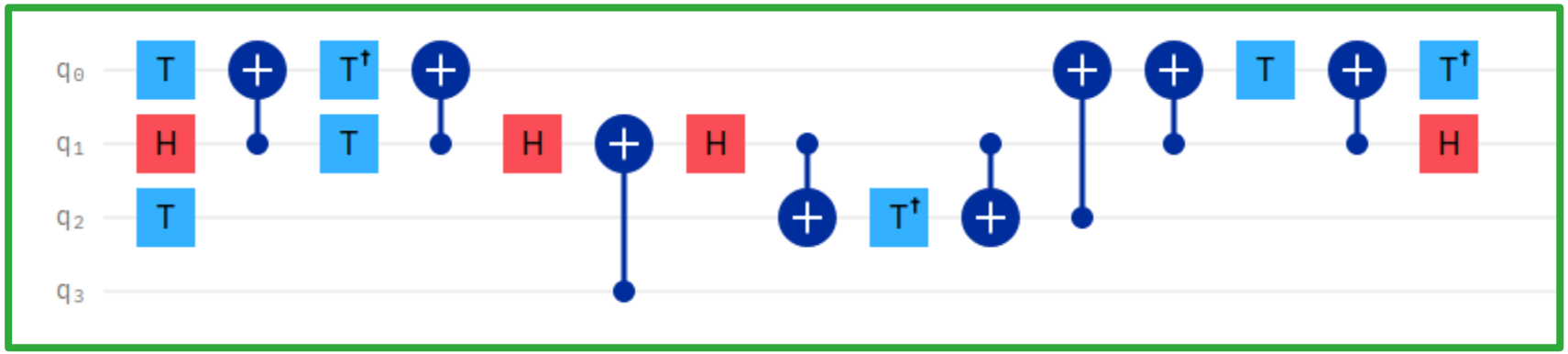
- What does a quantum computer **simulator** do?
  - It runs a quantum circuit



JUQCS

Jülich universal quantum computer simulator

- What does a quantum computer **simulator** do?
  - It runs a quantum circuit



- What does this mean, actually?
  - It performs **matrix-vector multiplications** that are

sparse **complex** **unitary**

  - with **huge** vectors and **huge²** matrices

JUQCS

Distribution of the quantum state

How does the simulator manage all these complex numbers?

JUQCS

Distribution of the quantum state

How does the simulator manage all these complex numbers?

→ Distribute quantum state $|\psi\rangle = (\psi_{\ldots q_2 q_1 q_0})$ over multiple compute nodes

JUQCS

Distribution of the quantum state

How does the simulator manage all these complex numbers?

→ Distribute quantum state $|\psi\rangle = (\psi_{\dots q_2 q_1 q_0})$ over multiple compute nodes

For 40 qubits: 2^{40} $\psi'\mathrm{s}$ = 16 TiB complex numbers = 64 GiB per node with 256 nodes

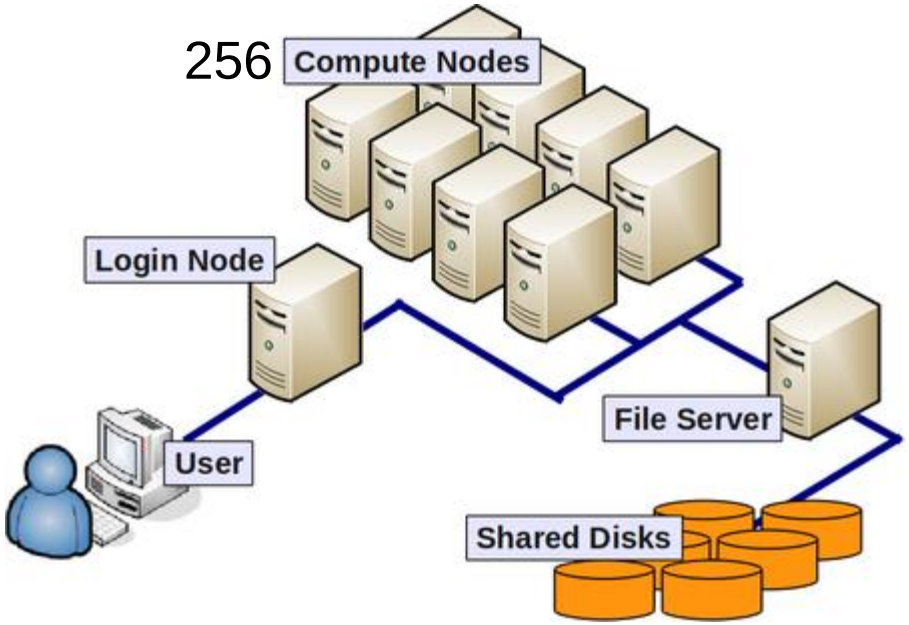
JUQCS

Distribution of the quantum state

How does the simulator manage all these complex numbers?

→ Distribute quantum state $|\psi\rangle = (\psi_{\ldots q_2 q_1 q_0})$ over multiple compute nodes

For 40 qubits: $2^{40}\ \psi'\mathrm{s}$ = 16 TiB complex numbers = 64 GiB per node with 256 nodes



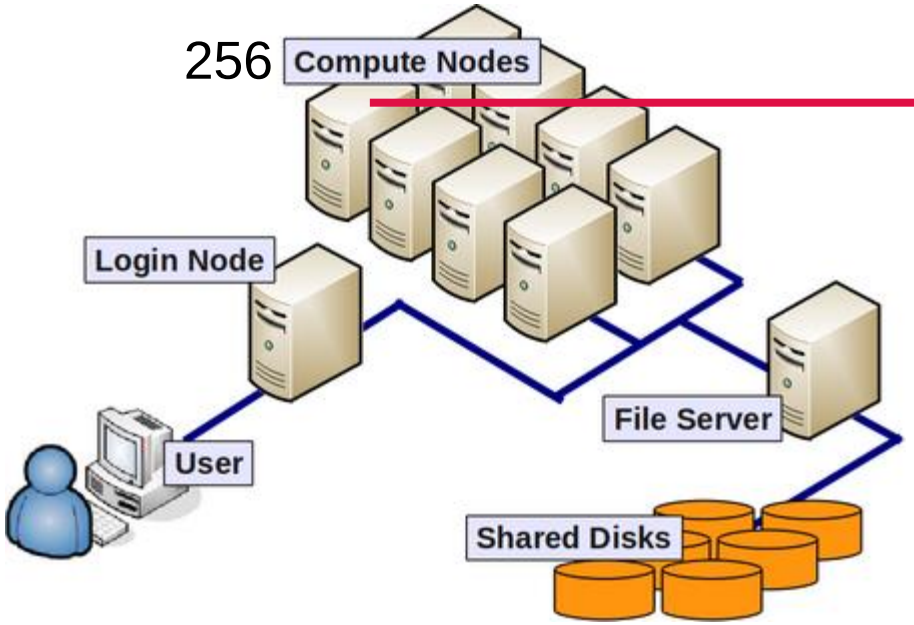
JUQCS

Distribution of the quantum state

How does the simulator manage all these complex numbers?

→ Distribute quantum state $|\psi\rangle = (\psi_{\ldots q_2 q_1 q_0})$ over multiple compute nodes

For 40 qubits: $2^{40}\ \psi'\mathrm{s}$ = 16 TiB complex numbers = 64 GiB per node with 256 nodes



MPI rank 0 = 0b00000000:

$$\begin{pmatrix} \psi_{00000000\ 0\cdots0} \\ \vdots \\ \psi_{00000000\ 1\cdots1} \end{pmatrix}$$

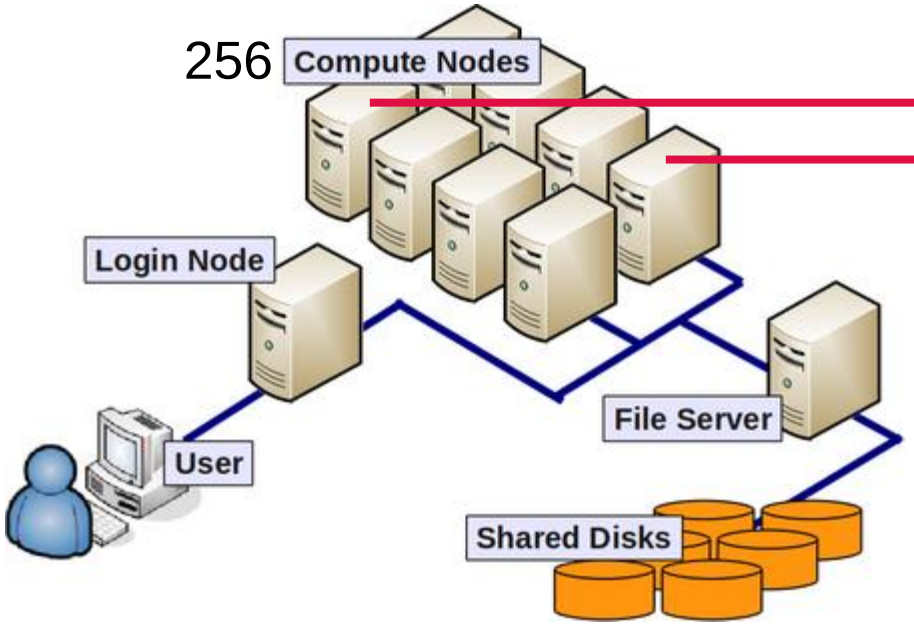
JUQCS

Distribution of the quantum state

How does the simulator manage all these complex numbers?

→ Distribute quantum state $|\psi\rangle = (\psi_{\ldots q_2 q_1 q_0})$ over multiple compute nodes

For 40 qubits: $2^{40}\ \psi'\mathrm{s}$ = 16 TiB complex numbers = 64 GiB per node with 256 nodes



MPI rank 0 = 0b00000000:

$$\begin{pmatrix} \psi_{00000000\,0\cdots0} \\ \vdots \\ \psi_{00000000\,1\cdots1} \end{pmatrix}$$

MPI rank 255 = 0b11111111:

$$\begin{pmatrix} \psi_{11111111\,0\cdots0} \\ \vdots \\ \psi_{11111111\,1\cdots1} \end{pmatrix}$$

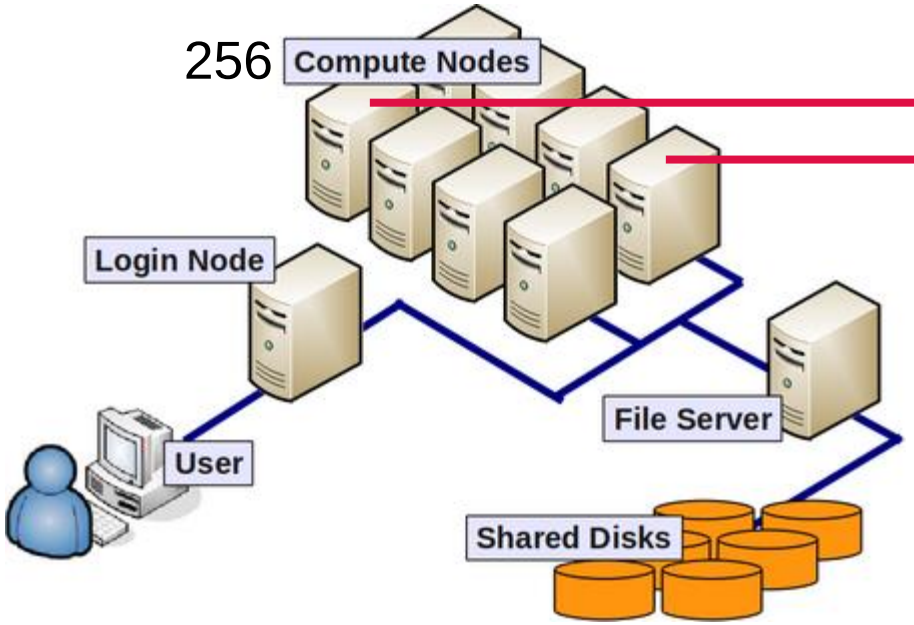
JUQCS

Distribution of the quantum state

How does the simulator manage all these complex numbers?

→ Distribute quantum state $|\psi\rangle = (\psi_{\ldots q_2 q_1 q_0})$ over multiple compute nodes

For 40 qubits: $2^{40}\ \psi'\mathrm{s}$ = 16 TiB complex numbers = 64 GiB per node with 256 nodes



MPI rank 0 = 0b00000000:

$$\begin{pmatrix} \psi_{00000000\ 0\cdots0} \\ \vdots \\ \psi_{00000000\ 1\cdots1} \end{pmatrix}$$

MPI rank 255 = 0b11111111:

$$\begin{pmatrix} \psi_{11111111\ 0\cdots0} \\ \vdots \\ \psi_{11111111\ 1\cdots1} \end{pmatrix}$$

8 **global** qubits

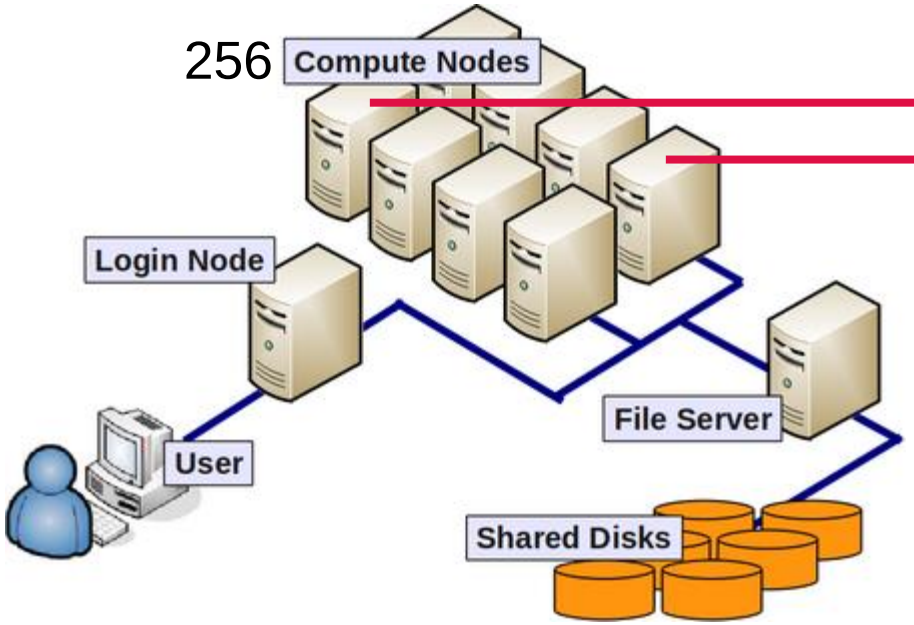
JUQCS

Distribution of the quantum state

How does the simulator manage all these complex numbers?

→ Distribute quantum state $|\psi\rangle = (\psi_{\ldots q_2 q_1 q_0})$ over multiple compute nodes

For 40 qubits: $2^{40}\ \psi'\mathrm{s}$ = 16 TiB complex numbers = 64 GiB per node with 256 nodes



MPI rank 0 = 0b00000000:

$$\begin{pmatrix} \psi_{00000000\ 0\cdots0} \\ \vdots \\ \psi_{00000000\ 1\cdots1} \end{pmatrix}$$

MPI rank 255 = 0b11111111:

$$\begin{pmatrix} \psi_{11111111\ 0\cdots0} \\ \vdots \\ \psi_{11111111\ 1\cdots1} \end{pmatrix}$$

8 **global** qubits
32 **local** qubits

JUQCS-G

Simulating quantum computers on GPUs CUDA MPI Fortran

- Distribute quantum state on GPUs (NVIDIA A100: 40GB per GPU)

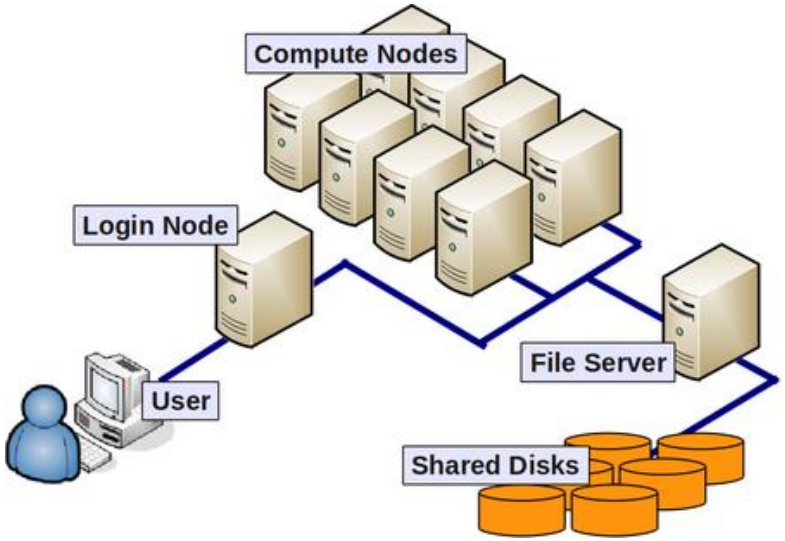
  For 40 qubits: $2^{40}\ \psi's$ = 16 TiB complex numbers = 16 GiB per GPU with 4*256 GPUs

JUQCS-G

Simulating quantum computers on GPUs CUDA MPI Fortran

- Distribute quantum state on GPUs (NVIDIA A100: 40GB per GPU)

For 40 qubits: $2^{40}\ \psi's$ = 16 TiB complex numbers = 16 GiB per GPU with 4*256 GPUs

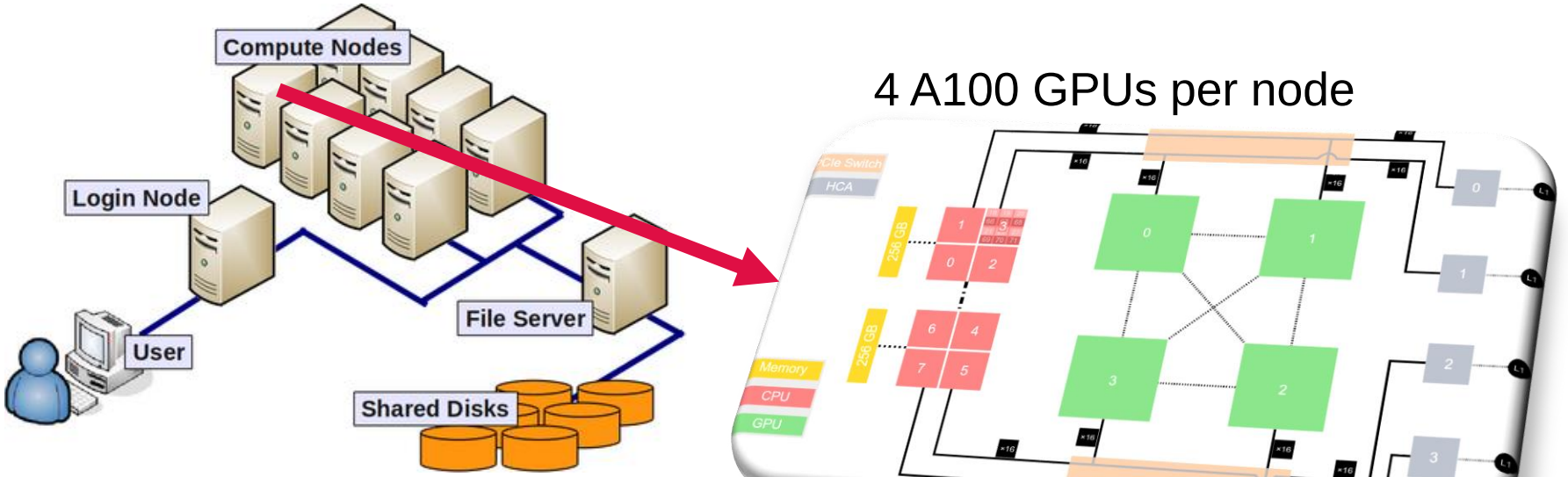


JUQCS-G

Simulating quantum computers on GPUs CUDA MPI Fortran

- Distribute quantum state on GPUs (NVIDIA A100: 40GB per GPU)

For 40 qubits: $2^{40}\ \psi's$ = 16 TiB complex numbers = 16 GiB per GPU with 4*256 GPUs



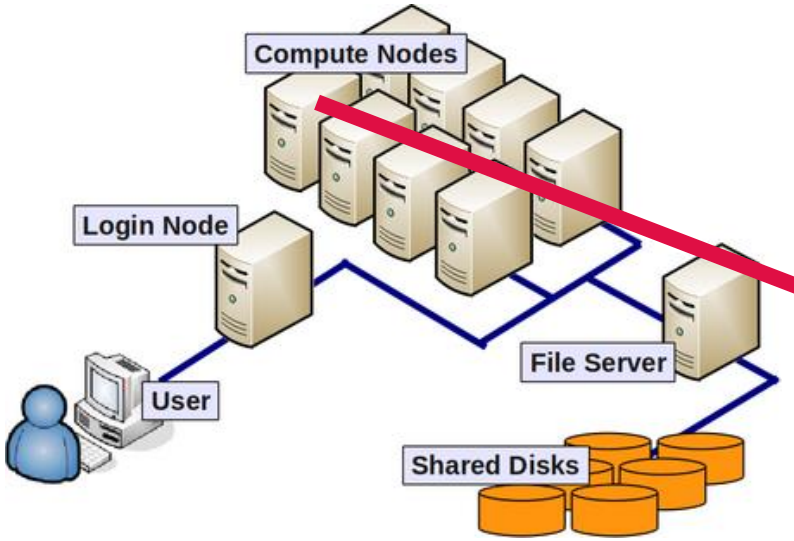
4 A100 GPUs per node

JUQCS-G

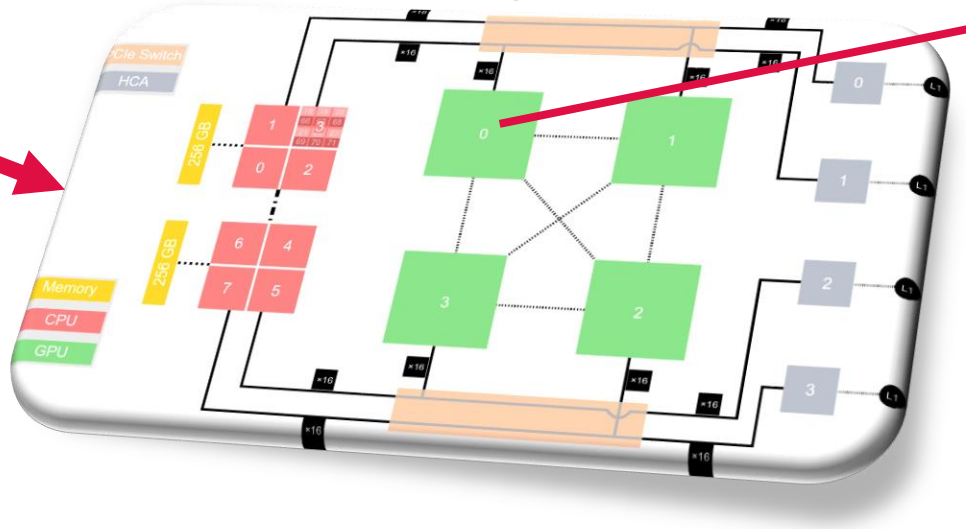
Simulating quantum computers on GPUs CUDA MPI Fortran

- Distribute quantum state on GPUs (NVIDIA A100: 40GB per GPU)

For 40 qubits: $2^{40}\ \psi's$ = 16 TiB complex numbers = 16 GiB per GPU with 4*256 GPUs



4 A100 GPUs per node



GPU rank 0 = 0b0000000000:

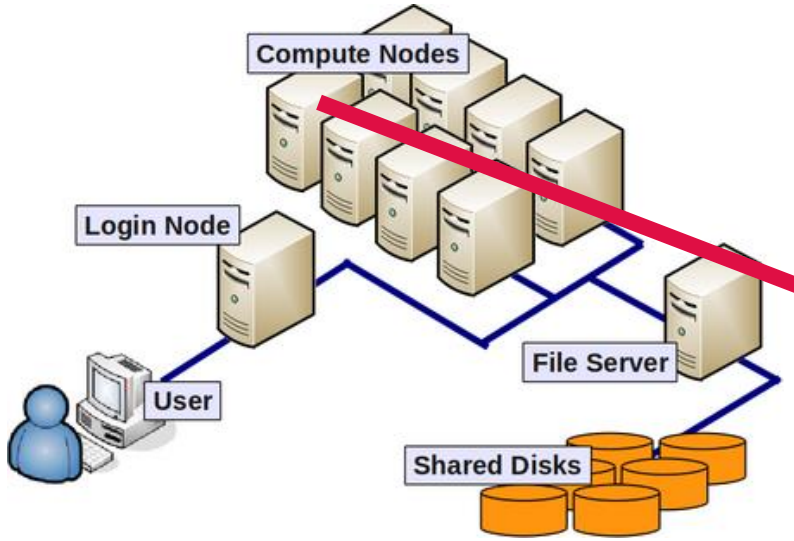
$$\begin{pmatrix} \psi_{0000000000\,0\cdots0} \\ \vdots \\ \psi_{0000000000\,1\cdots1} \end{pmatrix}$$

JUQCS-G

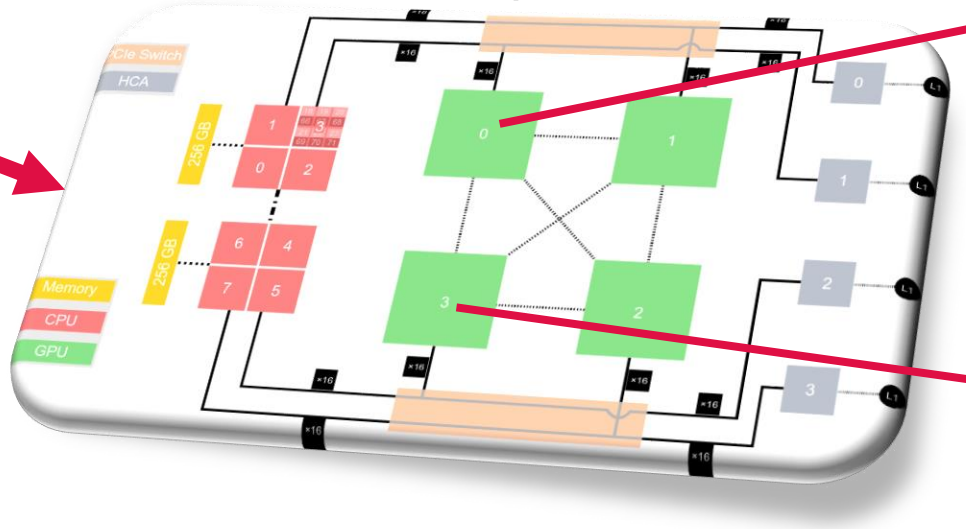
Simulating quantum computers on GPUs CUDA MPI Fortran

- Distribute quantum state on GPUs (NVIDIA A100: 40GB per GPU)

For 40 qubits: $2^{40}\ \psi's$ = 16 TiB complex numbers = 16 GiB per GPU with 4*256 GPUs



4 A100 GPUs per node



GPU rank 0 = 0b0000000000:

$$\begin{pmatrix} \psi_{0000000000\,0\cdots0} \\ \vdots \\ \psi_{0000000000\,1\cdots1} \end{pmatrix}$$

GPU rank 3 = 0b0000000011:

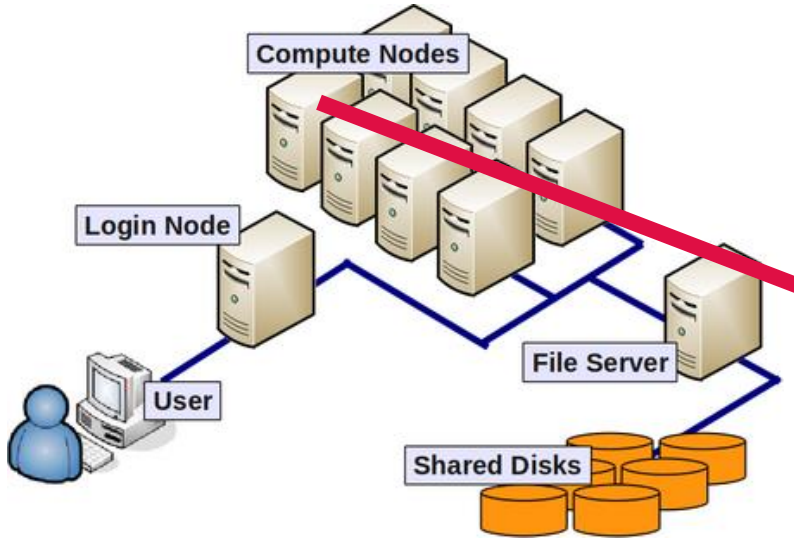
$$\begin{pmatrix} \psi_{0000000011\,0\cdots0} \\ \vdots \\ \psi_{0000000011\,1\cdots1} \end{pmatrix}$$

JUQCS-G

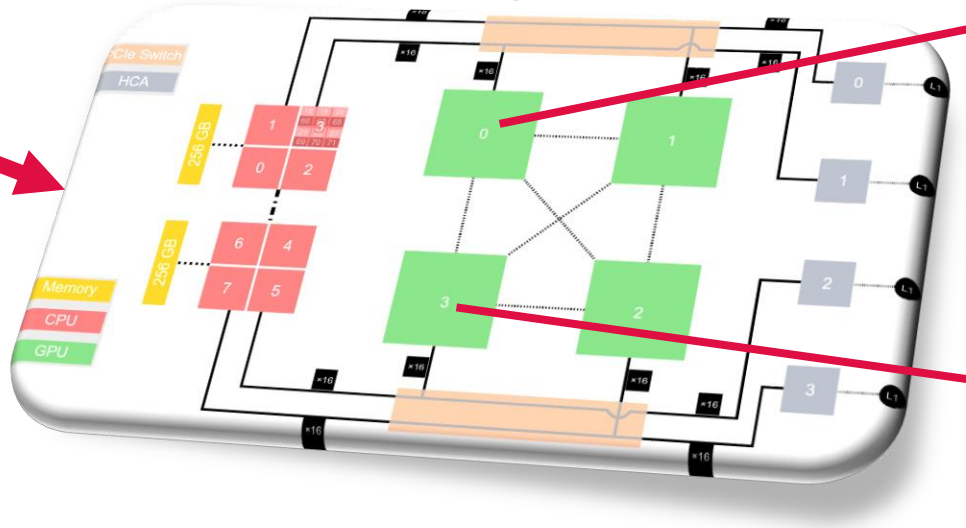
Simulating quantum computers on GPUs **CUDA MPI Fortran**

- Distribute quantum state on GPUs (NVIDIA A100: 40GB per GPU)

For 40 qubits: $2^{40}\ \psi's$ = 16 TiB complex numbers = 16 GiB per GPU with 4*256 GPUs



4 A100 GPUs per node



GPU rank 0 = 0b0000000000:

$$\begin{pmatrix} \psi_{0000000000\,0\cdots0} \\ \vdots \\ \psi_{0000000000\,1\cdots1} \end{pmatrix}$$

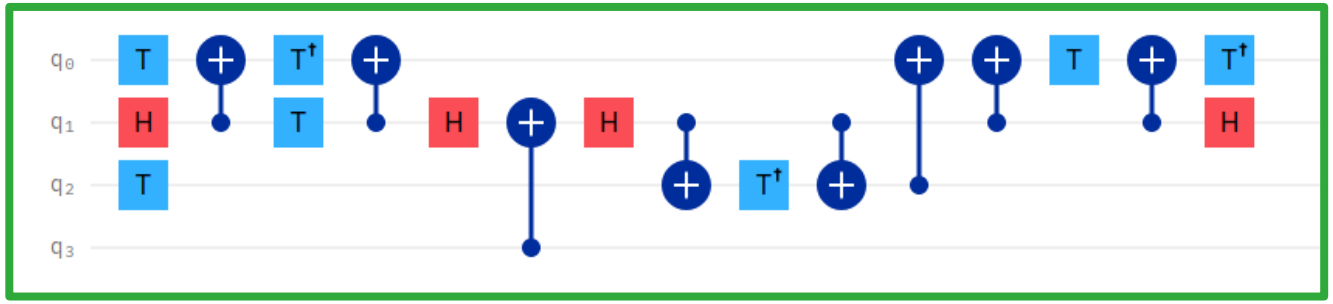
GPU rank 3 = 0b0000000011:

$$\begin{pmatrix} \psi_{0000000011\,0\cdots0} \\ \vdots \\ \psi_{0000000011\,1\cdots1} \end{pmatrix}$$

- The MPI communication scheme is the same

JUQCS-G

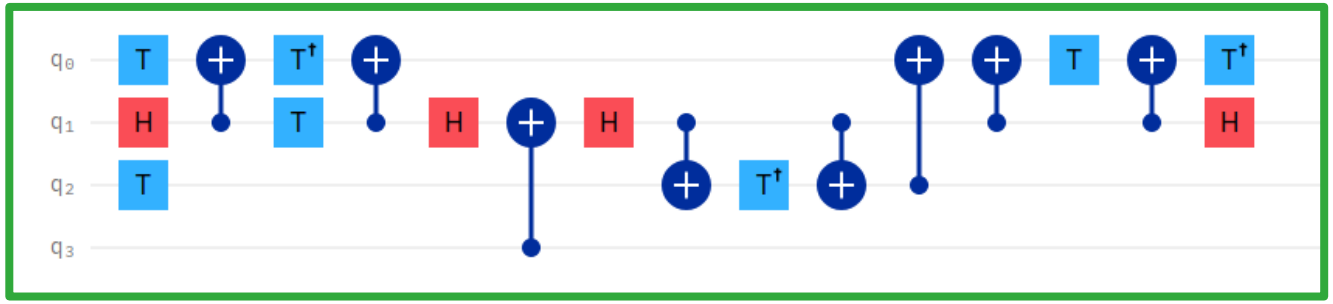
CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

JUQCS-G

CUDA implementation



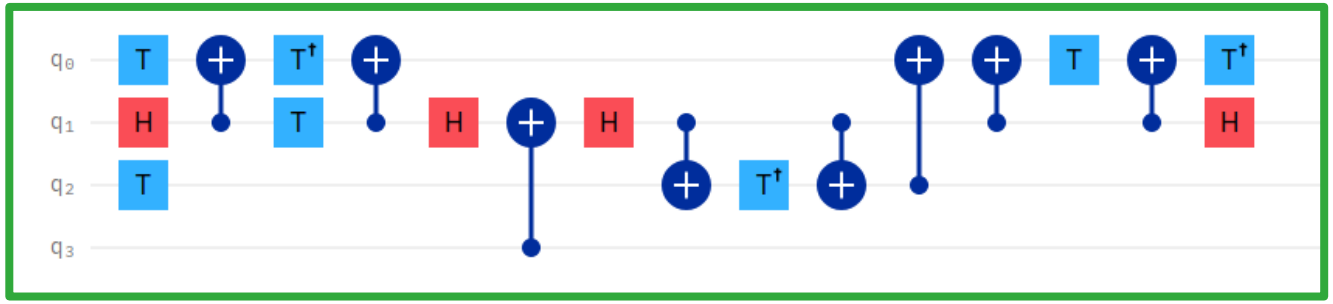
How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$$\begin{pmatrix} \psi_{00000000\,0\cdots0} \\ \vdots \\ \psi_{00000000\,1\cdots1} \\ \psi_{00000001\,0\cdots0} \\ \vdots \\ \psi_{00000001\,1\cdots1} \\ \vdots \\ \psi_{11111111\,0\cdots0} \\ \vdots \\ \psi_{11111111\,1\cdots1} \end{pmatrix}$$

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

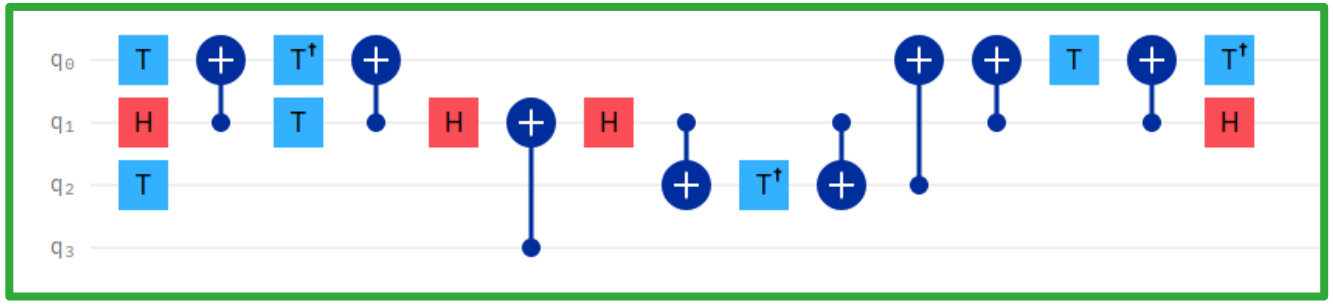
Full quantum state:

$$\begin{pmatrix} \psi_{00000000\,0\cdots0} \\ \vdots \\ \psi_{00000000\,1\cdots1} \\ \psi_{00000001\,0\cdots0} \\ \vdots \\ \psi_{00000001\,1\cdots1} \\ \vdots \\ \psi_{11111111\,0\cdots0} \\ \vdots \\ \psi_{11111111\,1\cdots1} \end{pmatrix}$$

GPU rank 0

JUQCS-G

CUDA implementation



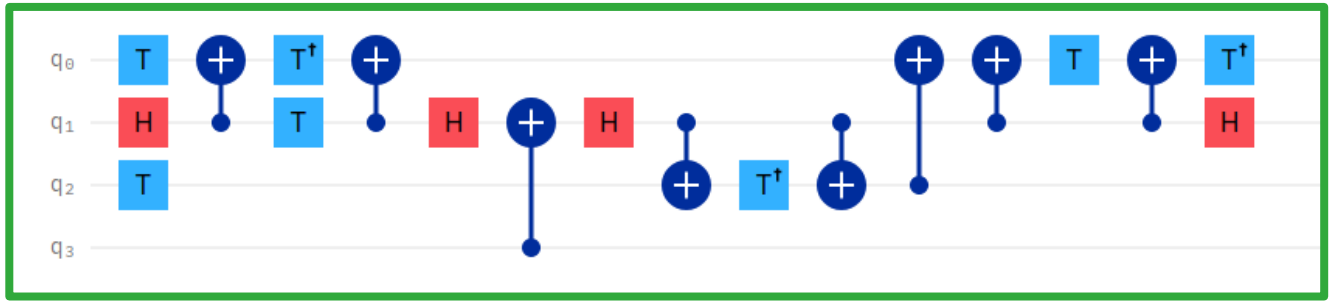
How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$$\begin{pmatrix} \psi_{00000000\,0\cdots0} \\ \text{GPU rank 0} \\ \psi_{00000000\,1\cdots1} \\ \psi_{00000001\,0\cdots0} \\ \text{GPU rank 1} \\ \psi_{00000001\,1\cdots1} \\ \vdots \\ \psi_{11111111\,0\cdots0} \\ \vdots \\ \psi_{11111111\,1\cdots1} \end{pmatrix}$$

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

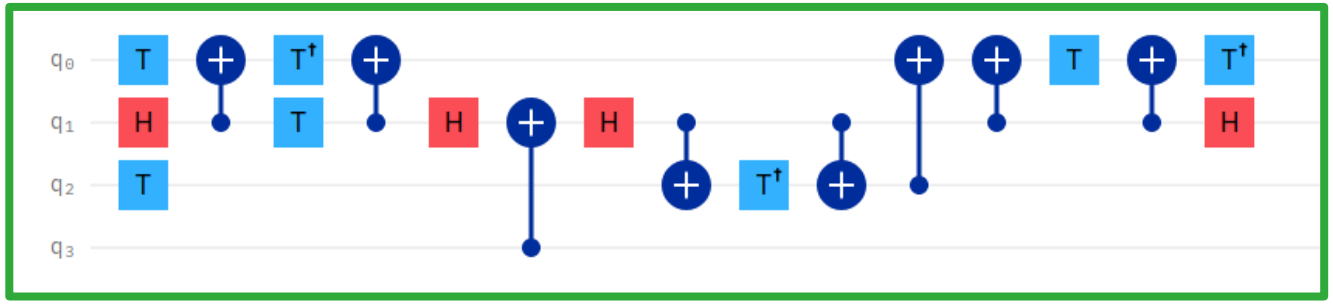
$\psi_{11111111\,0\cdots0}$

GPU rank 255

$\psi_{11111111\,1\cdots1}$

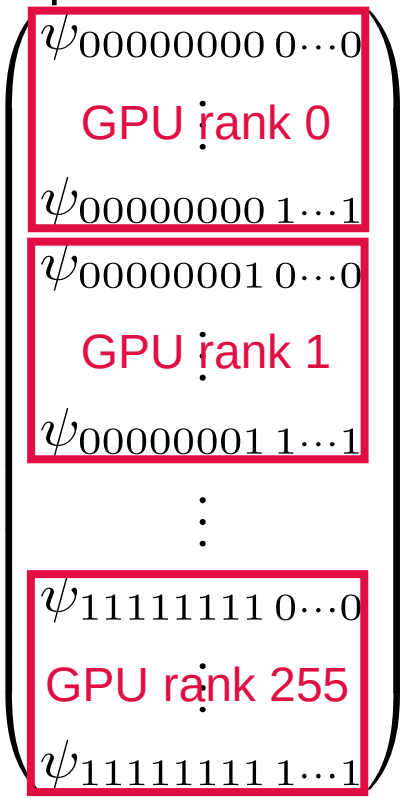
JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

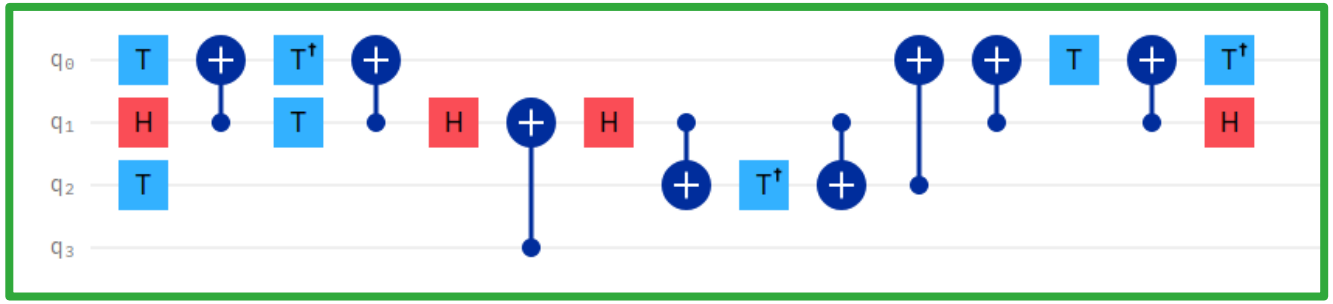
Full quantum state:



Quantum gate on **local** qubits:

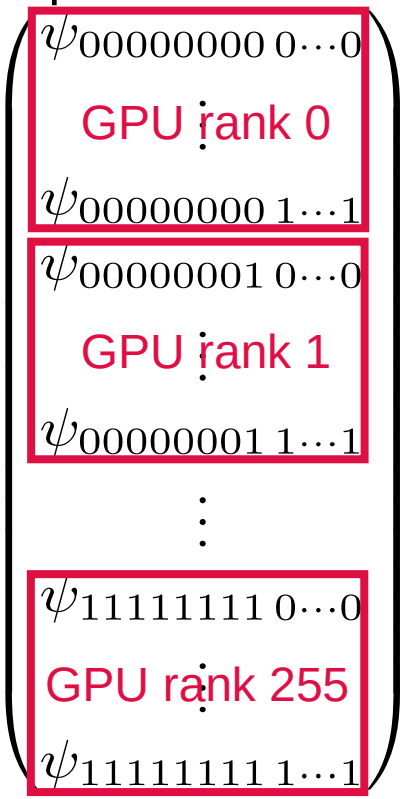
JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

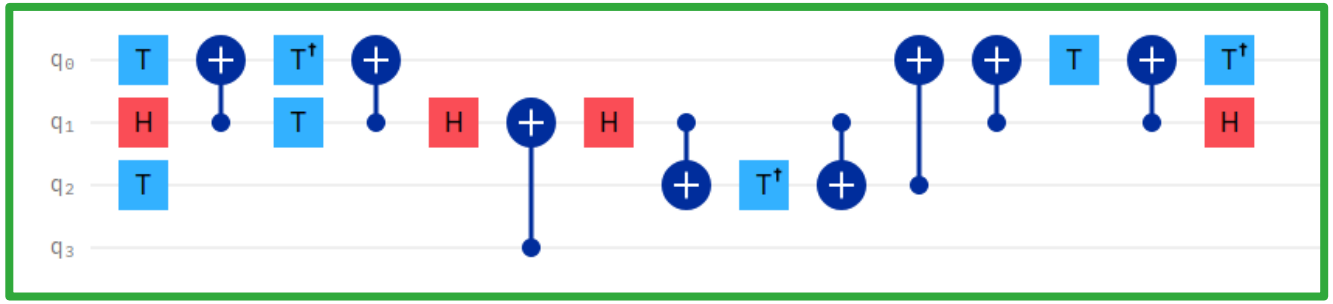


Quantum gate on **local** qubits:

e.g. H on qubit q_{30}

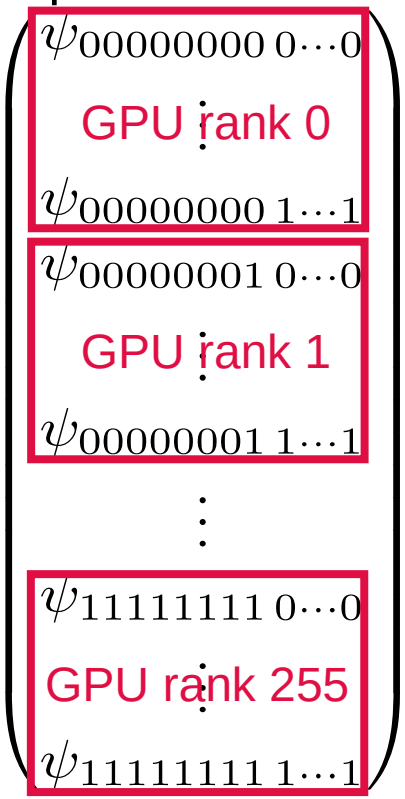
JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:



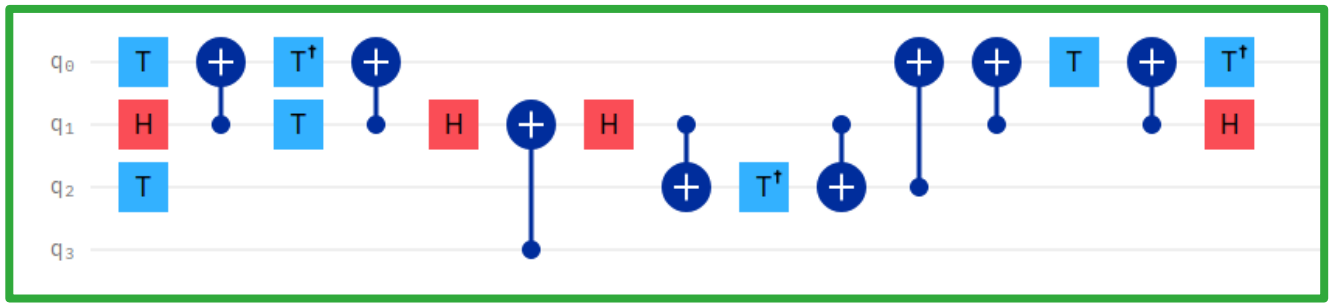
Quantum gate on **local** qubits:

e.g. H on qubit q_{30}

→ Each MPI rank r performs local 2x2 updates of the form

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$$\begin{pmatrix} \psi_{00000000\,0\cdots0} \\ \text{GPU rank 0} \\ \vdots \\ \psi_{00000000\,1\cdots1} \\ \psi_{00000001\,0\cdots0} \\ \text{GPU rank 1} \\ \vdots \\ \psi_{00000001\,1\cdots1} \\ \vdots \\ \psi_{11111111\,0\cdots0} \\ \text{GPU rank 255} \\ \vdots \\ \psi_{11111111\,1\cdots1} \end{pmatrix}$$

Quantum gate on **local** qubits:

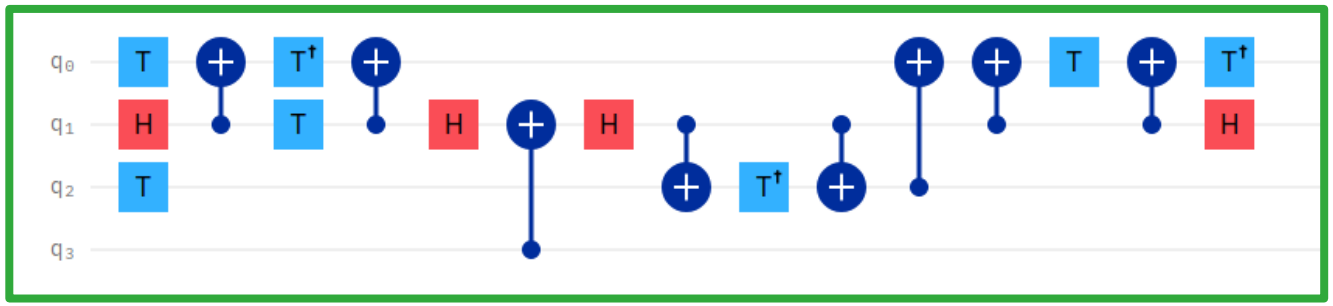
e.g. H on qubit q_{30}

→ Each MPI rank r performs local 2x2 updates of the form

$$\begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix} \leftarrow \mathrm{H} \begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix}$$

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$$\begin{pmatrix} \psi_{00000000\,0\cdots0} \\ \vdots \\ \psi_{00000000\,1\cdots1} \\ \psi_{00000001\,0\cdots0} \\ \vdots \\ \psi_{00000001\,1\cdots1} \\ \vdots \\ \psi_{11111111\,0\cdots0} \\ \vdots \\ \psi_{11111111\,1\cdots1} \end{pmatrix}$$

GPU rank 0

GPU rank 1

GPU rank 255

Quantum gate on **local** qubits:

e.g. H on qubit q_{30}

→ Each MPI rank r performs local 2x2 updates of the form

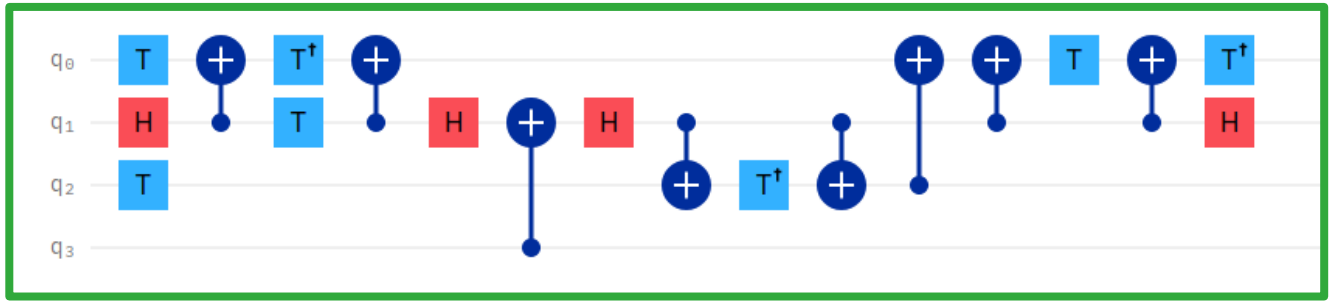
$$\begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix} \leftarrow H \begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix}$$

$$\uparrow$$

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

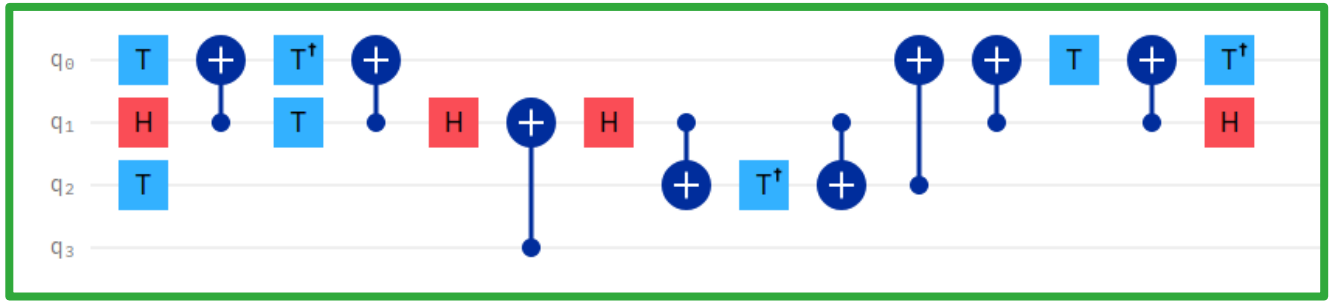
$$\begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix} \leftarrow \boxed{\mathrm{H}} \begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix}$$

$$\uparrow$$

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

$$\begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix} \leftarrow \boxed{H} \begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix}$$

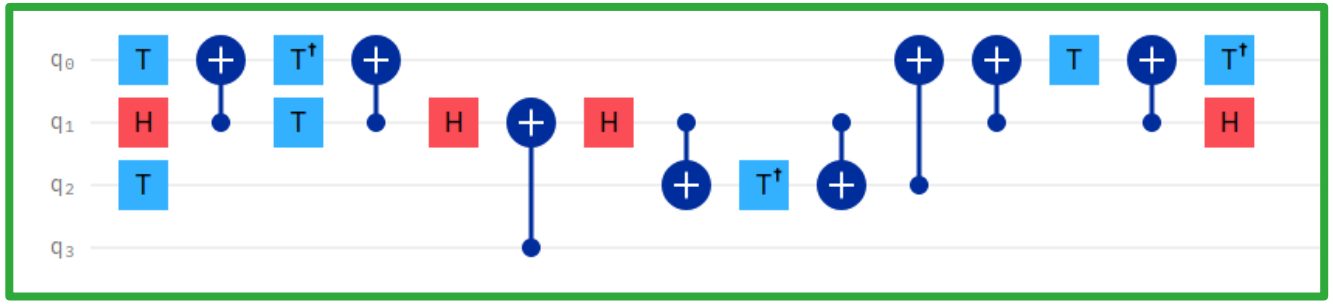
$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

```
attributes(global) subroutine H_operation_GPU(nstates,psi_R,psi_I,i0,i1,c)
        USE cudafor
        implicit real(kind=8) (a-h,o-z)
        integer(kind=8),parameter :: one=1
        integer(kind=8), value :: nstates,i1
        integer(kind=8) :: m1,nm1,i,j,k
        integer(kind=4), value :: i0
        real(kind=8), value :: c
        real(kind=8),dimension(0:nstates-1),device:: psi_R,psi_I

        k=blockidx%x-1
        k=k*blockdim%x+threadidx%x-1
        m1=i1-1
        nm1=not(m1)
        i=ishft(iand(k,nm1),one)+iand(k,m1)
        j=i+i1
        r0=psi_R(i)
        r1=psi_I(i)
        r2=psi_R(j)
        r3=psi_I(j)
        psi_R(i)=(r0+r2)*c
        psi_I(i)=(r1+r3)*c
        psi_R(j)=(r0-r2)*c
        psi_I(j)=(r1-r3)*c
end subroutine
```

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

$$\begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix} \leftarrow \boxed{\text{H}} \begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix}$$

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

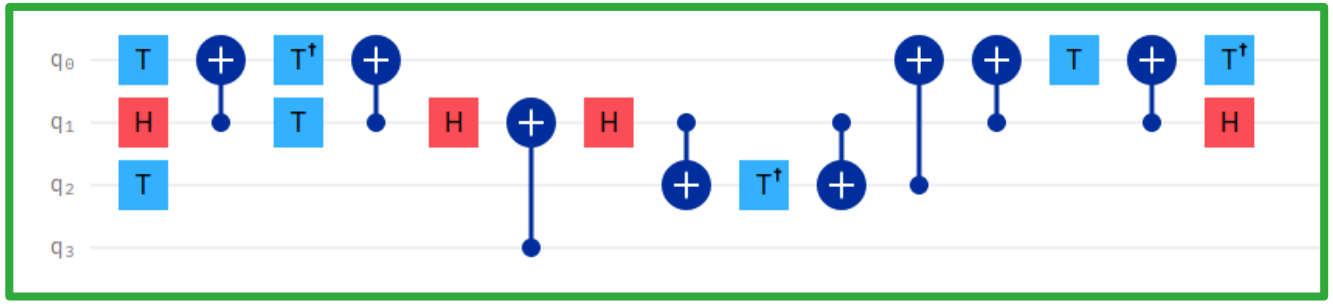
$$\begin{aligned} \texttt{i1} &= 00000000\,010\cdots0 \\ \texttt{k} &= \quad rrrrrrrr\ *\,*\cdots* \\ \texttt{i} &= rrrrrrrr\ *\,0\,*\cdots* \\ \texttt{j} &= rrrrrrrr\ *\,1\,*\cdots* \end{aligned}$$

```
attributes(global) subroutine H_operation_GPU(nstates,psi_R,psi_I,i0,i1,c)
        USE cudafor
        implicit real(kind=8) (a-h,o-z)
        integer(kind=8),parameter :: one=1
        integer(kind=8), value :: nstates,i1
        integer(kind=8) :: m1,nm1,i,j,k
        integer(kind=4), value :: i0
        real(kind=8), value :: c
        real(kind=8),dimension(0:nstates-1),device:: psi_R,psi_I

        k=blockidx%x-1
        k=k*blockdim%x+threadidx%x-1
        m1=i1-1
        nm1=not(m1)
        i=ishft(iand(k,nm1),one)+iand(k,m1)
        j=i+i1
        r0=psi_R(i)
        r1=psi_I(i)
        r2=psi_R(j)
        r3=psi_I(j)
        psi_R(i)=(r0+r2)*c
        psi_I(i)=(r1+r3)*c
        psi_R(j)=(r0-r2)*c
        psi_I(j)=(r1-r3)*c
end subroutine
```

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

$$\begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix} \leftarrow \boxed{\text{H}} \begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix}$$

$$\uparrow$$

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

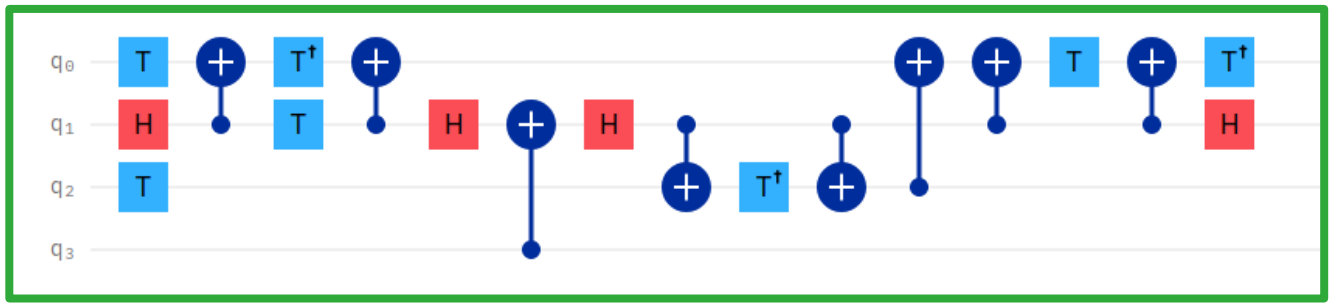
$$\begin{aligned} \texttt{i1} &= 00000000\,010\cdots0 \\ \texttt{k} &= \quad rrrrrrrr\ *\,*\cdots* \\ \texttt{i} &= rrrrrrrr\ *\,0*\cdots* \\ \texttt{j} &= rrrrrrrr\ *\,1*\cdots* \end{aligned}$$

```
attributes(global) subroutine H_operation_GPU(nstates,psi_R,psi_I,i0,i1,c)
        USE cudafor
        implicit real(kind=8) (a-h,o-z)
        integer(kind=8),parameter :: one=1
        integer(kind=8), value :: nstates,i1
        integer(kind=8) :: m1,nm1,i,j,k
        integer(kind=4), value :: i0
        real(kind=8), value :: c
        real(kind=8),dimension(0:nstates-1),device:: psi_R,psi_I

        k=blockidx%x-1
        k=k*blockdim%x+threadidx%x-1
        m1=i1-1
        nm1=not(m1)
        i=ishft(iand(k,nm1),one)+iand(k,m1)
        j=i+i1
        r0=psi_R(i)
        r1=psi_I(i)
        r2=psi_R(j)
        r3=psi_I(j)
        psi_R(i)=(r0+r2)*c
        psi_I(i)=(r1+r3)*c
        psi_R(j)=(r0-r2)*c
        psi_I(j)=(r1-r3)*c
end subroutine
```

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

$$\begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix} \leftarrow \boxed{\text{H}} \begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix}$$

$$\uparrow$$

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

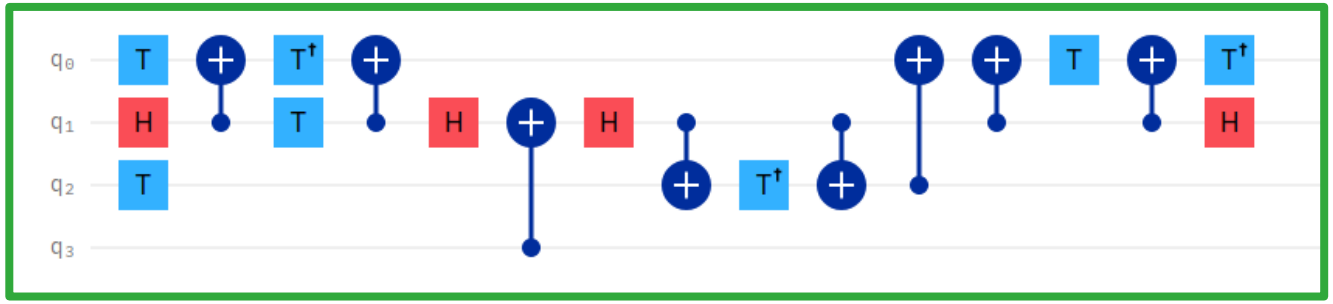
$$\begin{aligned} \texttt{i1} &= 00000000\,010\cdots0 \\ \texttt{k} &= \quad rrrrrrrr\ *\,*\cdots* \\ \texttt{i} &= rrrrrrrr\ *\,0\,*\cdots* \\ \texttt{j} &= rrrrrrrr\ *\,1\,*\cdots* \end{aligned}$$

```
attributes(global) subroutine H_operation_GPU(nstates,psi_R,psi_I,i0,i1,c)
        USE cudafor
        implicit real(kind=8) (a-h,o-z)
        integer(kind=8),parameter :: one=1
        integer(kind=8), value :: nstates,i1
        integer(kind=8) :: m1,nm1,i,j,k
        integer(kind=4), value :: i0
        real(kind=8), value :: c
        real(kind=8),dimension(0:nstates-1),device:: psi_R,psi_I

        k=blockidx%x-1
        k=k*blockdim%x+threadidx%x-1
        m1=i1-1
        nm1=not(m1)
        i=ishft(iand(k,nm1),one)+iand(k,m1)
        j=i+i1
        r0=psi_R(i)
        r1=psi_I(i)
        r2=psi_R(j)
        r3=psi_I(j)
        psi_R(i)=(r0+r2)*c
        psi_I(i)=(r1+r3)*c
        psi_R(j)=(r0-r2)*c
        psi_I(j)=(r1-r3)*c
end subroutine
```

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

$$\begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix} \leftarrow \boxed{\mathrm{H}} \begin{pmatrix} \psi_{rrrrrrrr\,*0*\cdots*} \\ \psi_{rrrrrrrr\,*1*\cdots*} \end{pmatrix}$$

$$\uparrow$$

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

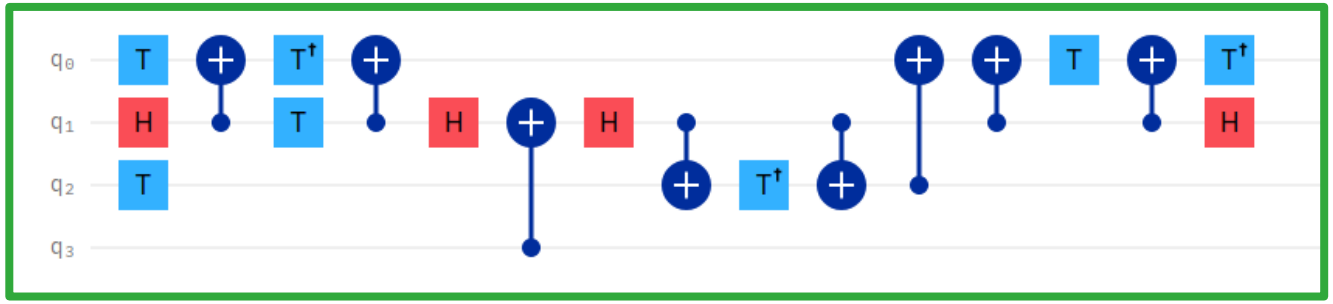
$$\mathtt{i1} = 00000000\,010\cdots0$$
$$\mathtt{k} = \quad rrrrrrrr\ *\,*\cdots*$$
$$\mathtt{i} = rrrrrrrr\ *\,0\,*\cdots*$$
$$\mathtt{j} = rrrrrrrr\ *\,1\,*\cdots*$$

```
attributes(global) subroutine H_operation_GPU(nstates,psi_R,psi_I,i0,i1,c)
        USE cudafor
        implicit real(kind=8) (a-h,o-z)
        integer(kind=8),parameter :: one=1
        integer(kind=8), value :: nstates,i1
        integer(kind=8) :: m1,nm1,i,j,k
        integer(kind=4), value :: i0
        real(kind=8), value :: c
        real(kind=8),dimension(0:nstates-1),device:: psi_R,psi_I

        k=blockidx%x-1
        k=k*blockdim%x+threadidx%x-1
        m1=i1-1
        nm1=not(m1)
        i=ishft(iand(k,nm1),one)+iand(k,m1)
        j=i+i1
        r0=psi_R(i)
        r1=psi_I(i)
        r2=psi_R(j)
        r3=psi_I(j)
        psi_R(i)=(r0+r2)*c
        psi_I(i)=(r1+r3)*c
        psi_R(j)=(r0-r2)*c
        psi_I(j)=(r1-r3)*c
end subroutine
```

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

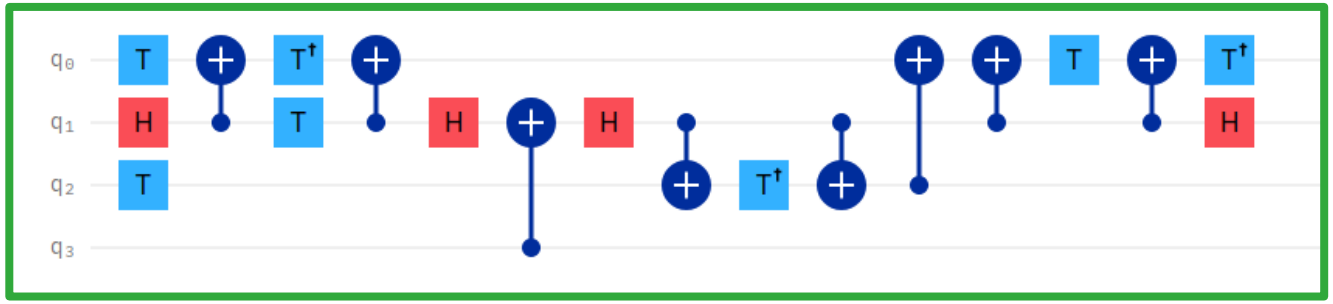
$\psi_{11111111\,0\cdots0}$

GPU rank 255

$\psi_{11111111\,1\cdots1}$

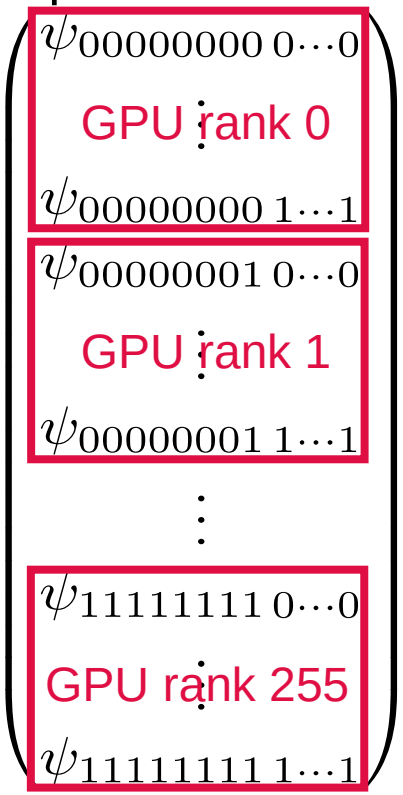
JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

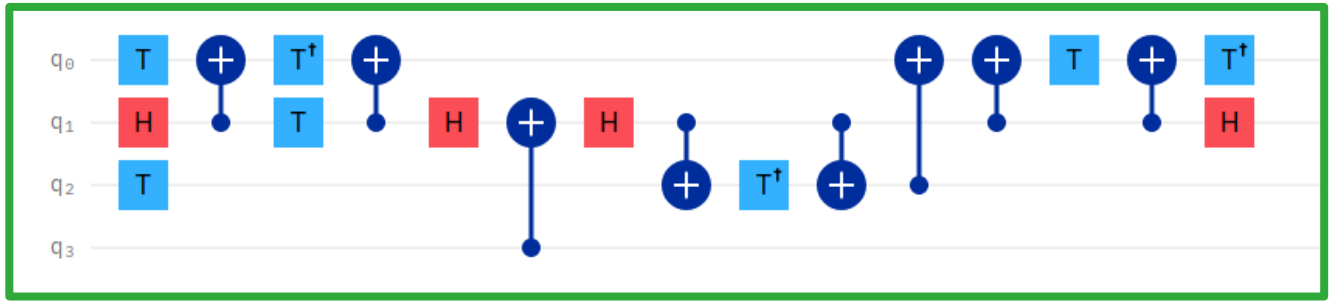
Full quantum state:



Quantum gate on **global** qubits:

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

$\psi_{11111111\,0\cdots0}$

GPU rank 255

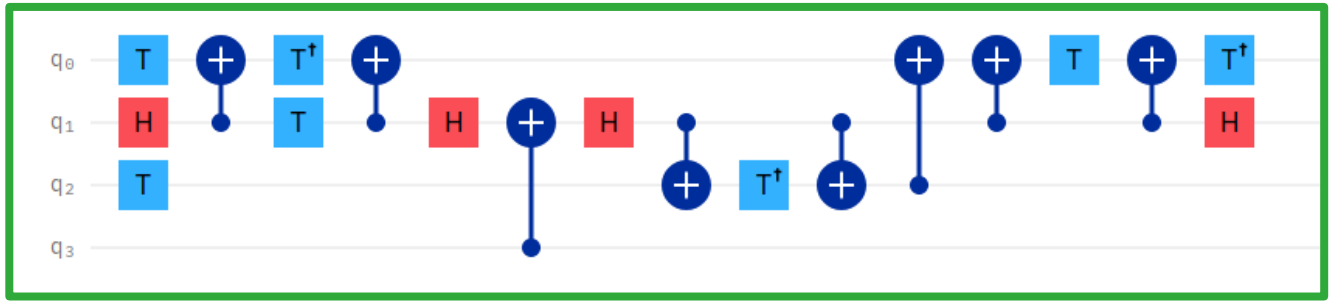
$\psi_{11111111\,1\cdots1}$

Quantum gate on **global** qubits:

e.g. H on qubit q_{32}

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

$\psi_{11111111\,0\cdots0}$

GPU rank 255

$\psi_{11111111\,1\cdots1}$

Quantum gate on **global** qubits:

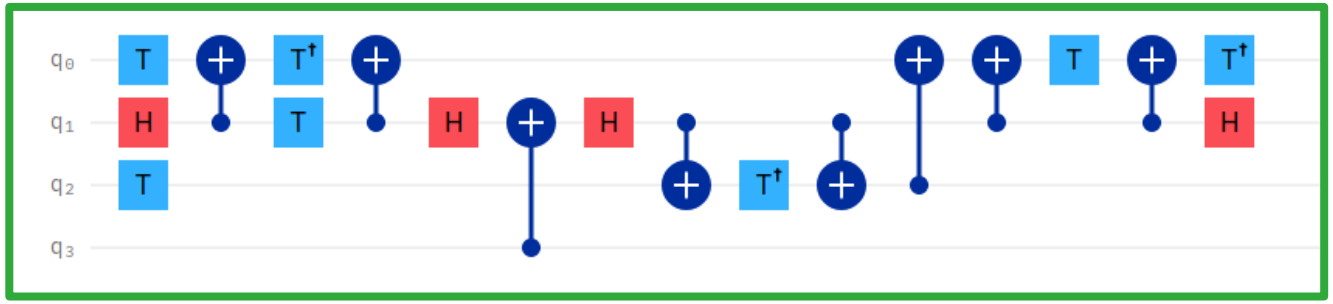
e.g. H on qubit q_{32}

- Need to perform 2x2 updates of the form

$$H \begin{pmatrix} \psi_{*******0\,*\cdots*} \\ \psi_{*******1\,*\cdots*} \end{pmatrix}$$

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

$\psi_{11111111\,0\cdots0}$

GPU rank 255

$\psi_{11111111\,1\cdots1}$

Quantum gate on **global** qubits:

e.g. H on qubit q_{32}

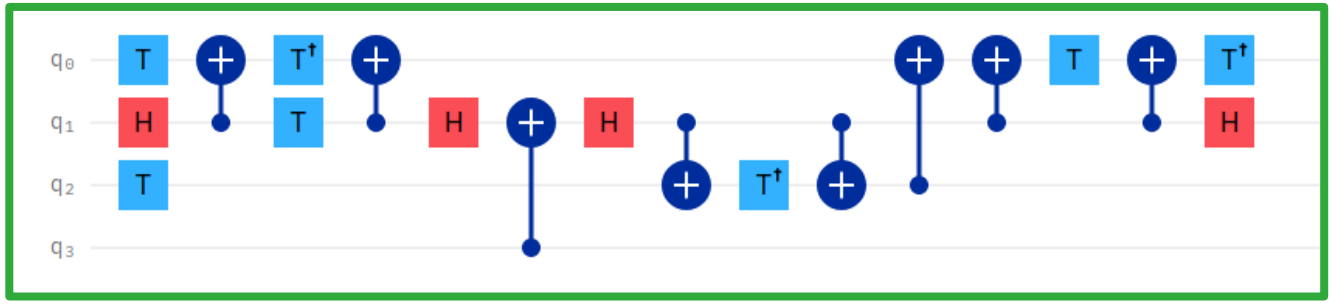
- Need to perform 2x2 updates of the form

$$H \begin{pmatrix} \psi_{*******0\,*\cdots*} \\ \psi_{*******1\,*\cdots*} \end{pmatrix}$$

- Problem: the numbers are on separate GPUs

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

$\psi_{11111111\,0\cdots0}$

GPU rank 255

$\psi_{11111111\,1\cdots1}$

Quantum gate on **global** qubits:

e.g. H on qubit q_{32}

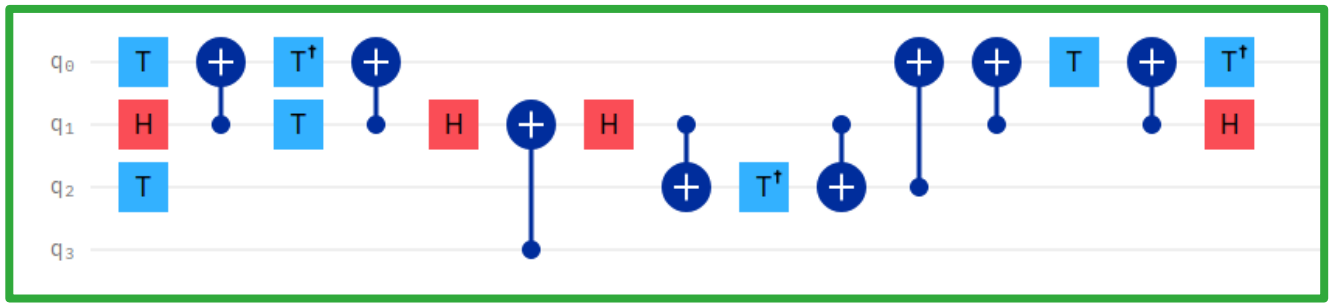
- Need to perform 2x2 updates of the form

$$\mathrm{H} \begin{pmatrix} \psi_{*******0\,*\cdots*} \\ \psi_{*******1\,*\cdots*} \end{pmatrix}$$

- Problem: the numbers are on separate GPUs
- Naïve solution:
  - Transfer $2^n/2$ ψ's (8 TiB), perform H, transfer back

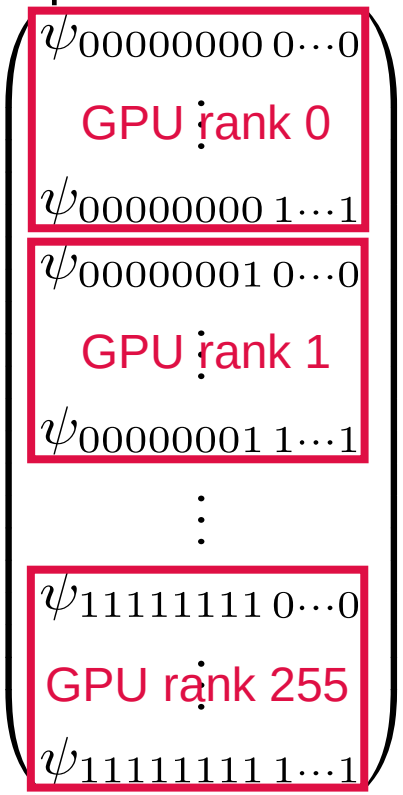
JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:



Quantum gate on **global** qubits:

e.g. H on qubit q_{32}

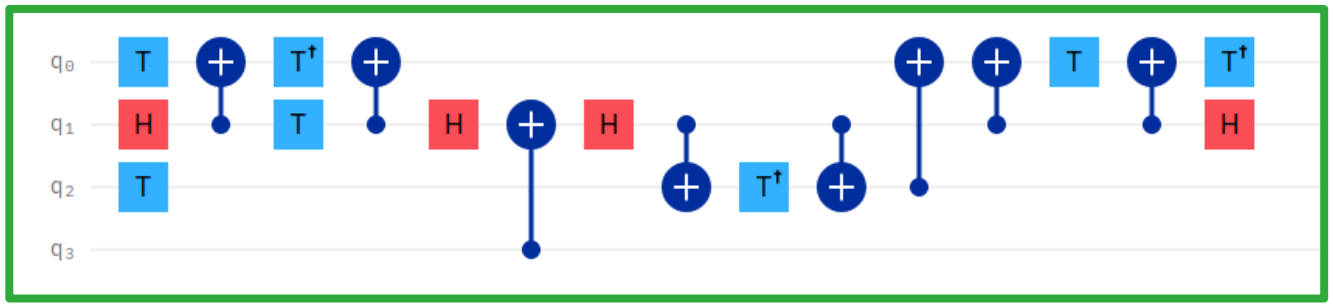
- Need to perform 2x2 updates of the form

$$\mathrm{H} \begin{pmatrix} \psi_{*******0\,*\cdots*} \\ \psi_{*******1\,*\cdots*} \end{pmatrix}$$

- Problem: the numbers are on separate GPUs
- Naïve solution:
  - Transfer $2^n/2$ ψ's (8 TiB), perform H, transfer back
- Optimal solution:
  - Exchange global and local qubit, e.g. $q_{32} \leftrightarrow q_0$

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

$\vdots$

$\psi_{11111111\,0\cdots0}$

GPU rank 255

$\psi_{11111111\,1\cdots1}$

Quantum gate on **global** qubits:

e.g. H on qubit q_{32}

- Need to perform 2x2 updates of the form

$$\mathrm{H} \begin{pmatrix} \psi_{*******0\,*\cdots*} \\ \psi_{*******1\,*\cdots*} \end{pmatrix}$$

- Problem: the numbers are on separate GPUs
- Naïve solution:
  - Transfer $2^n/2$ ψ's (8 TiB), perform H, transfer back
- Optimal solution:
  - Exchange global and local qubit, e.g. $q_{32} \leftrightarrow q_0$

$\psi_{0000000\,0\cdots0\,0}$

GPU rank 0

$\psi_{0000000\,1\cdots1\,0}$

$\psi_{0000000\,0\cdots0\,1}$

GPU rank 1

$\psi_{0000000\,1\cdots1\,1}$

$\vdots$

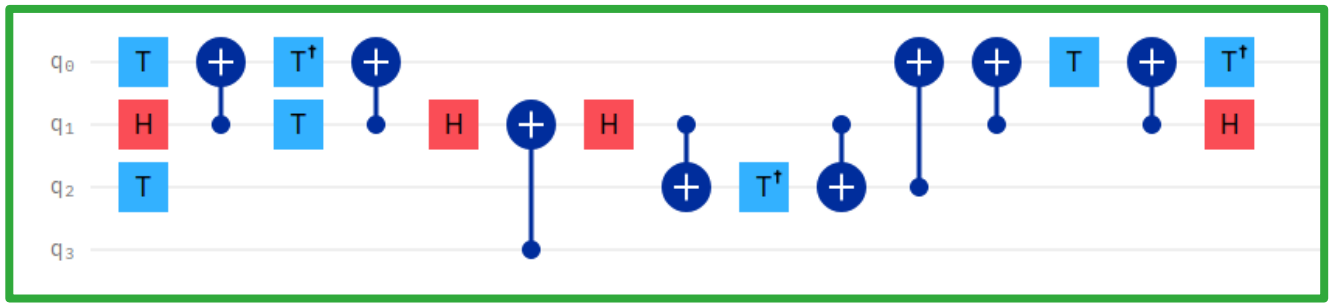
$\psi_{1111111\,0\cdots0\,1}$

GPU rank 255

$\psi_{1111111\,1\cdots1\,1}$

JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

$\vdots$

$\psi_{11111111\,0\cdots0}$

GPU rank 255

$\psi_{11111111\,1\cdots1}$

Quantum gate on **global** qubits:

e.g. H on qubit q_{32}

- Need to perform 2x2 updates of the form

$$\mathrm{H} \begin{pmatrix} \psi_{rrrrrrr0\,*\cdots*\,r} \\ \psi_{rrrrrrr1\,*\cdots*\,r} \end{pmatrix}$$

- Problem: the numbers are on separate GPUs
- Naïve solution:
  - Transfer $2^n/2$ ψ's (8 TiB), perform H, transfer back
- Optimal solution:
  - Exchange global and local qubit, e.g. $q_{32} \leftrightarrow q_0$

$\psi_{0000000\,0\cdots0\,0}$

GPU rank 0

$\psi_{0000000\,1\cdots1\,0}$

$\psi_{0000000\,0\cdots0\,1}$

GPU rank 1

$\psi_{0000000\,1\cdots1\,1}$

$\vdots$

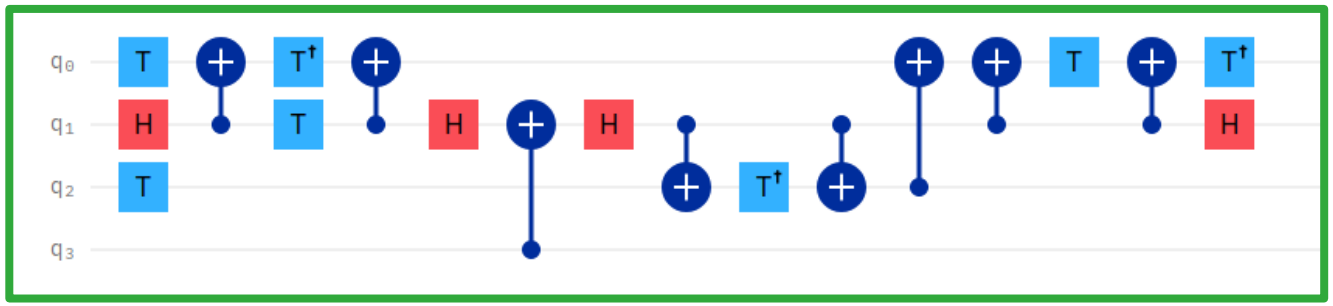
$\psi_{1111111\,0\cdots0\,1}$

GPU rank 255

$\psi_{1111111\,1\cdots1\,1}$

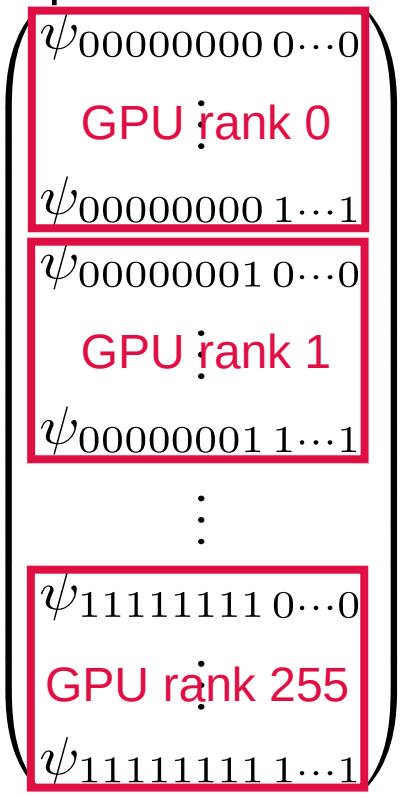
JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:



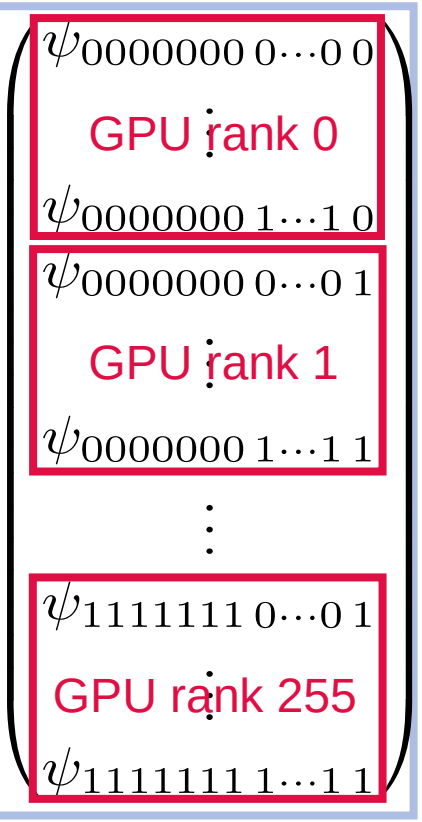
Quantum gate on **global** qubits:

e.g. H on qubit q_{32}

- Need to perform 2x2 updates of the form

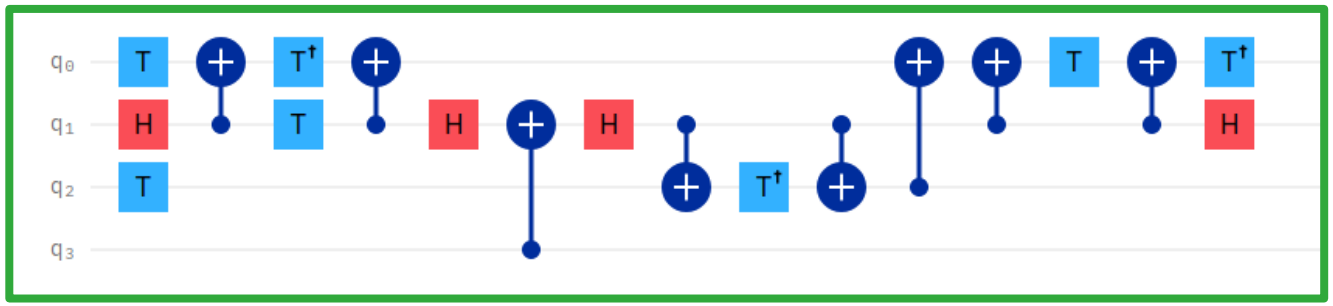
$$\mathrm{H} \begin{pmatrix} \psi_{rrrrrrr0\,*\cdots*\,r} \\ \psi_{rrrrrrr1\,*\cdots*\,r} \end{pmatrix}$$

- Problem: the numbers are on separate GPUs
- Naïve solution:
  - Transfer $2^n/2$ ψ's (8 TiB), perform H, transfer back
- Optimal solution:
  - Exchange global and local qubit, e.g. $q_{32} \leftrightarrow q_0$
  - Keep track of qubit assignment in a permutation



JUQCS-G

CUDA implementation



How to implement these **matrix-vector multiplications** in the most efficient way?

Full quantum state:

$\psi_{00000000\,0\cdots0}$

GPU rank 0

$\psi_{00000000\,1\cdots1}$

$\psi_{00000001\,0\cdots0}$

GPU rank 1

$\psi_{00000001\,1\cdots1}$

$\psi_{11111111\,0\cdots0}$

GPU rank 255

$\psi_{11111111\,1\cdots1}$

Quantum gate on **global** qubits:

e.g. H on qubit q_{32}

- Need to perform 2x2 updates of the form

$$\mathrm{H} \begin{pmatrix} \psi_{rrrrrrr0\,*\cdots*\,r} \\ \psi_{rrrrrrr1\,*\cdots*\,r} \end{pmatrix}$$

- Problem: the numbers are on separate GPUs
- Naïve solution:
  - Transfer $2^n/2$ ψ's (8 TiB), perform H, transfer back
- Optimal solution:
  - Exchange global and local qubit, e.g. $q_{32} \leftrightarrow q_0$
  - Keep track of qubit assignment in a permutation
  - Transfer $2^n/2$ ψ's only **once**

$\psi_{0000000\,0\cdots0\,0}$

GPU rank 0

$\psi_{0000000\,1\cdots1\,0}$

$\psi_{0000000\,0\cdots0\,1}$

GPU rank 1

$\psi_{0000000\,1\cdots1\,1}$

$\psi_{1111111\,0\cdots0\,1}$

GPU rank 255

$\psi_{1111111\,1\cdots1\,1}$

JUQCS-G

MPI communication scheme

Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

JUQCS-G

MPI communication scheme

Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

After transfer:

$$\begin{pmatrix} \psi_{rrrrrrr*\cdots*r} \\ \psi_{rrrrrrr*\cdots*r} \end{pmatrix}$$

```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

JUQCS-G

MPI communication scheme

Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

After transfer:

$$\begin{pmatrix} \psi_{rrrrrrr*\cdots*r} \\ \psi_{rrrrrrr*\cdots*r} \end{pmatrix}$$

- Build buffer on each GPU

```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

JUQCS-G

MPI communication scheme

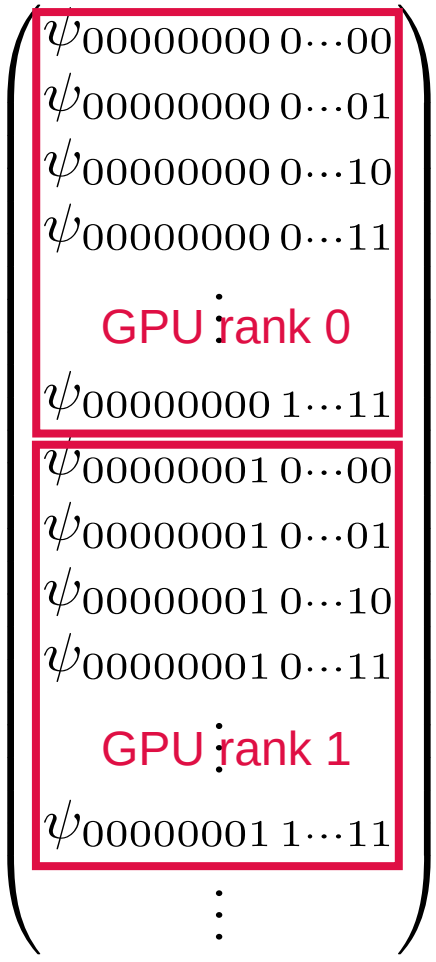
Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

After transfer:

$$\begin{pmatrix} \psi_{rrrrrrr*\cdots*r} \\ \psi_{rrrrrrr*\cdots*r} \end{pmatrix}$$

- Build buffer on each GPU




```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

JUQCS-G

MPI communication scheme

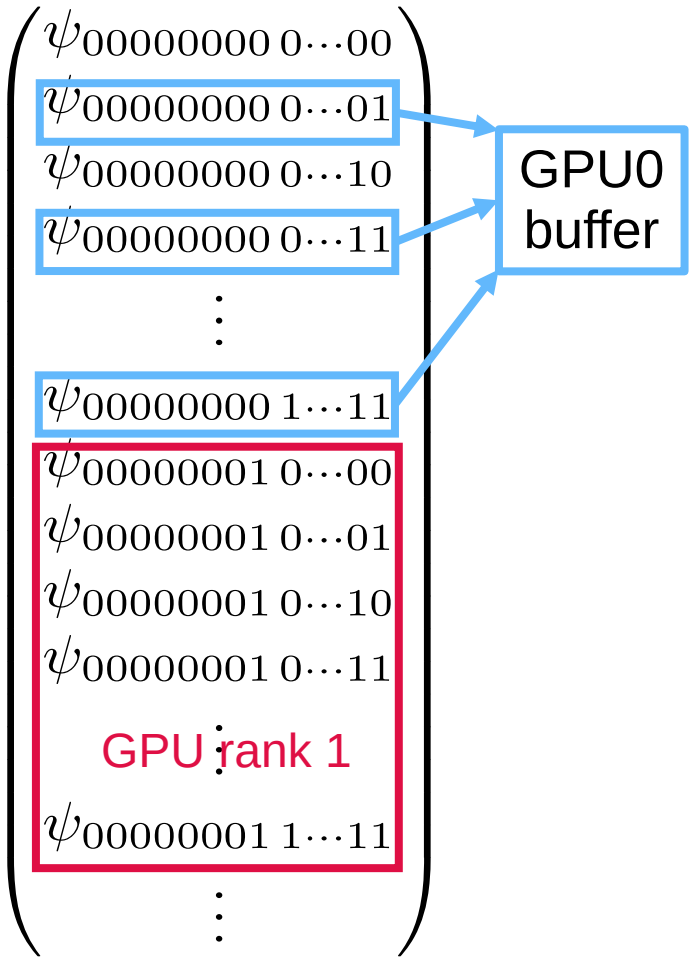
Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

After transfer:

$$\begin{pmatrix} \psi_{rrrrrrr*\cdots*r} \\ \psi_{rrrrrrr*\cdots*r} \end{pmatrix}$$

- Build buffer on each GPU




```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

JUQCS-G

MPI communication scheme

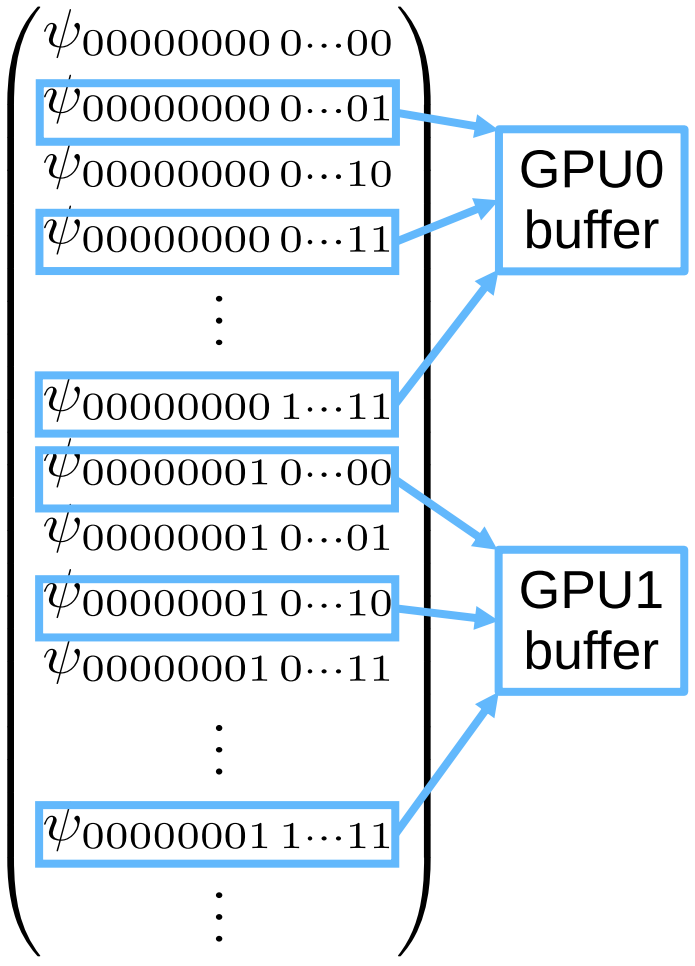
Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

After transfer:

$$\begin{pmatrix} \psi_{rrrrrrr*\cdots*r} \\ \psi_{rrrrrrr*\cdots*r} \end{pmatrix}$$

- Build buffer on each GPU




```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

JUQCS-G

MPI communication scheme

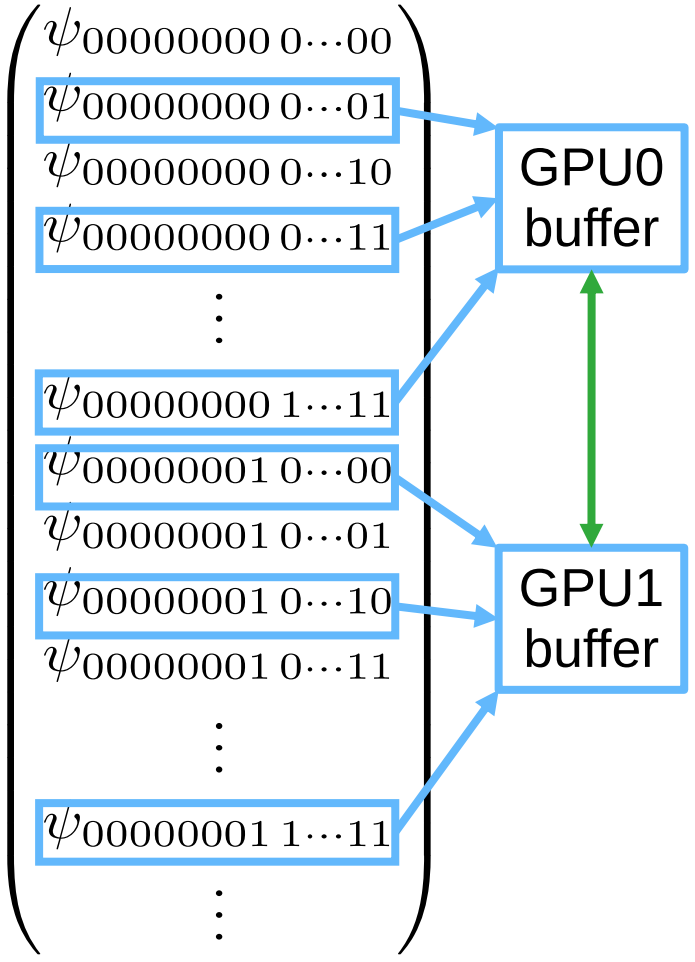
Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

After transfer:

$$\begin{pmatrix} \psi_{rrrrrrr*\cdots*r} \\ \psi_{rrrrrrr*\cdots*r} \end{pmatrix}$$

- Build buffer on each GPU
- Transfer with MPI




```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

JUQCS-G

MPI communication scheme

Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

After transfer:

$$\begin{pmatrix} \psi_{rrrrrrr*\cdots*r} \\ \psi_{rrrrrrr*\cdots*r} \end{pmatrix}$$

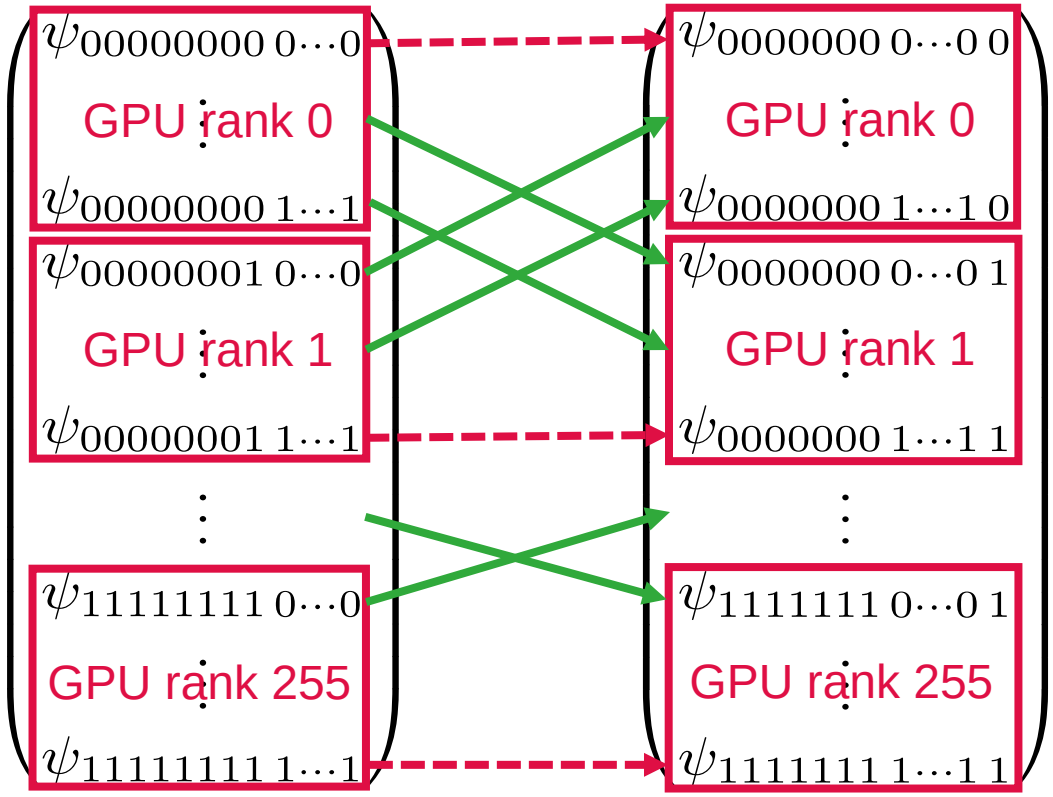
- Build buffer on each GPU
- Transfer with MPI

$$\begin{pmatrix} \psi_{00000000\,0\cdots00} \\ \psi_{00000000\,0\cdots01} \\ \psi_{00000000\,0\cdots10} \\ \psi_{00000000\,0\cdots11} \\ \vdots \\ \psi_{00000000\,1\cdots11} \\ \psi_{00000001\,0\cdots00} \\ \psi_{00000001\,0\cdots01} \\ \psi_{00000001\,0\cdots10} \\ \psi_{00000001\,0\cdots11} \\ \vdots \\ \psi_{00000001\,1\cdots11} \\ \vdots \end{pmatrix}$$

GPU0 buffer

GPU1 buffer

```
n=blockidx%x-1
n=n*blockdim%x+threadidx%x-1
idx=n+noff
do i=0,nswap-1
j=bitmasklist(i)
idx=ishft(iand(idx,not(j)),one)+iand(idx,j)
enddo
buf(n)=phi(ior(idx,jsourmask))
```




```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

JUQCS-G

MPI communication scheme

Before transfer:

$$\begin{pmatrix} \psi_{rrrrrrrr*\cdots*} \\ \psi_{rrrrrrrr*\cdots*} \end{pmatrix}$$

After transfer:

$$\begin{pmatrix} \psi_{rrrrrrr*\cdots*r} \\ \psi_{rrrrrrr*\cdots*r} \end{pmatrix}$$

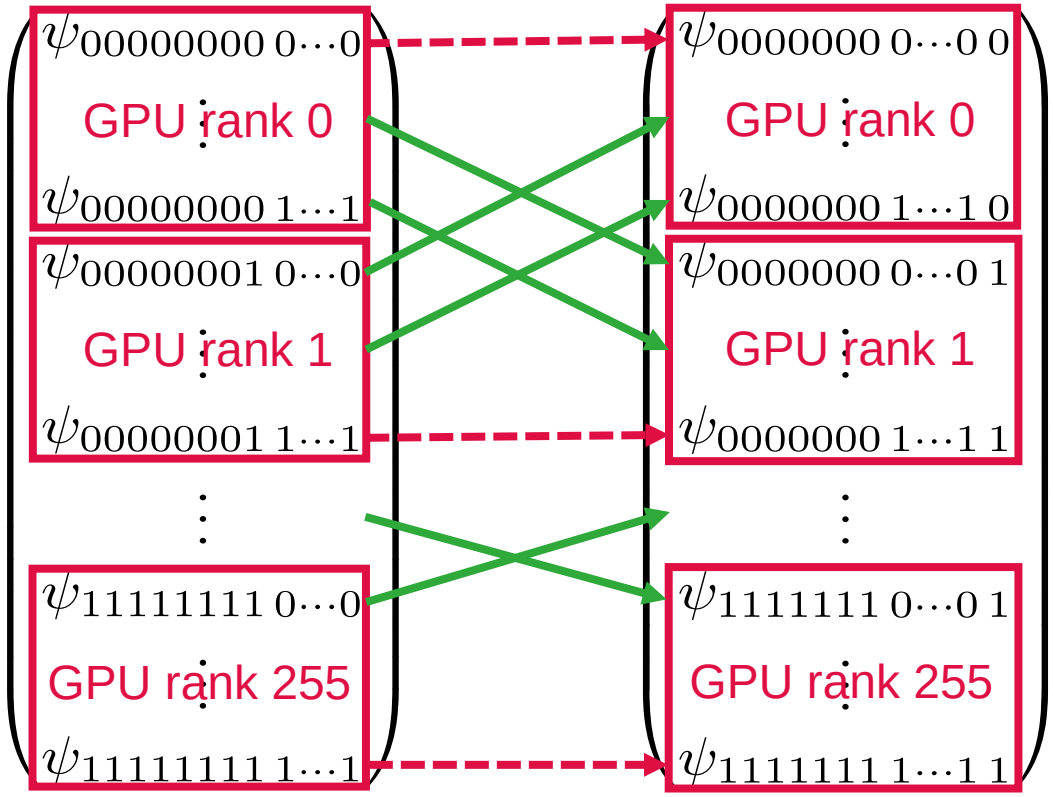
- Build buffer on each GPU
- Transfer with MPI
- Retrieve from buffer

$$\begin{pmatrix} \psi_{00000000\,0\cdots00} \\ \psi_{00000000\,0\cdots01} \\ \psi_{00000000\,0\cdots10} \\ \psi_{00000000\,0\cdots11} \\ \vdots \\ \psi_{00000000\,1\cdots11} \\ \psi_{00000001\,0\cdots00} \\ \psi_{00000001\,0\cdots01} \\ \psi_{00000001\,0\cdots10} \\ \psi_{00000001\,0\cdots11} \\ \vdots \\ \psi_{00000001\,1\cdots11} \\ \vdots \end{pmatrix}$$

GPU0 buffer

GPU1 buffer

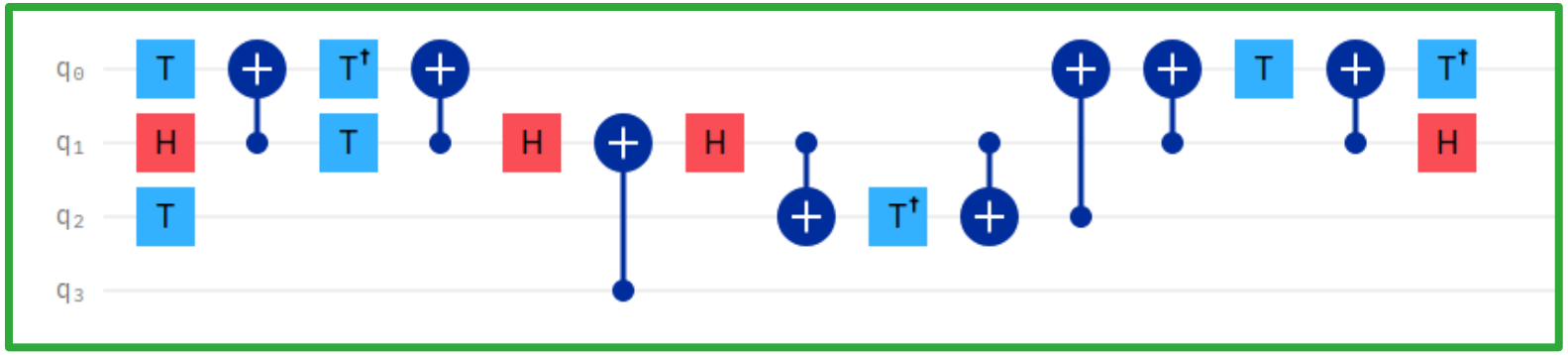
```
n=blockidx%x-1
n=n*blockdim%x+threadidx%x-1
idx=n+noff
do i=0,nswap-1
j=bitmasklist(i)
idx=ishft(iand(idx,not(j)),one)+iand(idx,j)
enddo
buf(n)=phi(ior(idx,jsourmask))
```




```
...
call psi2MPIbuf<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_I)
istat=cudaStreamSynchronize()
call MPI_SENDRECV(DD(0)%buf_I,m0,MPI_REAL8,idest,0,DD(0)%buf_R,m0,MPI_REAL8,idest,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE,IERR)
call MPIbuf2psi<<<m,n>>> (nbits,nstatesGPU,m0,n0,nswap,isourmask,DD(0)%bitmasklist,DD(0)%phi_R,DD(0)%buf_R)
istat=cudaStreamSynchronize()
...
```

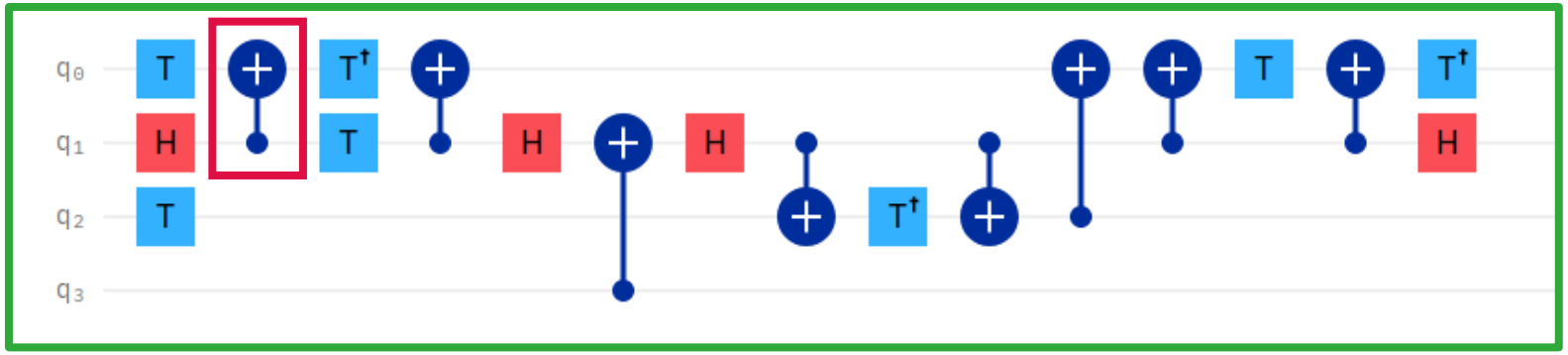
JUQCS-G

MPI communication: Two-qubit gates



JUQCS-G

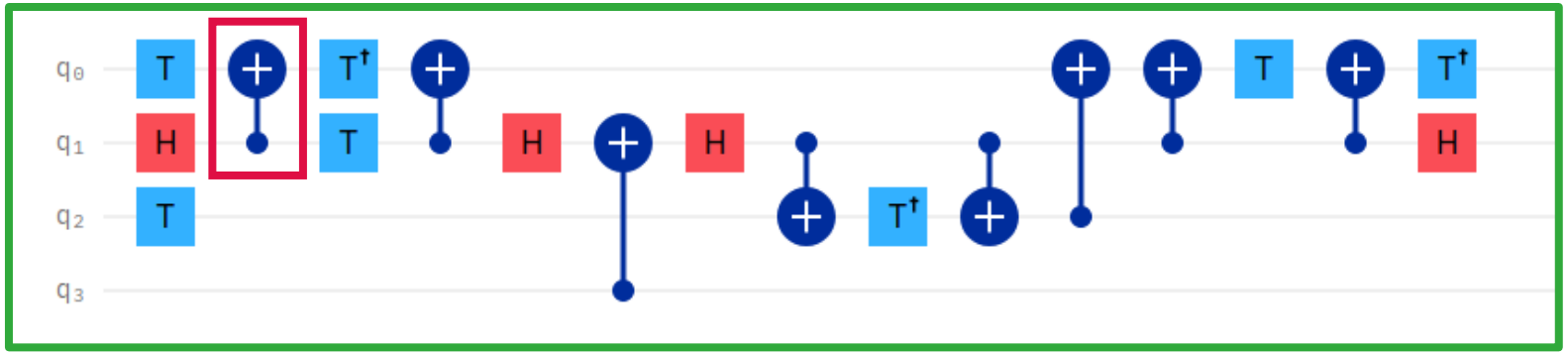
MPI communication: Two-qubit gates



➢ CNOT gate (controlled-NOT):

JUQCS-G

MPI communication: Two-qubit gates

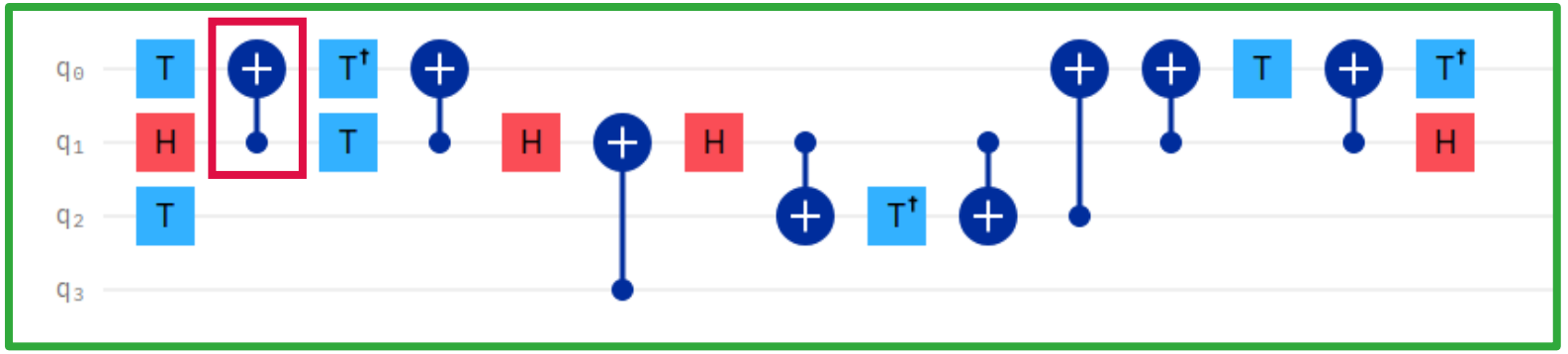


- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \text{CT} & \text{CT} & \text{CT} & \text{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

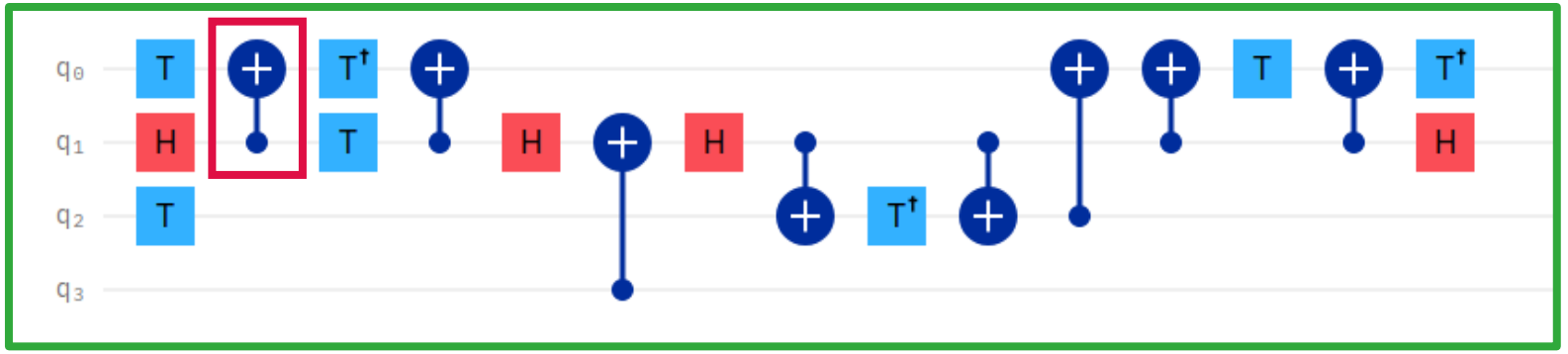
$$\begin{array}{cccc} \mathrm{CT} & \mathrm{CT} & \mathrm{CT} & \mathrm{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\psi_{q_3 q_2 q_1 q_0}$$

GPU0	GPU1	GPU2	GPU3
$\psi_{00\,00}$	$\psi_{01\,00}$	$\psi_{10\,00}$	$\psi_{11\,00}$
$\psi_{00\,01}$	$\psi_{01\,01}$	$\psi_{10\,01}$	$\psi_{11\,01}$
$\psi_{00\,10}$	$\psi_{01\,10}$	$\psi_{10\,10}$	$\psi_{11\,10}$
$\psi_{00\,11}$	$\psi_{01\,11}$	$\psi_{10\,11}$	$\psi_{11\,11}$

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \mathrm{CT} & \mathrm{CT} & \mathrm{CT} & \mathrm{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

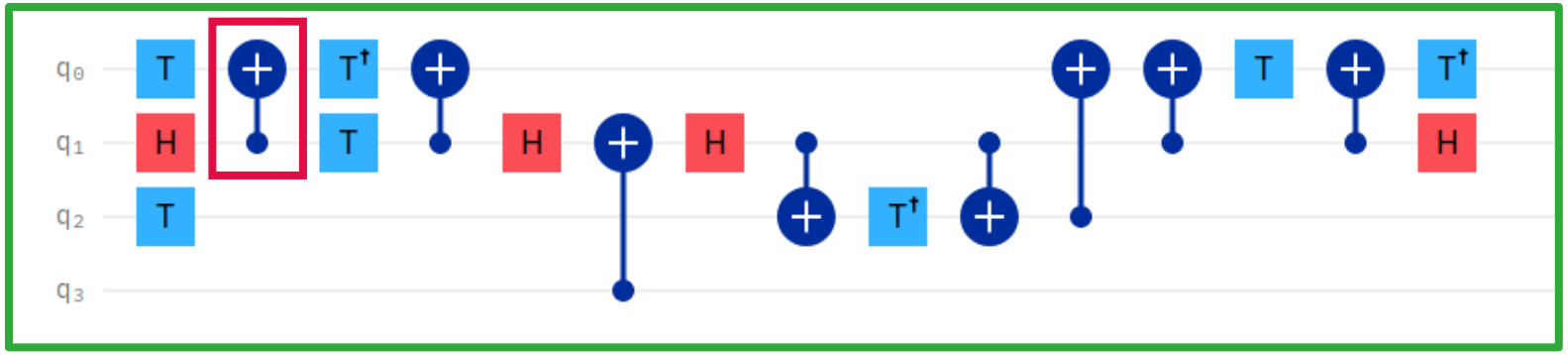
$$\psi_{q_3 q_2 q_1 q_0}$$
C T

GPU0	GPU1	GPU2	GPU3
$\psi_{00\,00}$	$\psi_{01\,00}$	$\psi_{10\,00}$	$\psi_{11\,00}$
$\psi_{00\,01}$	$\psi_{01\,01}$	$\psi_{10\,01}$	$\psi_{11\,01}$
$\psi_{00\,10}$	$\psi_{01\,10}$	$\psi_{10\,10}$	$\psi_{11\,10}$
$\psi_{00\,11}$	$\psi_{01\,11}$	$\psi_{10\,11}$	$\psi_{11\,11}$

  - “2 local”: GPUs swap locally

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \text{CT} & \text{CT} & \text{CT} & \text{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\psi_{q_3 q_2 q_1 q_0}$$
C T

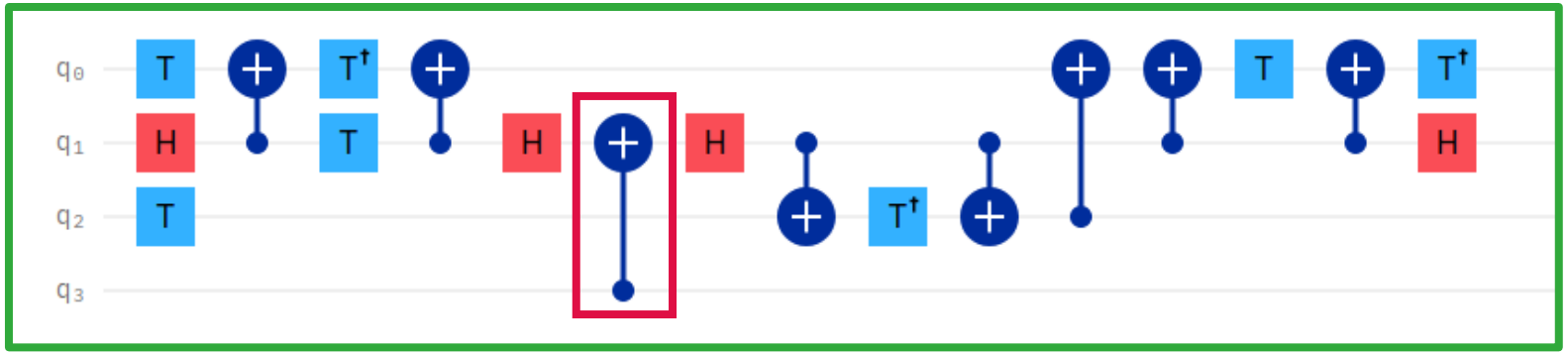
GPU0 GPU1 GPU2 GPU3

$$\begin{pmatrix} \psi_{00\,00} \\ \psi_{00\,01} \\ \psi_{00\,10} \\ \psi_{00\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{01\,00} \\ \psi_{01\,01} \\ \psi_{01\,10} \\ \psi_{01\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{10\,00} \\ \psi_{10\,01} \\ \psi_{10\,10} \\ \psi_{10\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{11\,00} \\ \psi_{11\,01} \\ \psi_{11\,10} \\ \psi_{11\,11} \end{pmatrix}$$

  - “2 local”: GPUs swap locally

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \mathrm{CT} & \mathrm{CT} & \mathrm{CT} & \mathrm{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

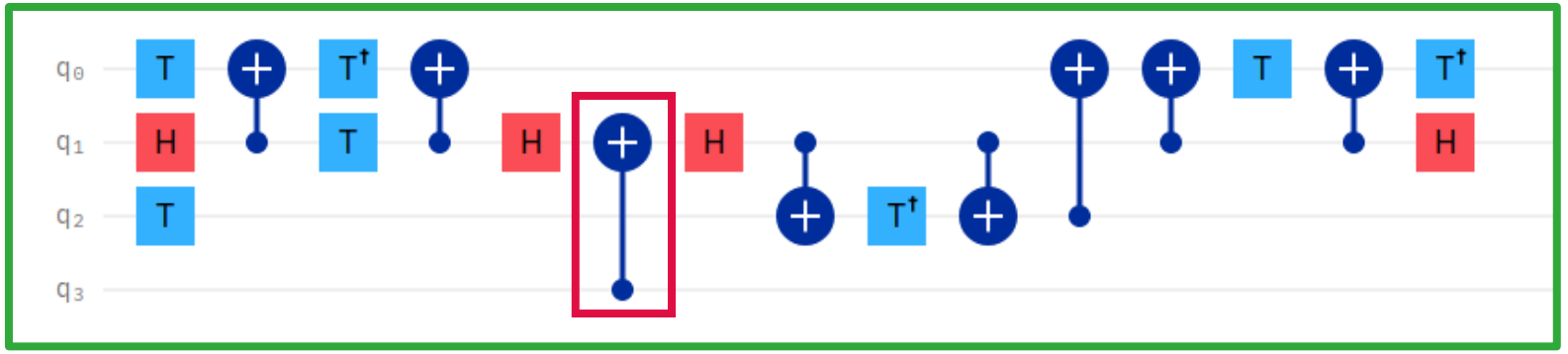
$$\psi_{q_3 q_2 q_1 q_0}$$
C T

GPU0	GPU1	GPU2	GPU3
$\psi_{00\,00}$	$\psi_{01\,00}$	$\psi_{10\,00}$	$\psi_{11\,00}$
$\psi_{00\,01}$	$\psi_{01\,01}$	$\psi_{10\,01}$	$\psi_{11\,01}$
$\psi_{00\,10}$	$\psi_{01\,10}$	$\psi_{10\,10}$	$\psi_{11\,10}$
$\psi_{00\,11}$	$\psi_{01\,11}$	$\psi_{10\,11}$	$\psi_{11\,11}$

  - “2 local”: GPUs swap locally
  - “C global, T local”: GPUs swap locally

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \mathrm{CT} & \mathrm{CT} & \mathrm{CT} & \mathrm{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\psi_{q_3 q_2 q_1 q_0}$$
C T

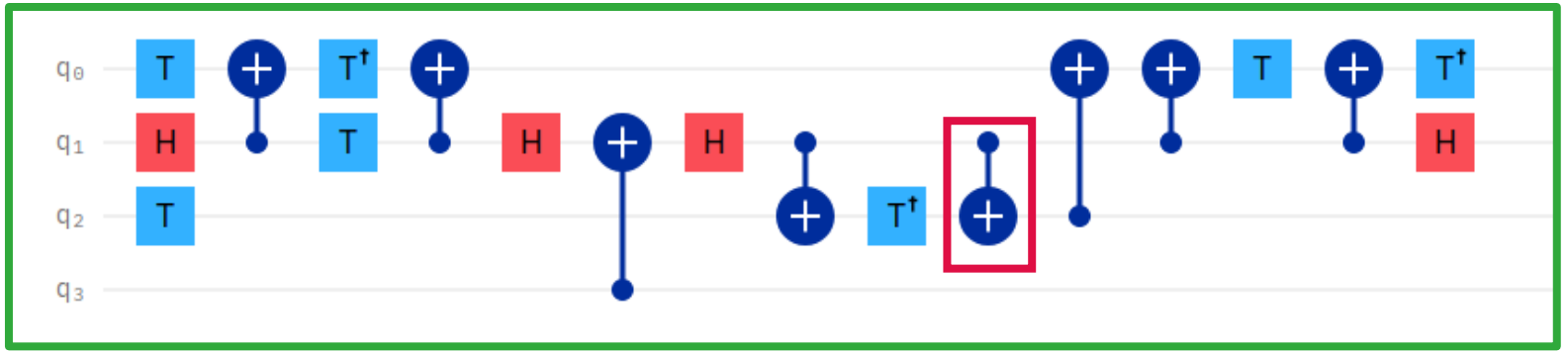
GPU0 GPU1 GPU2 GPU3

$$\begin{pmatrix} \psi_{00\,00} \\ \psi_{00\,01} \\ \psi_{00\,10} \\ \psi_{00\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{01\,00} \\ \psi_{01\,01} \\ \psi_{01\,10} \\ \psi_{01\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{10\,00} \\ \psi_{10\,01} \\ \psi_{10\,10} \\ \psi_{10\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{11\,00} \\ \psi_{11\,01} \\ \psi_{11\,10} \\ \psi_{11\,11} \end{pmatrix}$$

  - “2 local”: GPUs swap locally
  - “C global, T local”: GPUs swap locally

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \text{CT} & \text{CT} & \text{CT} & \text{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

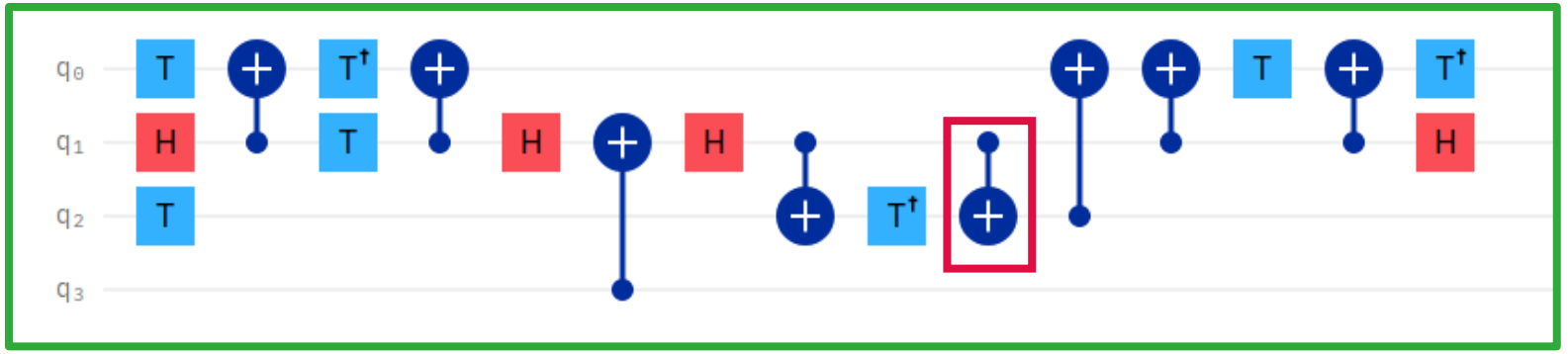
$$\psi_{q_3 q_2 q_1 q_0}$$
T C

GPU0	GPU1	GPU2	GPU3
$\psi_{00\,00}$	$\psi_{01\,00}$	$\psi_{10\,00}$	$\psi_{11\,00}$
$\psi_{00\,01}$	$\psi_{01\,01}$	$\psi_{10\,01}$	$\psi_{11\,01}$
$\psi_{00\,10}$	$\psi_{01\,10}$	$\psi_{10\,10}$	$\psi_{11\,10}$
$\psi_{00\,11}$	$\psi_{01\,11}$	$\psi_{10\,11}$	$\psi_{11\,11}$

  - “2 local”: GPUs swap locally
  - “C global, T local”: GPUs swap locally
  - “C local, T global”: MPI transfer ½ of all coefficients

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \text{CT} & \text{CT} & \text{CT} & \text{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\psi_{q_3 q_2 q_1 q_0}$$
T C

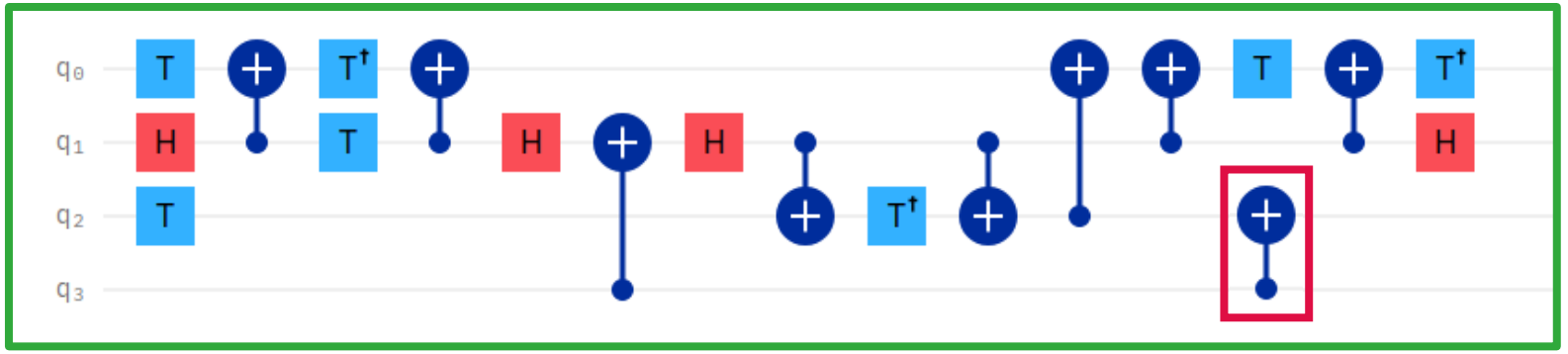
GPU0 GPU1 GPU2 GPU3

$$\begin{pmatrix} \psi_{00\,00} \\ \psi_{00\,01} \\ \psi_{00\,10} \\ \psi_{00\,11} \end{pmatrix} \begin{pmatrix} \psi_{01\,00} \\ \psi_{01\,01} \\ \psi_{01\,10} \\ \psi_{01\,11} \end{pmatrix} \begin{pmatrix} \psi_{10\,00} \\ \psi_{10\,01} \\ \psi_{10\,10} \\ \psi_{10\,11} \end{pmatrix} \begin{pmatrix} \psi_{11\,00} \\ \psi_{11\,01} \\ \psi_{11\,10} \\ \psi_{11\,11} \end{pmatrix}$$

  - “2 local”: GPUs swap locally
  - “C global, T local”: GPUs swap locally
  - “C local, T global”: MPI transfer ½ of all coefficients

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \text{CT} & \text{CT} & \text{CT} & \text{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

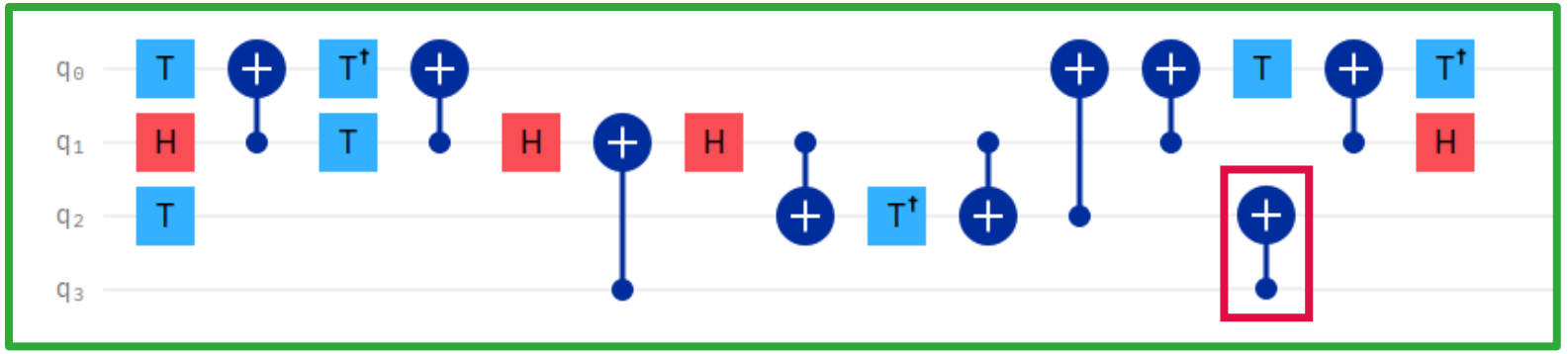
$$\psi_{q_3 q_2 q_1 q_0}$$
C T

GPU0	GPU1	GPU2	GPU3
$\psi_{00\,00}$	$\psi_{01\,00}$	$\psi_{10\,00}$	$\psi_{11\,00}$
$\psi_{00\,01}$	$\psi_{01\,01}$	$\psi_{10\,01}$	$\psi_{11\,01}$
$\psi_{00\,10}$	$\psi_{01\,10}$	$\psi_{10\,10}$	$\psi_{11\,10}$
$\psi_{00\,11}$	$\psi_{01\,11}$	$\psi_{10\,11}$	$\psi_{11\,11}$

  - “2 local”: GPUs swap locally
  - “C global, T local”: GPUs swap locally
  - “C local, T global”: MPI transfer ½ of all coefficients
  - “2 global”: MPI transfer ½ of all coefficients (or relabel GPUs)

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{cccc} \text{CT} & \text{CT} & \text{CT} & \text{CT} \\ 00 & 01 & 10 & 11 \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\psi_{q_3 q_2 q_1 q_0}$$
C T

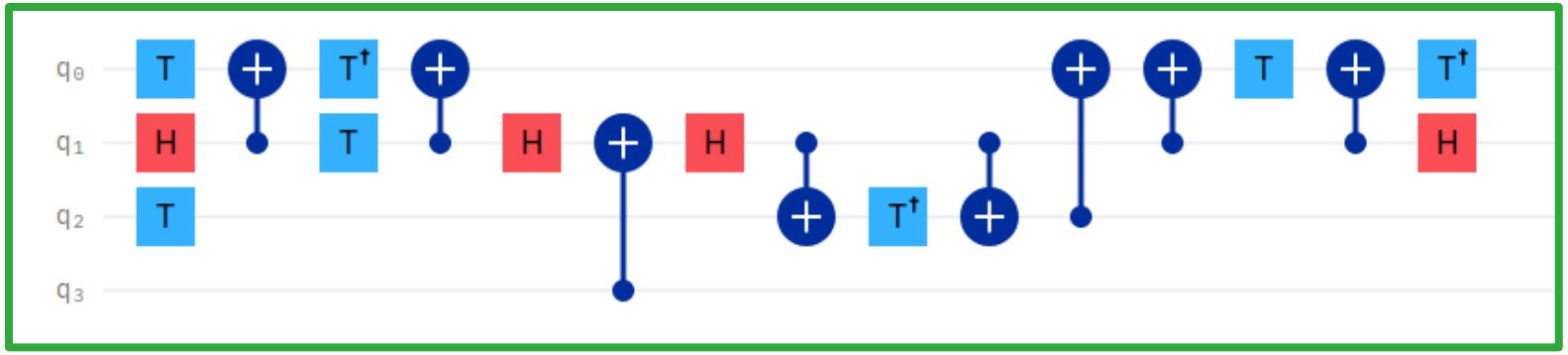
GPU0 GPU1 GPU2 GPU3

$$\begin{pmatrix} \psi_{00\,00} \\ \psi_{00\,01} \\ \psi_{00\,10} \\ \psi_{00\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{01\,00} \\ \psi_{01\,01} \\ \psi_{01\,10} \\ \psi_{01\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{10\,00} \\ \psi_{10\,01} \\ \psi_{10\,10} \\ \psi_{10\,11} \end{pmatrix} \quad \begin{pmatrix} \psi_{11\,00} \\ \psi_{11\,01} \\ \psi_{11\,10} \\ \psi_{11\,11} \end{pmatrix}$$

  - “2 local”: GPUs swap locally
  - “C global, T local”: GPUs swap locally
  - “C local, T global”: MPI transfer ½ of all coefficients
  - “2 global”: MPI transfer ½ of all coefficients (or relabel GPUs)

JUQCS-G

MPI communication: Two-qubit gates



- CNOT gate (controlled-NOT):
  - Swap all coefficients where control qubit C is 1

$$\begin{array}{c} \begin{matrix} \text{CT} & \text{CT} & \text{CT} & \text{CT} \\ 00 & 01 & 10 & 11 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \qquad \psi_{q_3 q_2 q_1 q_0}$$

$$\begin{matrix} \text{GPU0} & \text{GPU1} & \text{GPU2} & \text{GPU3} \\ \begin{pmatrix} \psi_{00\,00} \\ \psi_{00\,01} \\ \psi_{00\,10} \\ \psi_{00\,11} \end{pmatrix} & \begin{pmatrix} \psi_{01\,00} \\ \psi_{01\,01} \\ \psi_{01\,10} \\ \psi_{01\,11} \end{pmatrix} & \begin{pmatrix} \psi_{10\,00} \\ \psi_{10\,01} \\ \psi_{10\,10} \\ \psi_{10\,11} \end{pmatrix} & \begin{pmatrix} \psi_{11\,00} \\ \psi_{11\,01} \\ \psi_{11\,10} \\ \psi_{11\,11} \end{pmatrix} \end{matrix}$$

  - “2 local”: GPUs swap locally
  - “C global, T local”: GPUs swap locally
  - “C local, T global”: MPI transfer ½ of all coefficients
  - “2 global”: MPI transfer ½ of all coefficients (or relabel GPUs)
- For benchmarking: MPI transfer whenever coefficients on different GPUs are combined

JUQCS-G

Running on JUWELS Booster

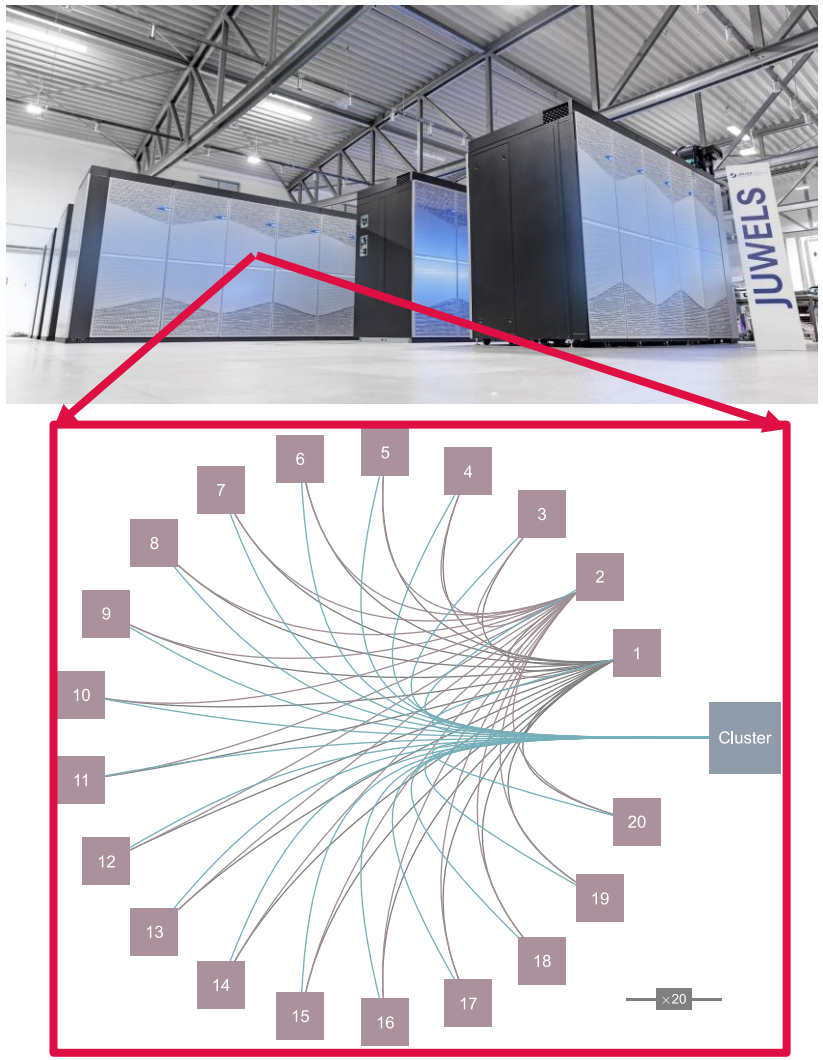
JUQCS-G

Running on JUWELS Booster

- 936 Compute nodes

JUQCS-G

Running on JUWELS Booster

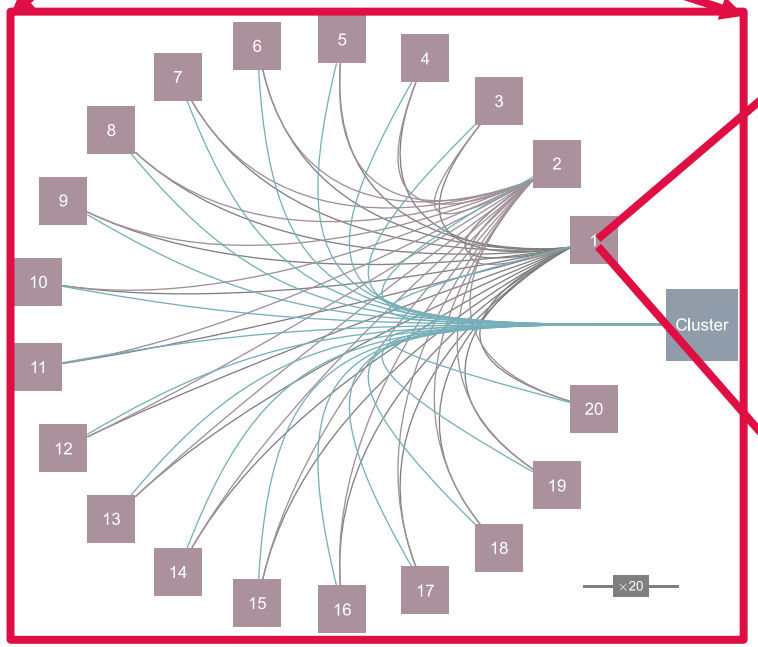


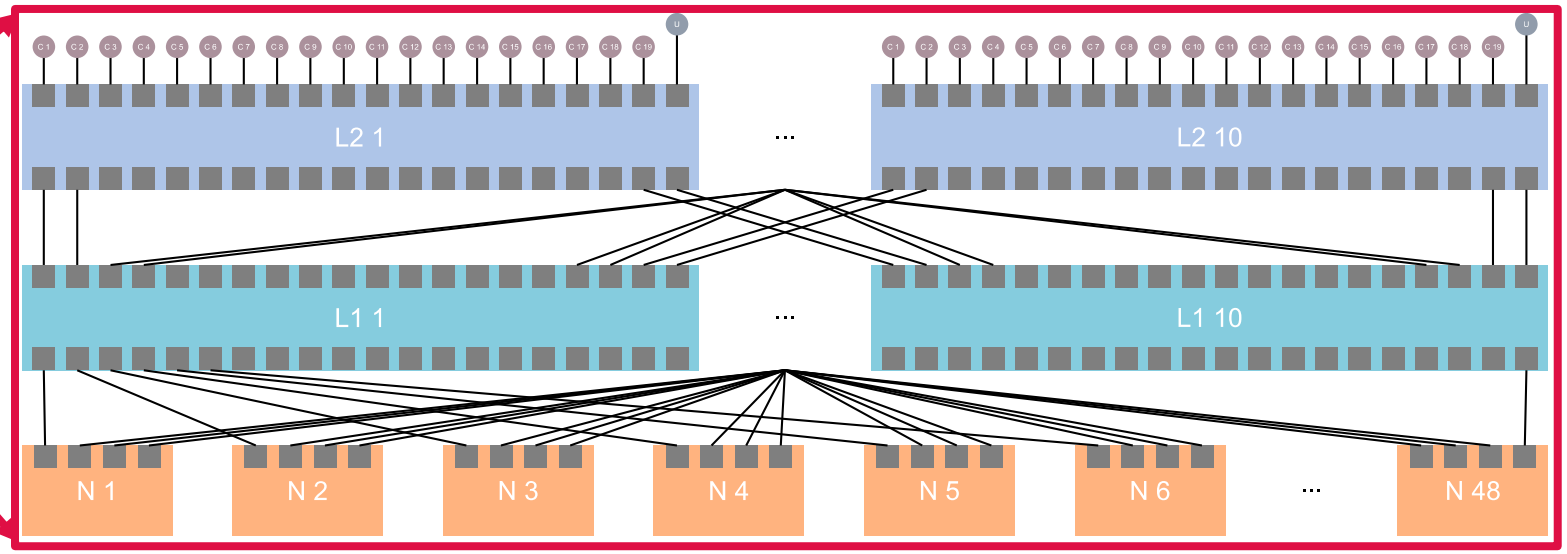
- 936 Compute nodes
- DragonFly+ Topology

JUQCS-G

Running on JUWELS Booster

- 936 Compute nodes
- DragonFly+ Topology



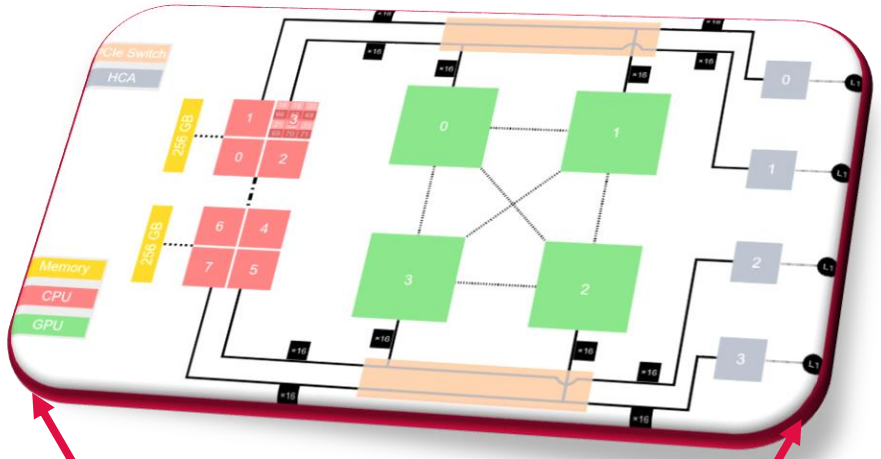


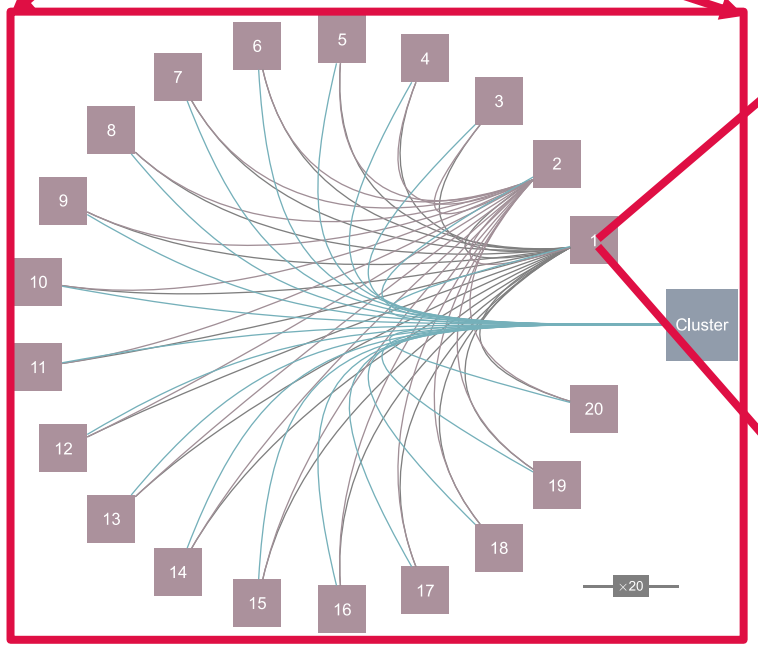
JUQCS-G

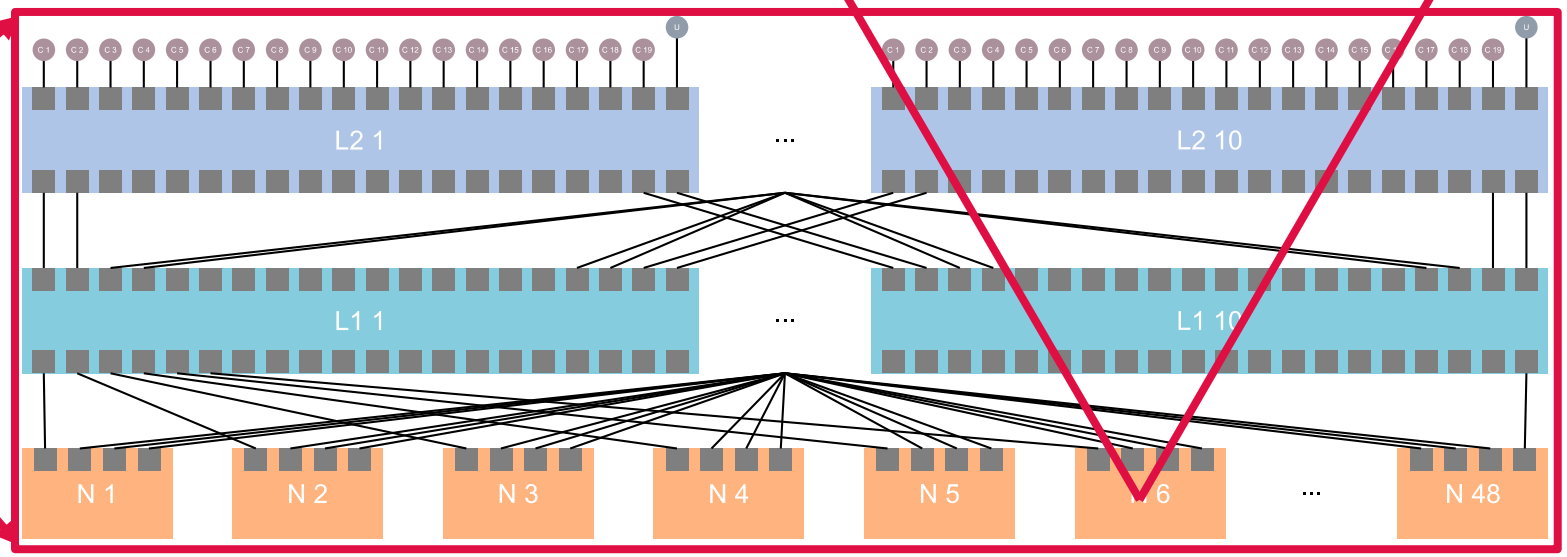
Running on JUWELS Booster

- 936 Compute nodes
- DragonFly+ Topology

4 A100 GPUs per node





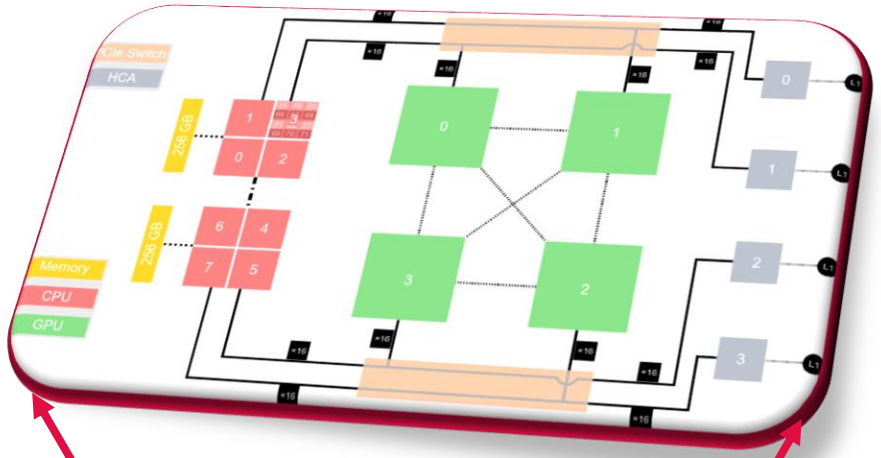


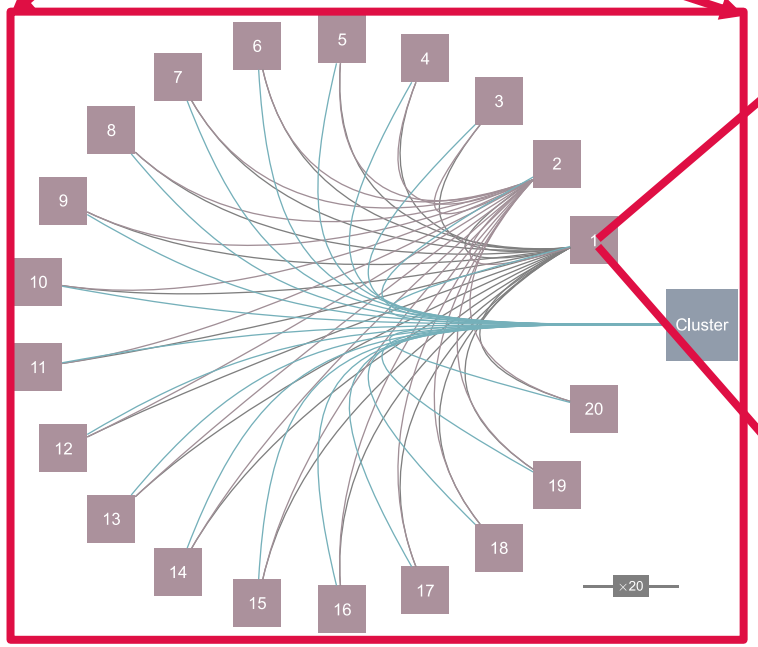
JUQCS-G

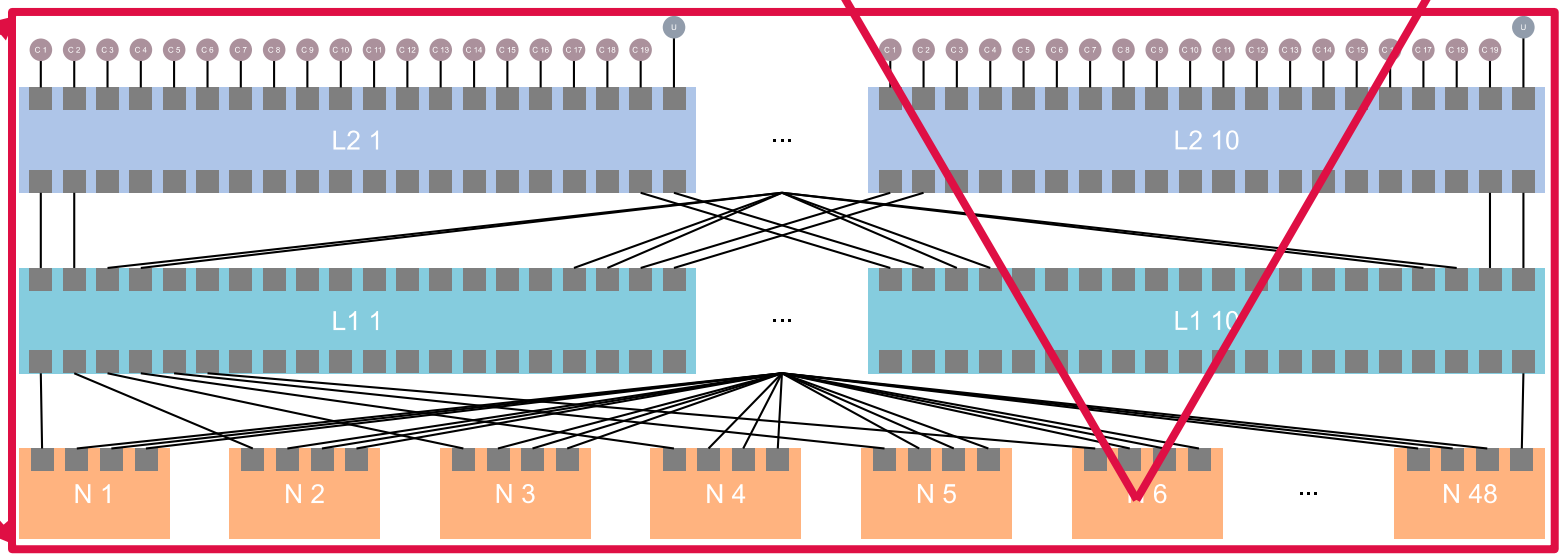
Running on JUWELS Booster

- 936 Compute nodes
- DragonFly+ Topology
- 3744 NVIDIA A100 GPUs

4 A100 GPUs per node





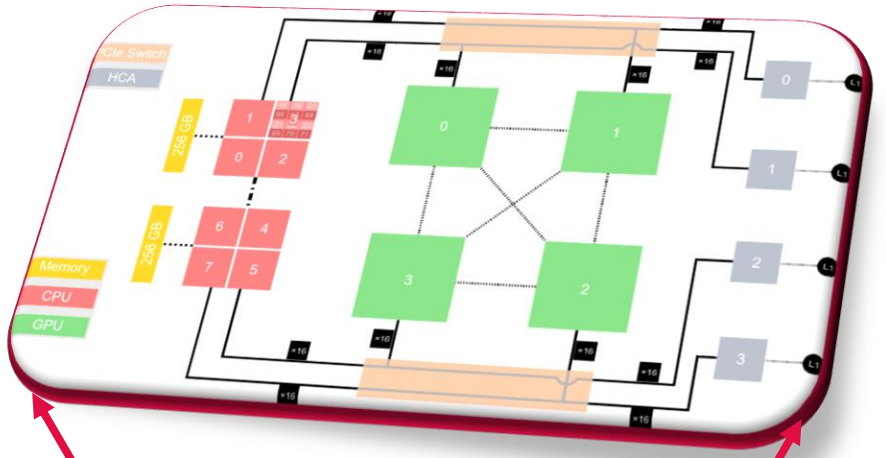


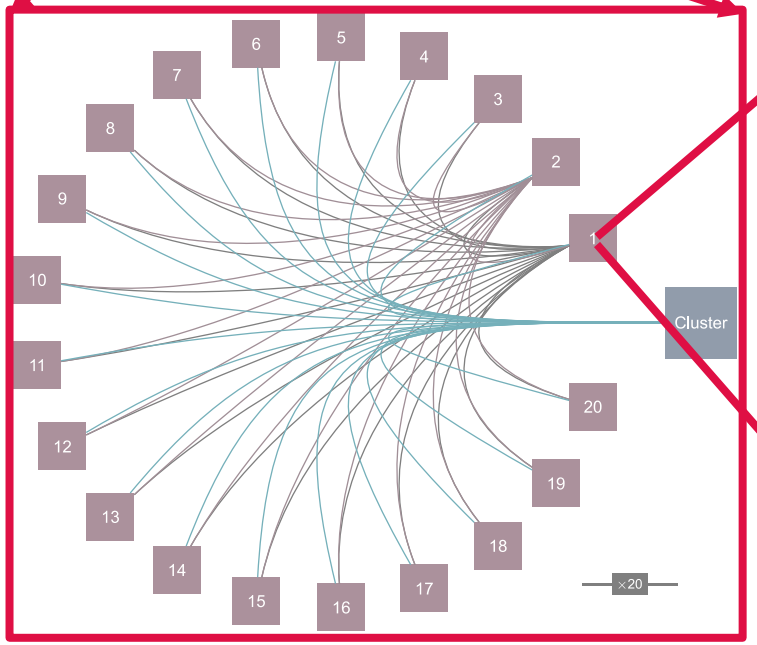
JUQCS-G

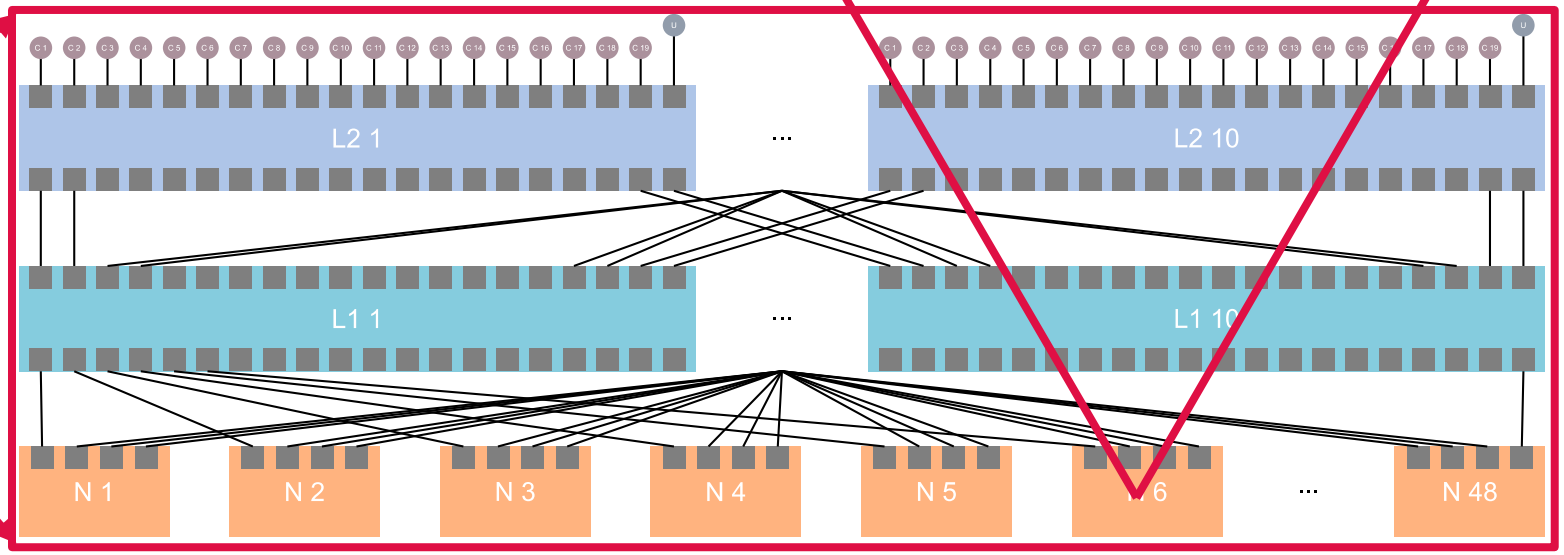
Running on JUWELS Booster

- 936 Compute nodes
- DragonFly+ Topology
- 3744 NVIDIA A100 GPUs
- 73 PFLOP/s

4 A100 GPUs per node







JUQCS-G

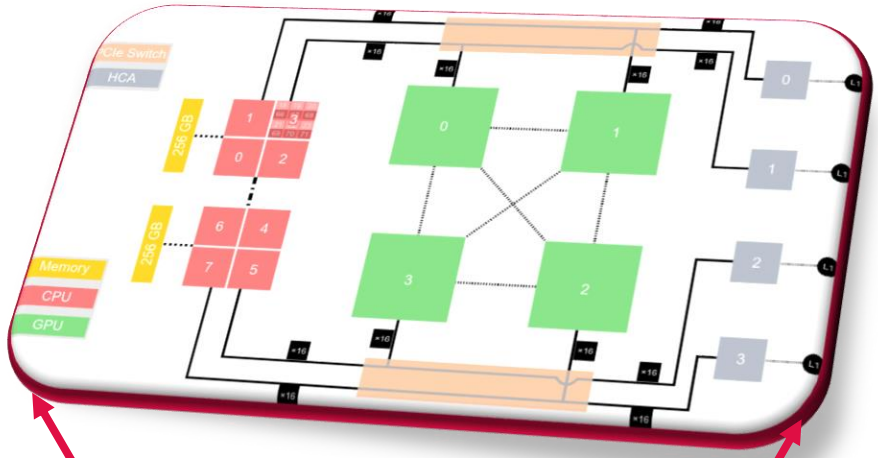
Running on JUWELS Booster

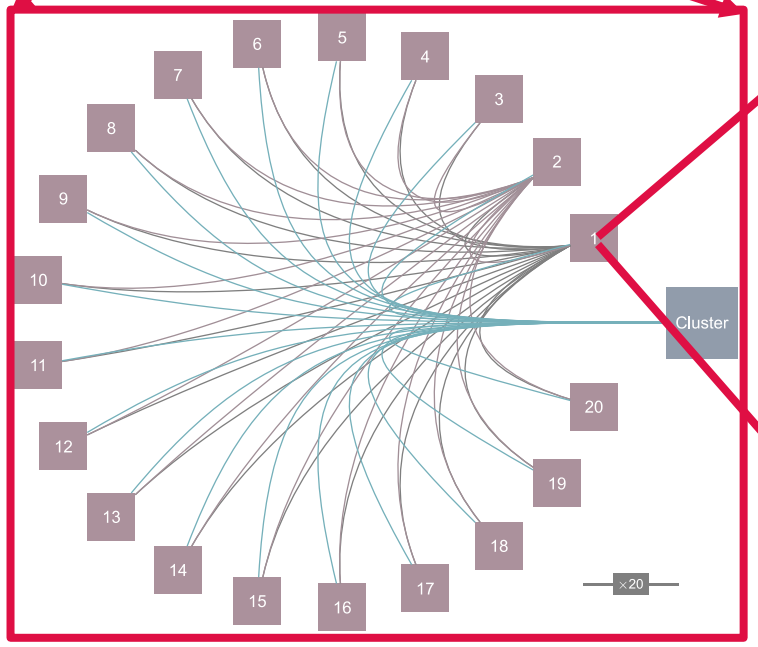
Top500 Nov-2020:

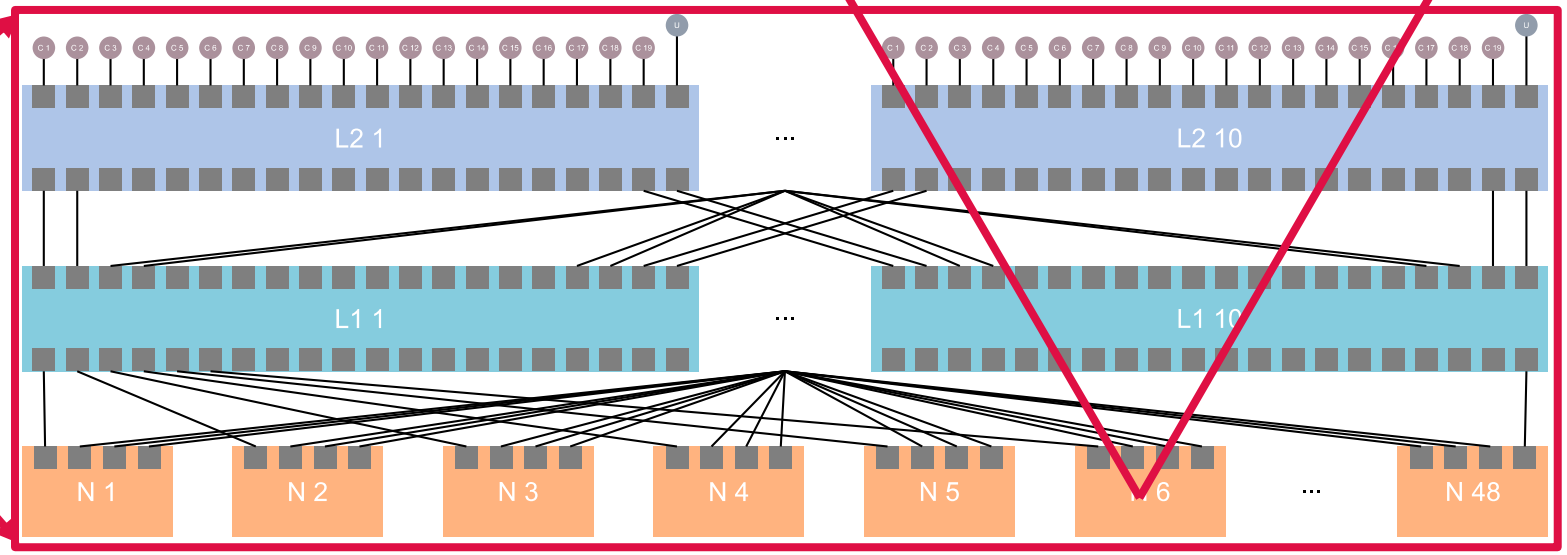
- #1 Europe
- #7 World
- #3 Green500

- 936 Compute nodes
- DragonFly+ Topology
- 3744 NVIDIA A100 GPUs
- 73 PFLOP/s

4 A100 GPUs per node







JUQCS-G

Why we can use it to benchmark GPU clusters like JUWELS Booster

JUQCS-G

Why we can use it to benchmark GPU clusters like JUWELS Booster

- **Memory-intensive:**
  - For 40 qubits: $2^{40}\ \psi's$ = 16 TiB memory

JUQCS-G

Why we can use it to benchmark GPU clusters like JUWELS Booster

- **Memory-intensive:**
  - For 40 qubits: $2^{40}\,\psi'\mathrm{s}$ = 16 TiB memory
- **Network-intensive:**
  - Global gates need transfer of **one half** of all memory
  - For 40 qubits: $2^{40}/2\,\psi'\mathrm{s}$ = 8 TiB transfer

JUQCS-G

Why we can use it to benchmark GPU clusters like JUWELS Booster

- **Memory-intensive:**
  - For 40 qubits: $2^{40}\,\psi'\mathrm{s}$ = 16 TiB memory
- **Network-intensive:**
  - Global gates need transfer of **one half** of all memory
  - For 40 qubits: $2^{40}/2\,\psi'\mathrm{s}$ = 8 TiB transfer
- **High GPU utilization**
  - For 40 qubits:
    - 32 GiB on 512 GPUs
    - 16 GiB on 1024 GPUs
    - 8 GiB on 2048 GPUs

JUQCS-G

Why we can use it to benchmark GPU clusters like JUWELS Booster

- **Memory-intensive:**
  - For 40 qubits: $2^{40}\,\psi'\mathrm{s}$ = 16 TiB memory
- **Network-intensive:**
  - Global gates need transfer of **one half** of all memory
  - For 40 qubits: $2^{40}/2\,\psi'\mathrm{s}$ = 8 TiB transfer
- **High GPU utilization**
  - For 40 qubits:
    - 32 GiB on 512 GPUs
    - 16 GiB on 1024 GPUs
    - 8 GiB on 2048 GPUs
- Using **GPUs** to simulate **universal QPUs**

QPU = Quantum Processing Unit

JUQCS-G

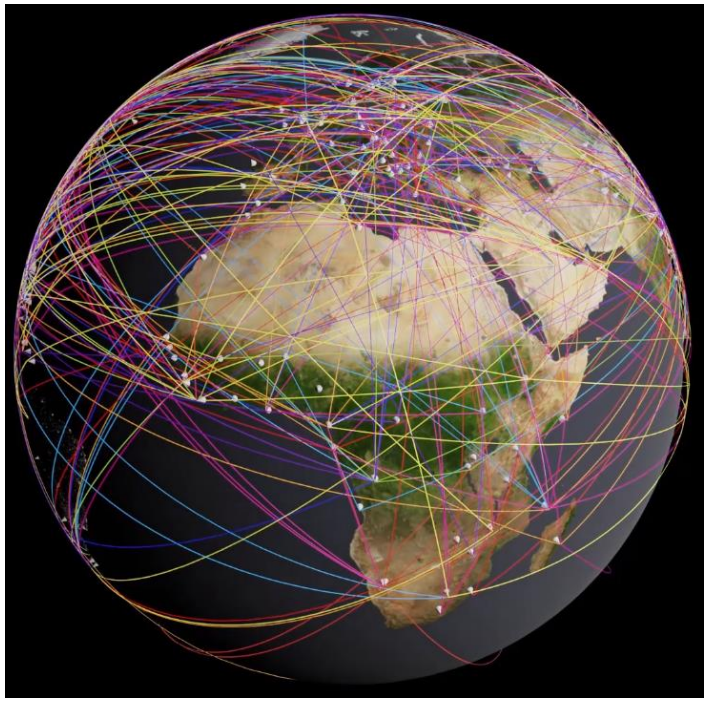
Why we can use it to benchmark GPU clusters like JUWELS Booster

- **Memory-intensive:**
  - For 40 qubits: $2^{40}\ \psi's$ = 16 TiB memory
- **Network-intensive:**
  - Global gates need transfer of **one half** of all memory
  - For 40 qubits: $2^{40}/2\ \psi's$ = 8 TiB transfer
- **High GPU utilization**
  - For 40 qubits:
    - 32 GiB on 512 GPUs
    - 16 GiB on 1024 GPUs
    - 8 GiB on 2048 GPUs
- Using **GPUs** to simulate **universal QPUs**

QPU = Quantum Processing Unit

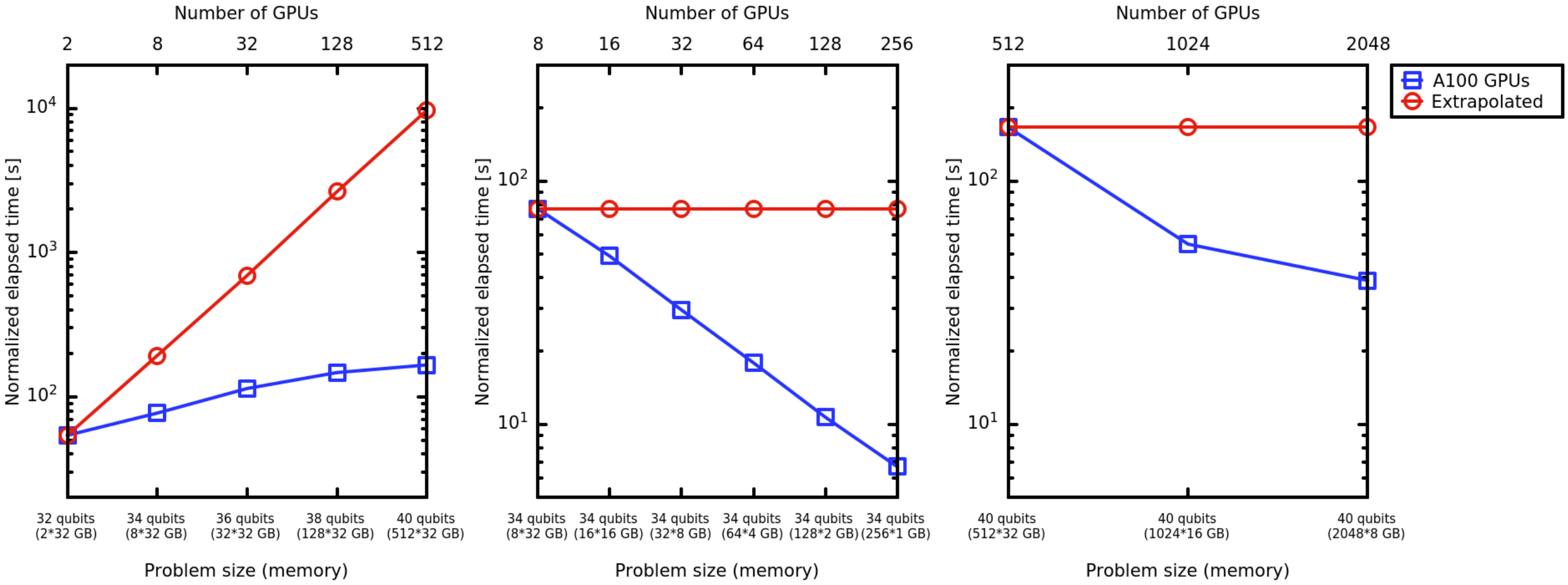
Application:

Solve airplane scheduling problems using the **Q**uantum **A**pproximate **O**ptimization **A**lgorithm



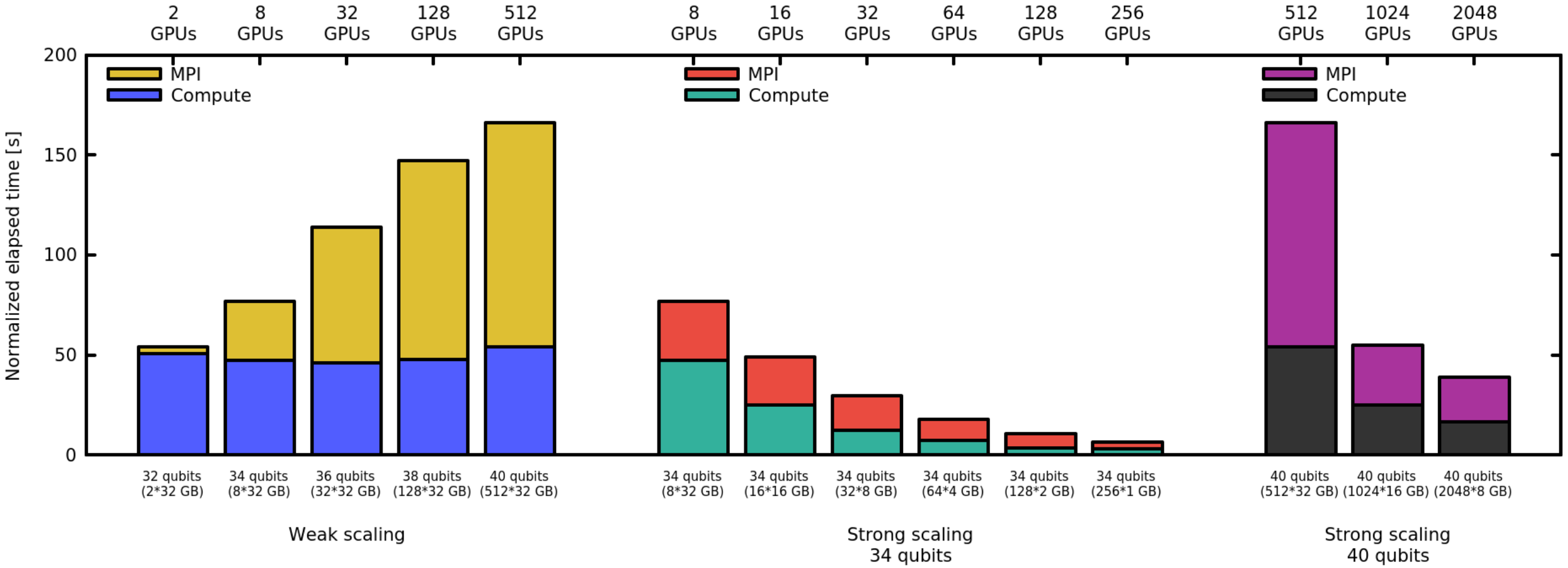
JUQCS-G

Weak and strong scaling results



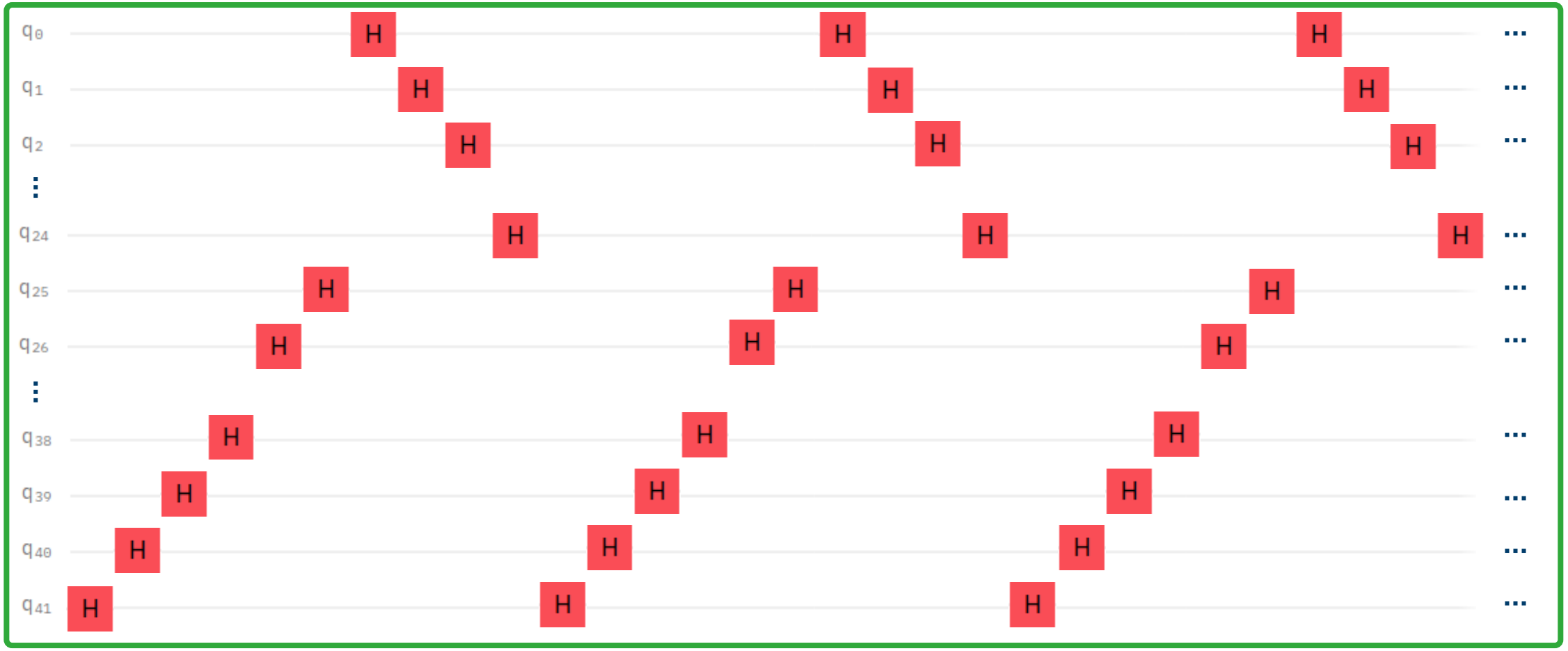
JUQCS-G

Weak and strong scaling results: MPI vs. Compute Time



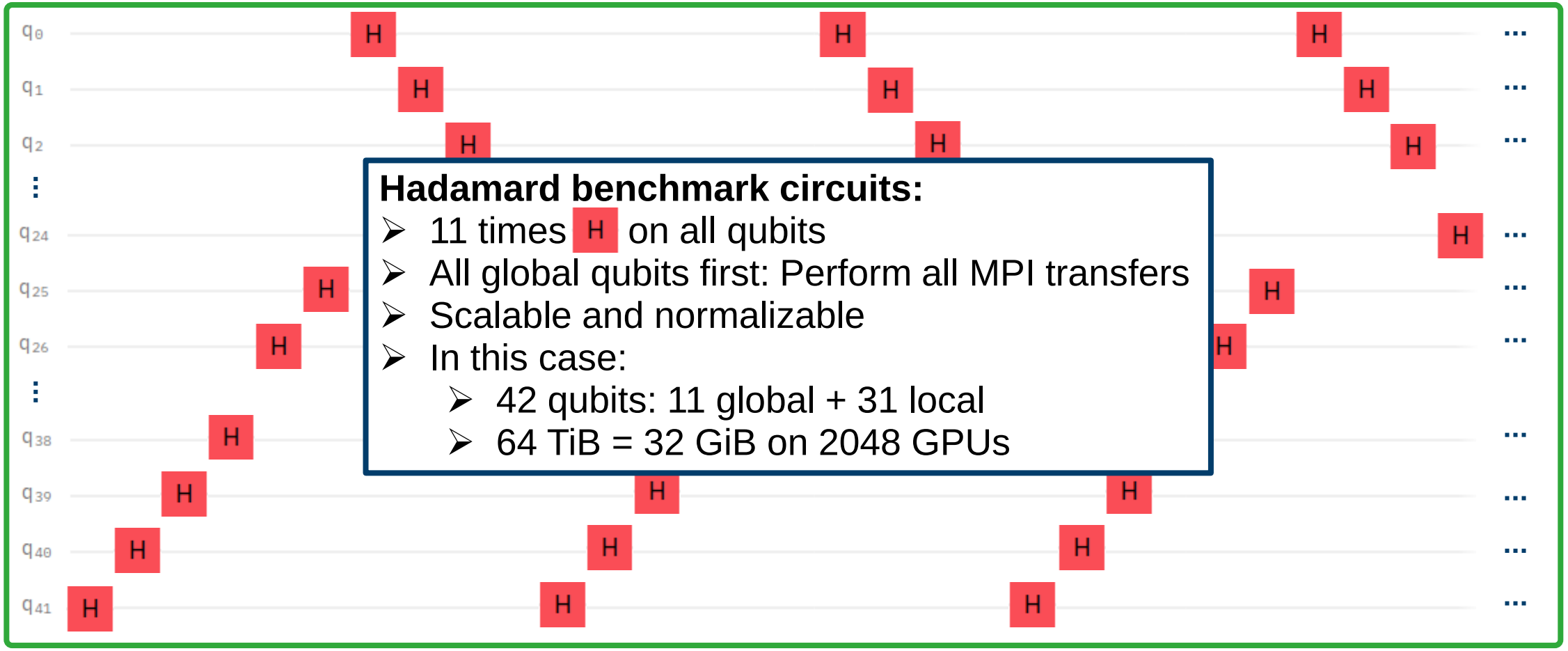
JUQCS-G

Weak and strong scaling results: Hadamard benchmark circuits



JUQCS-G

Weak and strong scaling results: Hadamard benchmark circuits

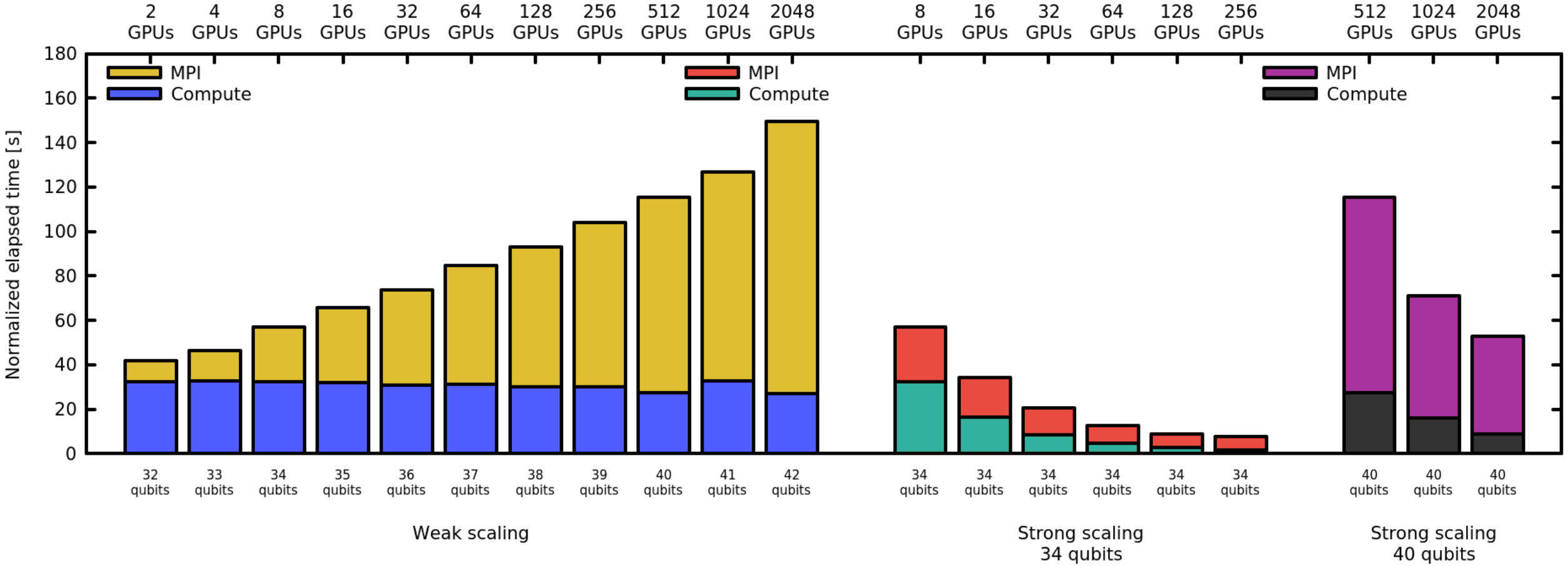


Hadamard benchmark circuits:

- 11 times H on all qubits
- All global qubits first: Perform all MPI transfers
- Scalable and normalizable
- In this case:
  - 42 qubits: 11 global + 31 local
  - 64 TiB = 32 GiB on 2048 GPUs

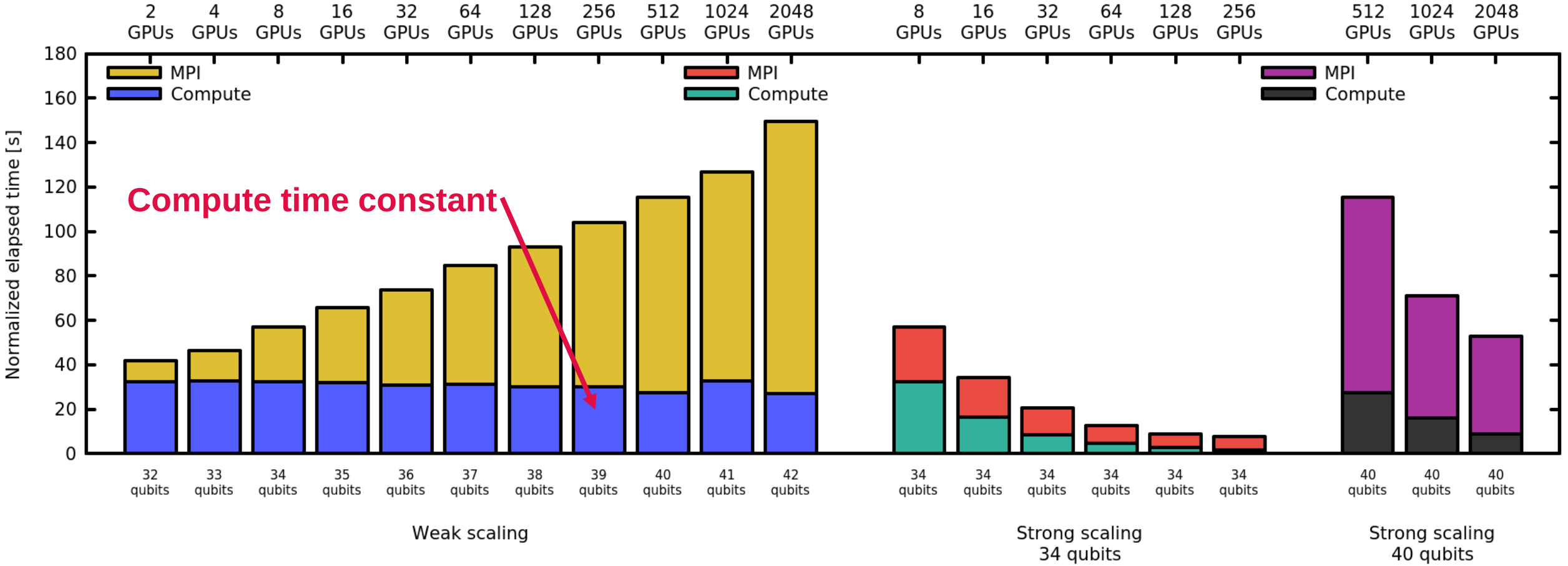
JUQCS-G

Weak and strong scaling results: Hadamard benchmark circuits



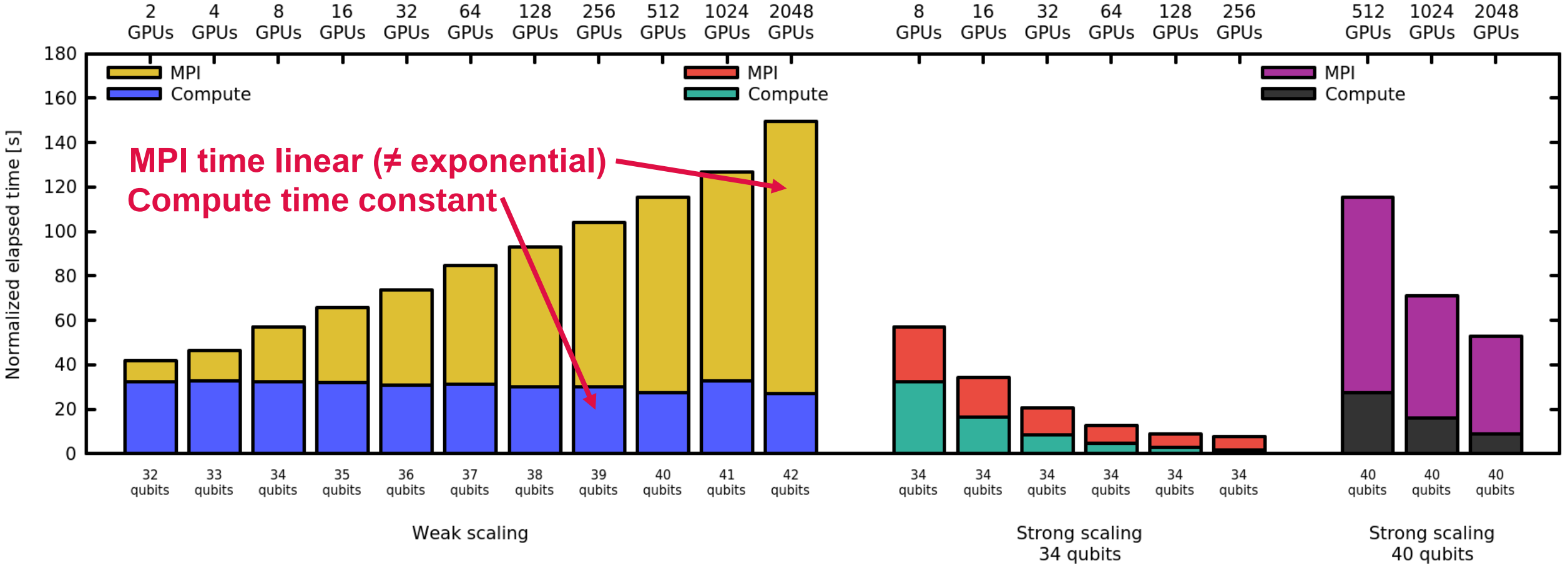
JUQCS-G

Weak and strong scaling results: Hadamard benchmark circuits



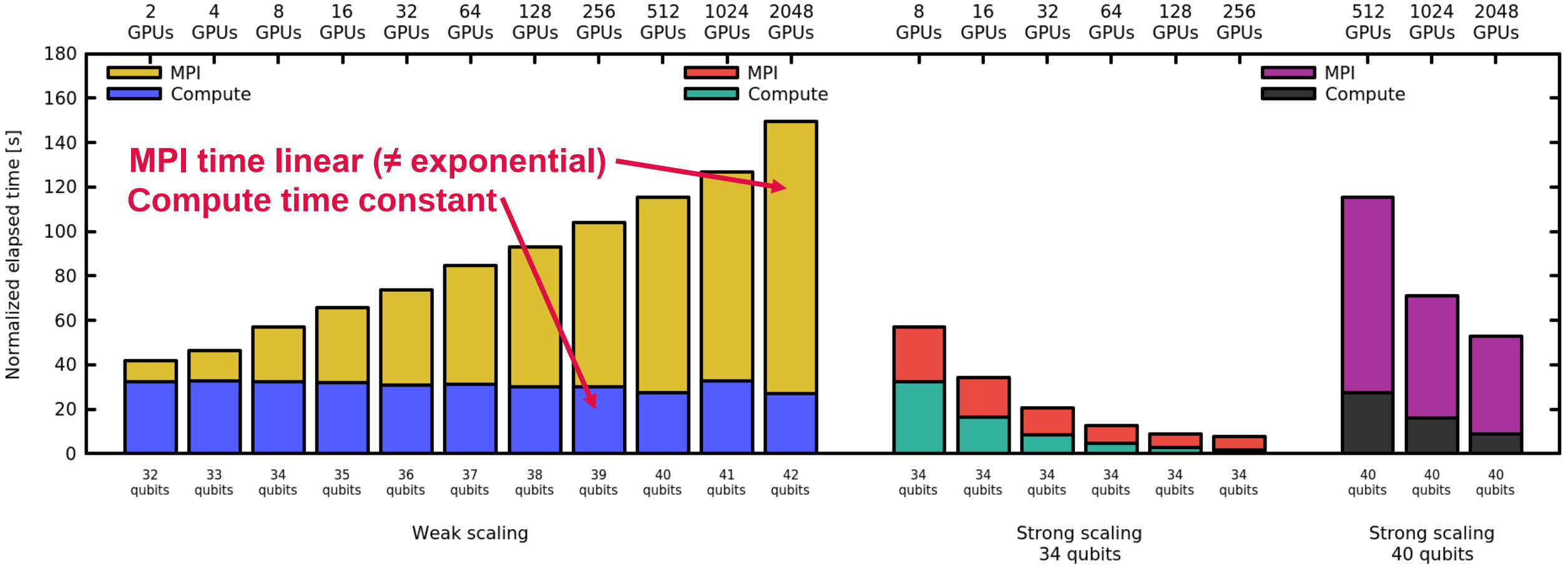
JUQCS-G

Weak and strong scaling results: Hadamard benchmark circuits



JUQCS-G

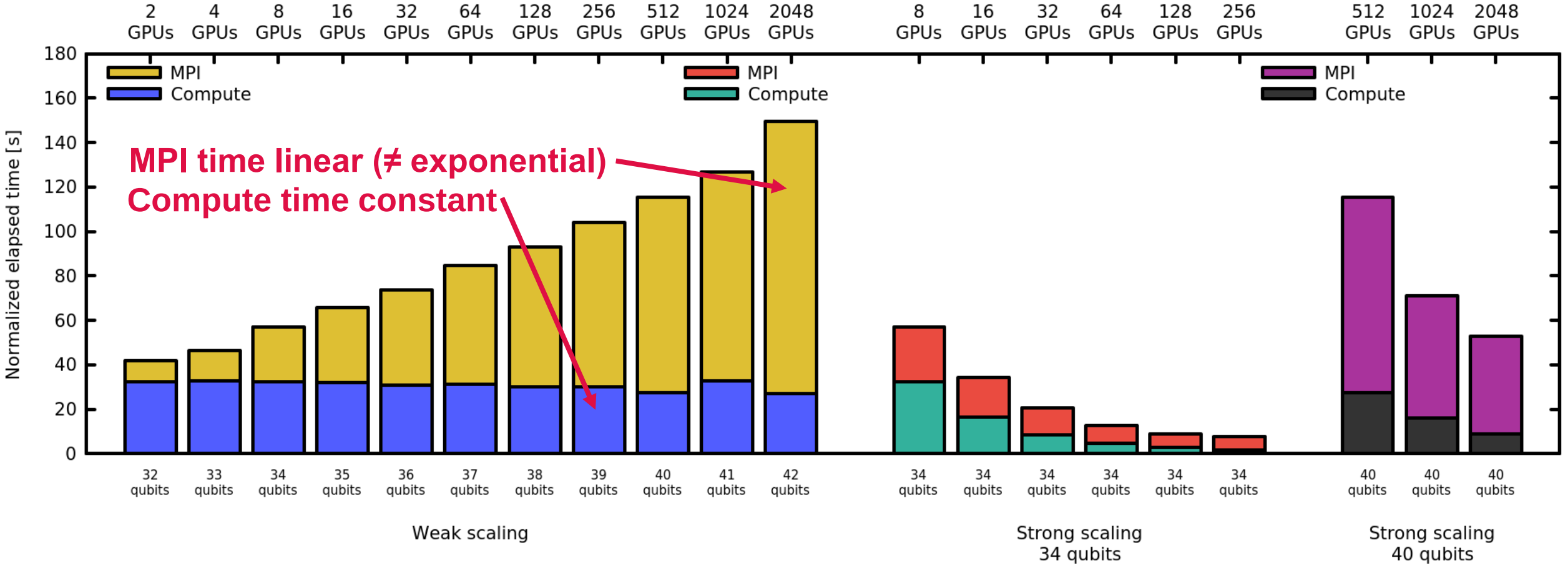
Weak and strong scaling results: Hadamard benchmark circuits



➢ **Speedup** (compute time per node)**:** JUWELS Cluster → JUWELS GPUs (**NVIDIA V100**): 10

JUQCS-G

Weak and strong scaling results: Hadamard benchmark circuits

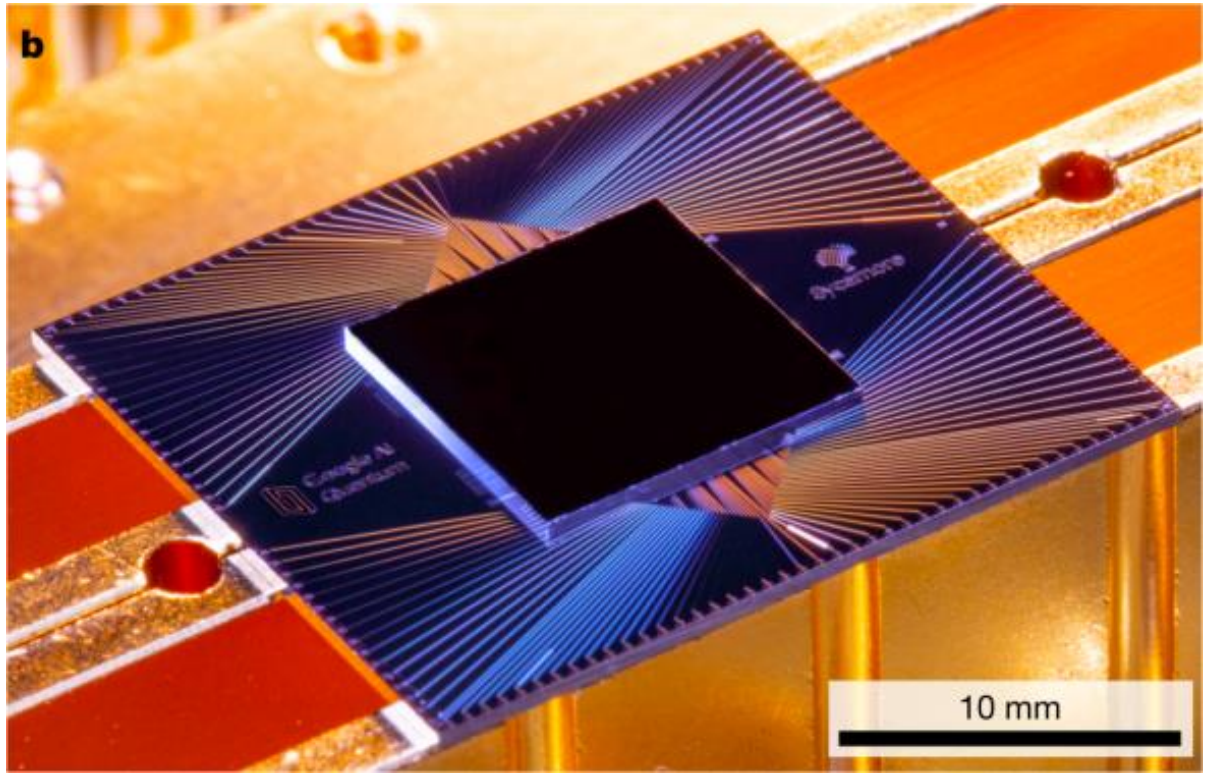


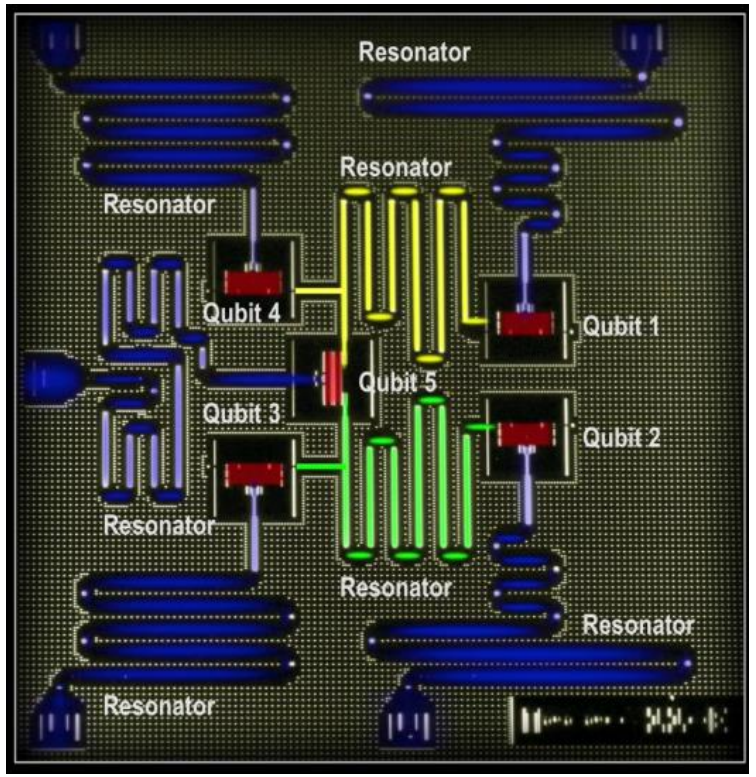
- **Speedup** (compute time per node)**:** JUWELS Cluster → JUWELS GPUs (**NVIDIA V100**): 10
- **Speedup** (compute time per node)**:** JUWELS GPUs → JUWELS Booster (**NVIDIA A100**): 2 – 3

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++





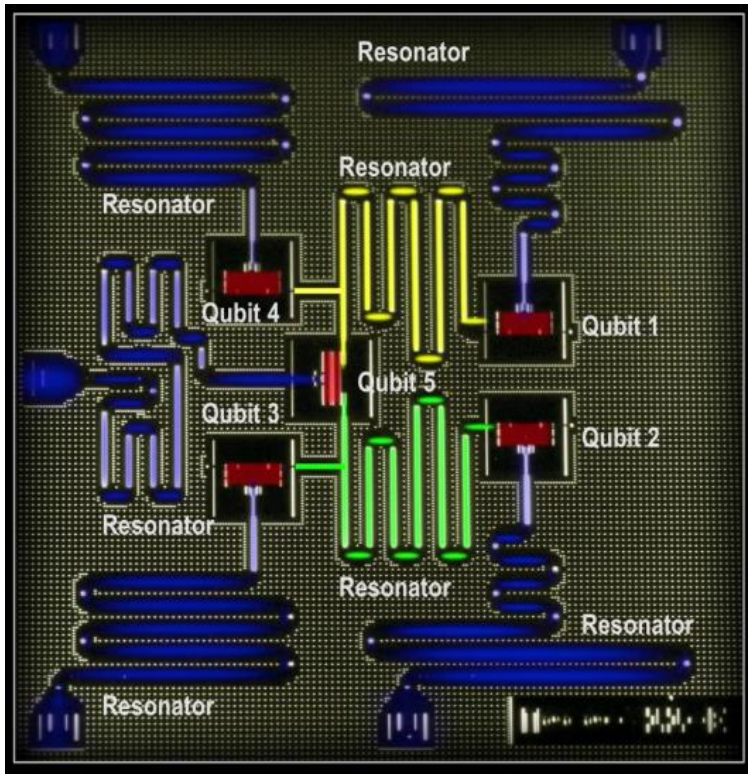
JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

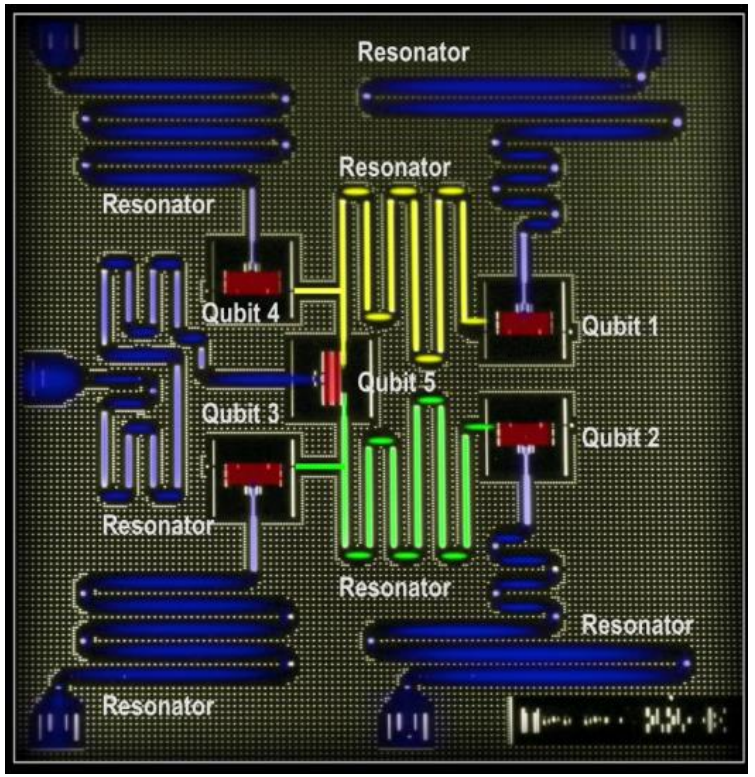
Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

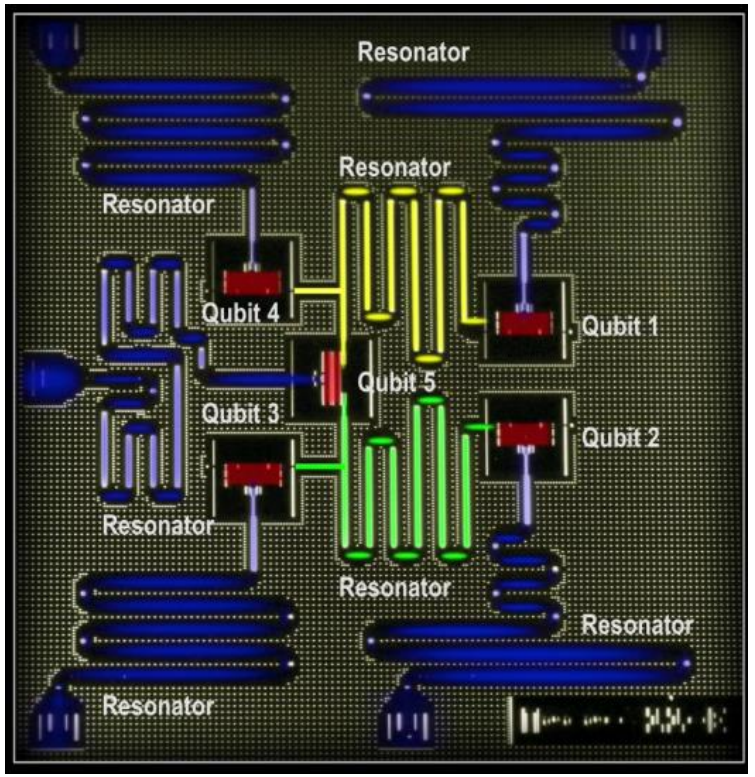
CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

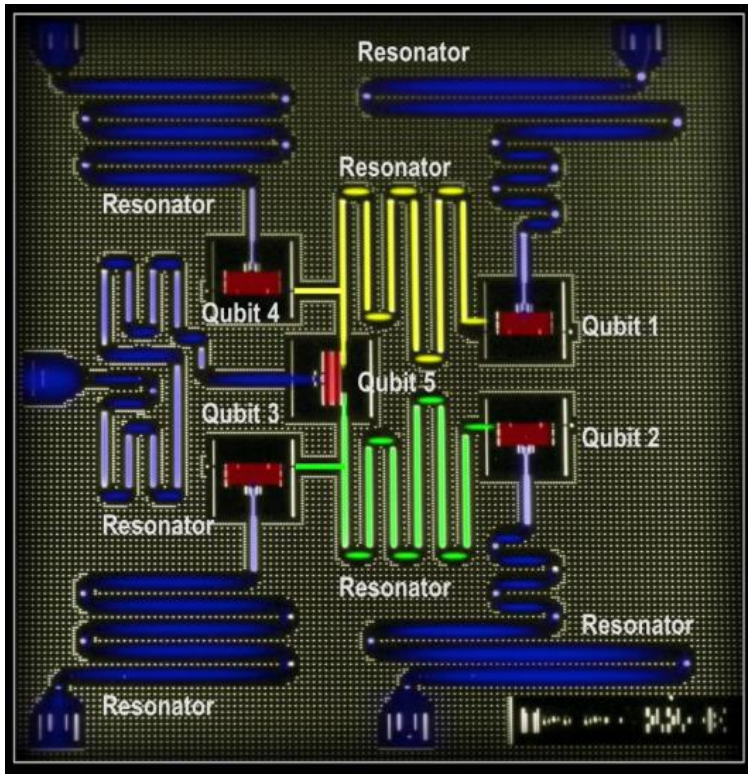
- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

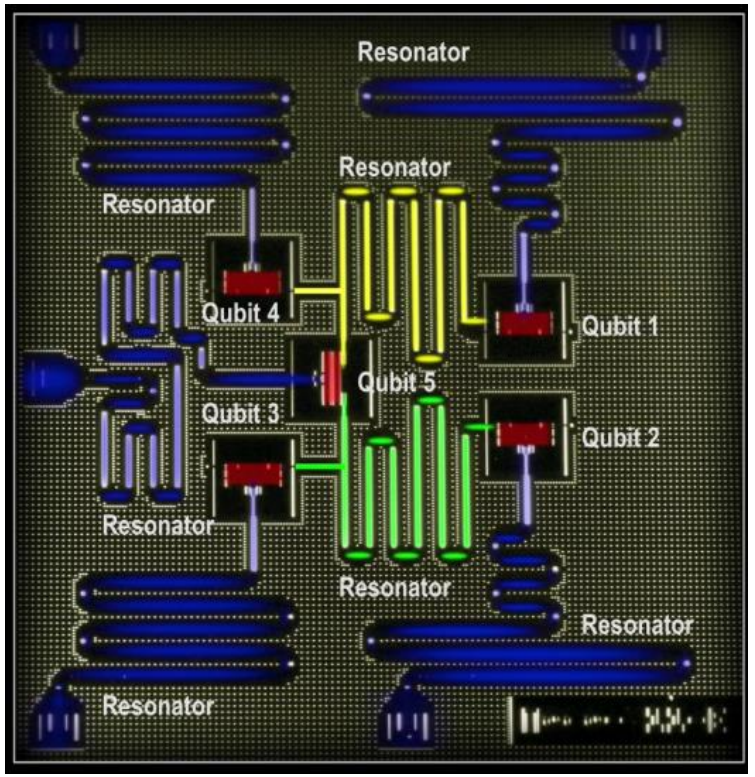
- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs **CUDA+OpenACC MPI C++**

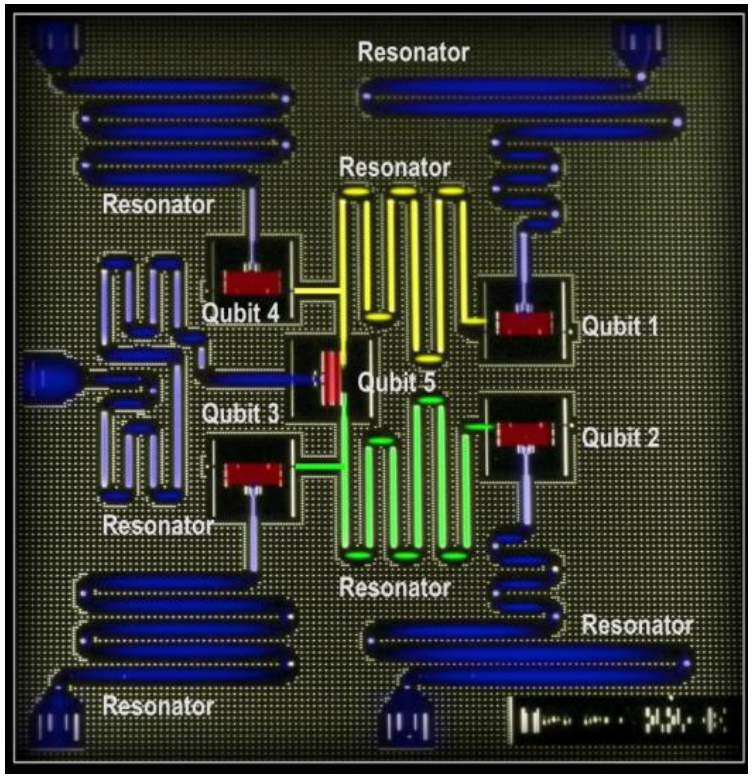
- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$
    - Now: $\rho = (\rho_{k_0 m_0 k_1 m_1})$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

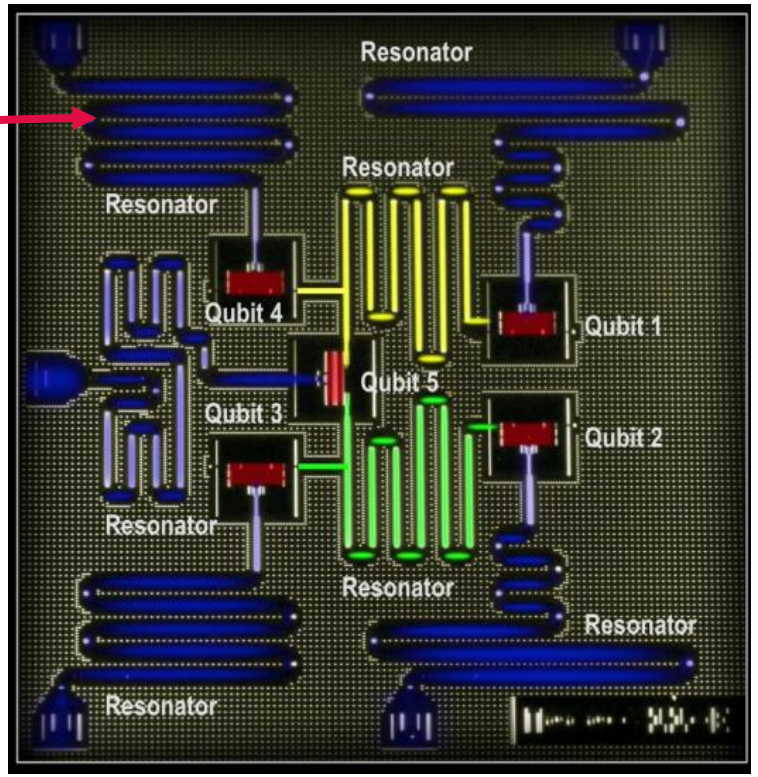
$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$
    - Now: $\rho = (\rho_{k_0 m_0 k_1 m_1})$

$0 \le k_0, k_1 < 1000$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

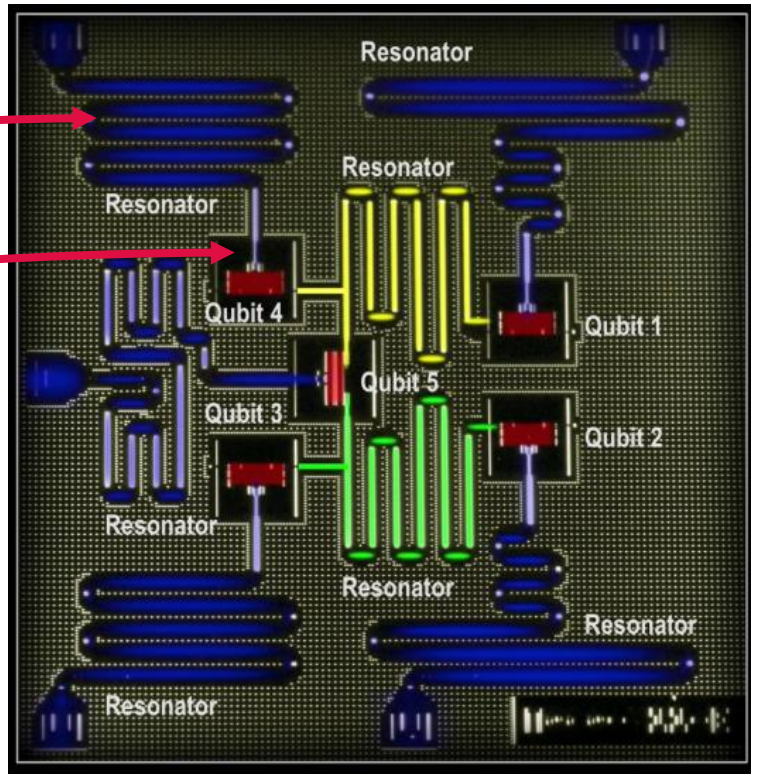
- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$
    - Now: $\rho = (\rho_{k_0 m_0 k_1 m_1})$

$0 \le k_0, k_1 < 1000$

$0 \le m_0, m_1 < 4 \text{ to } 12$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

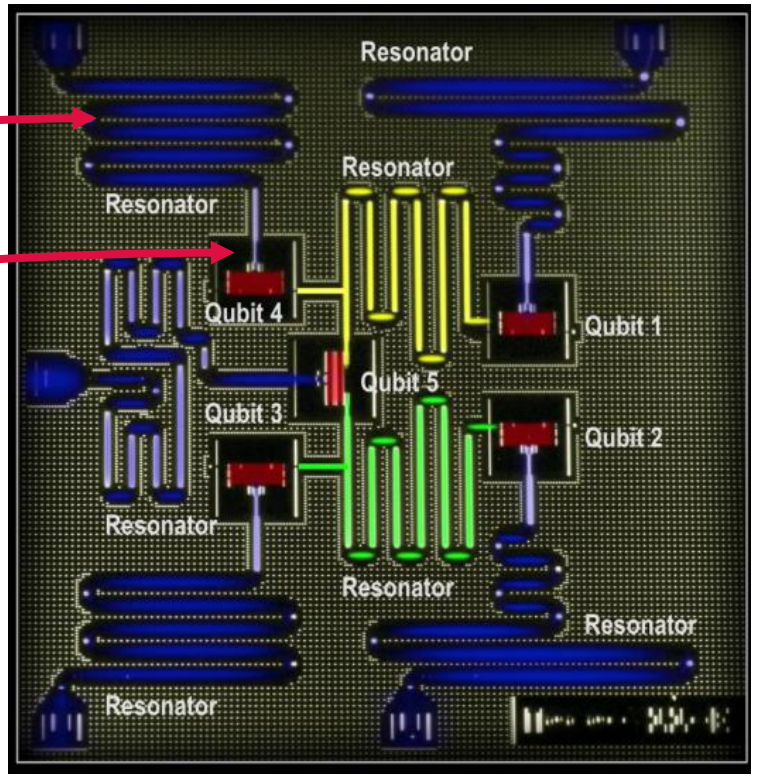
- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$
    - Now: $\rho = (\rho_{k_0 m_0 k_1 m_1})$
  - More complicated sparse matrices (sin, sinh, cos, cosh, exp, ...)

$0 \le k_0, k_1 < 1000$

$0 \le m_0, m_1 < 4 \text{ to } 12$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$
    - Now: $\rho = (\rho_{k_0 m_0 k_1 m_1})$
  - More complicated sparse matrices (sin, sinh, cos, cosh, exp, ...)

$0 \le k_0, k_1 < 1000$

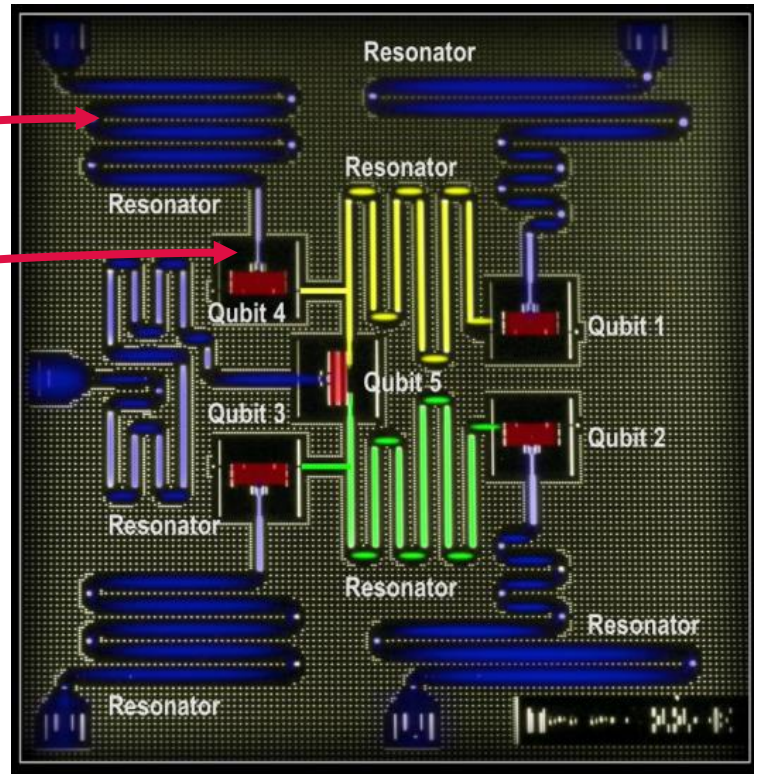
$0 \le m_0, m_1 < 4 \text{ to } 12$

Before:

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Now:

$$\begin{pmatrix} \tilde{c}_{k_0 m_0} & -i\tilde{s}_{k_0 m_0} \\ -i\tilde{s}_{k_0 m_0} & \tilde{c}_{k_0 m_0} \end{pmatrix}$$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

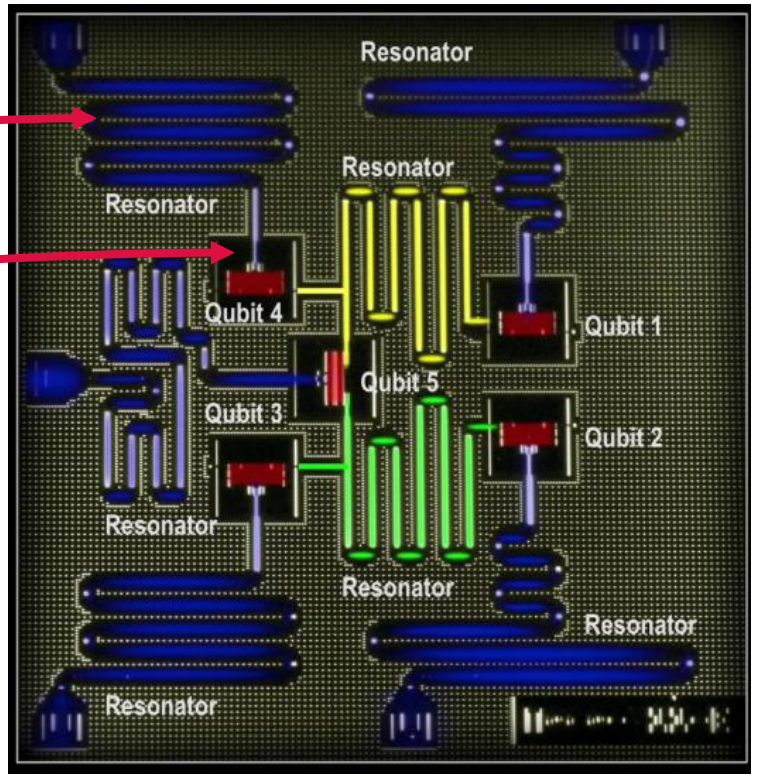
- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$
    - Now: $\rho = (\rho_{k_0 m_0 k_1 m_1})$
  - More complicated sparse matrices (sin, sinh, cos, cosh, exp, ...)



$0 \le k_0, k_1 < 1000$

$0 \le m_0, m_1 < 4 \text{ to } 12$

Before:

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Now:

$$\begin{pmatrix} \tilde{c}_{k_0 m_0} & -i\tilde{s}_{k_0 m_0} \\ -i\tilde{s}_{k_0 m_0} & \tilde{c}_{k_0 m_0} \end{pmatrix}$$

$$\tilde{c}_{k_0 m_0} = \cos(\tau\sqrt{k_0+1}(G\lambda_{m_0} + \varepsilon(\tilde{t})))$$

$$\tilde{s}_{k_0 m_0} = \sin(\tau\sqrt{k_0+1}(G\lambda_{m_0} + \varepsilon(\tilde{t})))$$

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$
    - Now: $\rho = (\rho_{k_0 m_0 k_1 m_1})$
  - More complicated sparse matrices (sin, sinh, cos, cosh, exp, ...)
- Very computation-intensive (memory “only” 2 GiB)

$0 \le k_0, k_1 < 1000$

$0 \le m_0, m_1 < 4 \text{ to } 12$

Before:

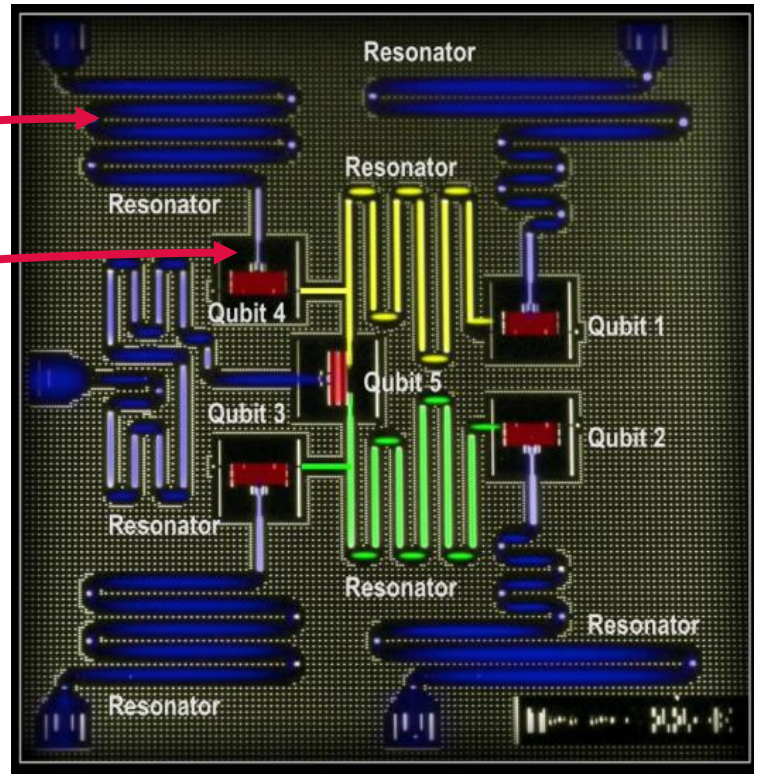
$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Now:

$$\begin{pmatrix} \tilde{c}_{k_0 m_0} & -i\tilde{s}_{k_0 m_0} \\ -i\tilde{s}_{k_0 m_0} & \tilde{c}_{k_0 m_0} \end{pmatrix}$$

$$\tilde{c}_{k_0 m_0} = \cos(\tau\sqrt{k_0+1}(G\lambda_{m_0} + \varepsilon(\tilde{t})))$$

$$\tilde{s}_{k_0 m_0} = \sin(\tau\sqrt{k_0+1}(G\lambda_{m_0} + \varepsilon(\tilde{t})))$$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

- Physical realization: Solve Schrödinger / Master Equation

$$\frac{\partial}{\partial t}|\psi\rangle = -iH|\psi\rangle \quad \text{or} \quad \frac{\partial}{\partial t}\rho = -i[H,\rho] + \mathcal{D}[\rho] = \mathcal{L}[\rho]$$

- Similar SPMV updates **but:**
  - Many updates per time step

$$\rho(t+\tau) = e^{\mathcal{L}(t+\tau/2)}\rho(t)$$

  - More than 2 states per subsystem
    - Before: $|\psi\rangle = (\psi_{q_3 q_2 q_1 q_0})$
    - Now: $\rho = (\rho_{k_0 m_0 k_1 m_1})$
  - More complicated sparse matrices (sin, sinh, cos, cosh, exp, ...)
- Very computation-intensive (memory “only” 2 GiB)

→ Useful to measure single-GPU performance

$0 \le k_0, k_1 < 1000$

$0 \le m_0, m_1 < 4 \text{ to } 12$

Before:

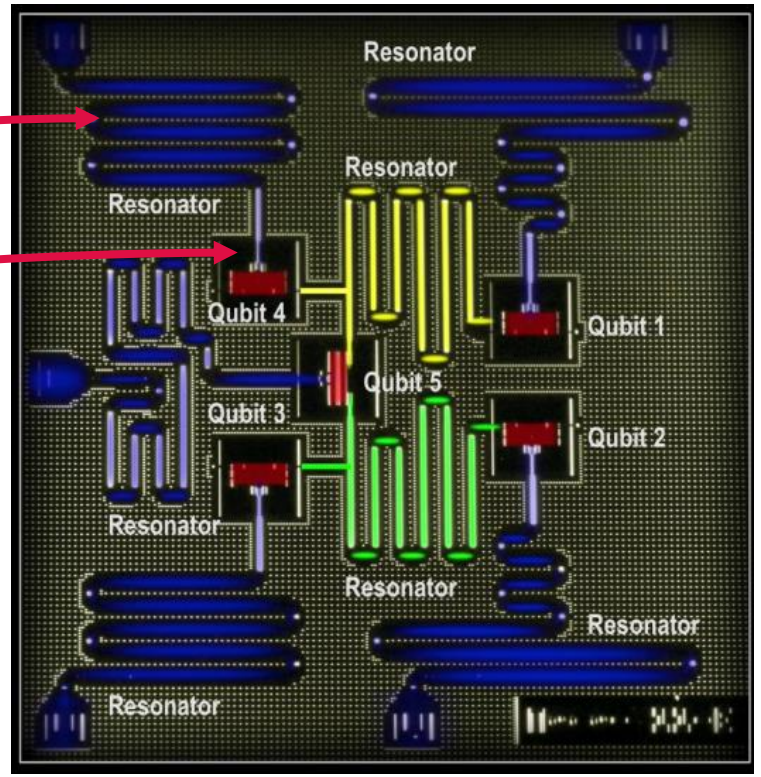
$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Now:

$$\begin{pmatrix} \tilde{c}_{k_0 m_0} & -i\tilde{s}_{k_0 m_0} \\ -i\tilde{s}_{k_0 m_0} & \tilde{c}_{k_0 m_0} \end{pmatrix}$$

$$\tilde{c}_{k_0 m_0} = \cos(\tau\sqrt{k_0+1}(G\lambda_{m_0} + \varepsilon(\tilde{t})))$$

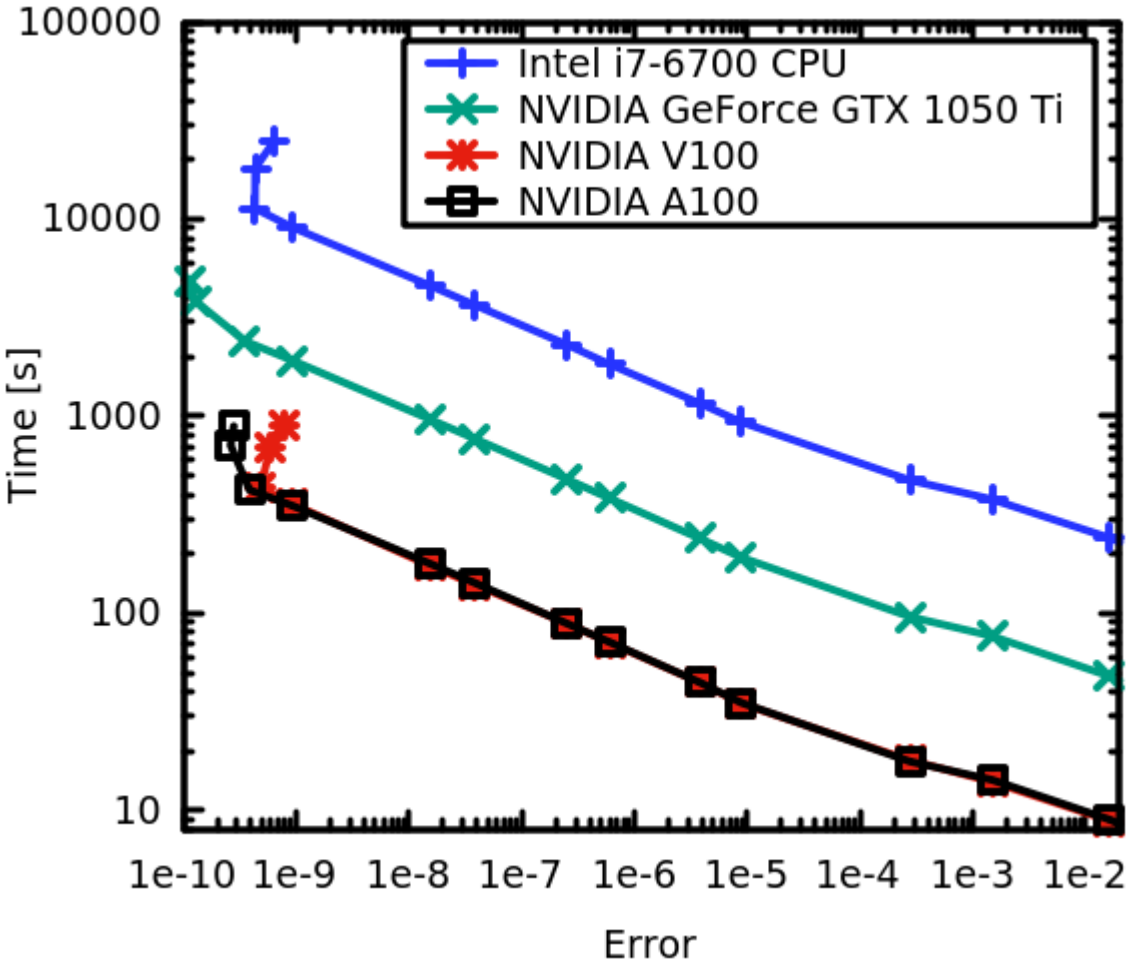
$$\tilde{s}_{k_0 m_0} = \sin(\tau\sqrt{k_0+1}(G\lambda_{m_0} + \varepsilon(\tilde{t})))$$



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

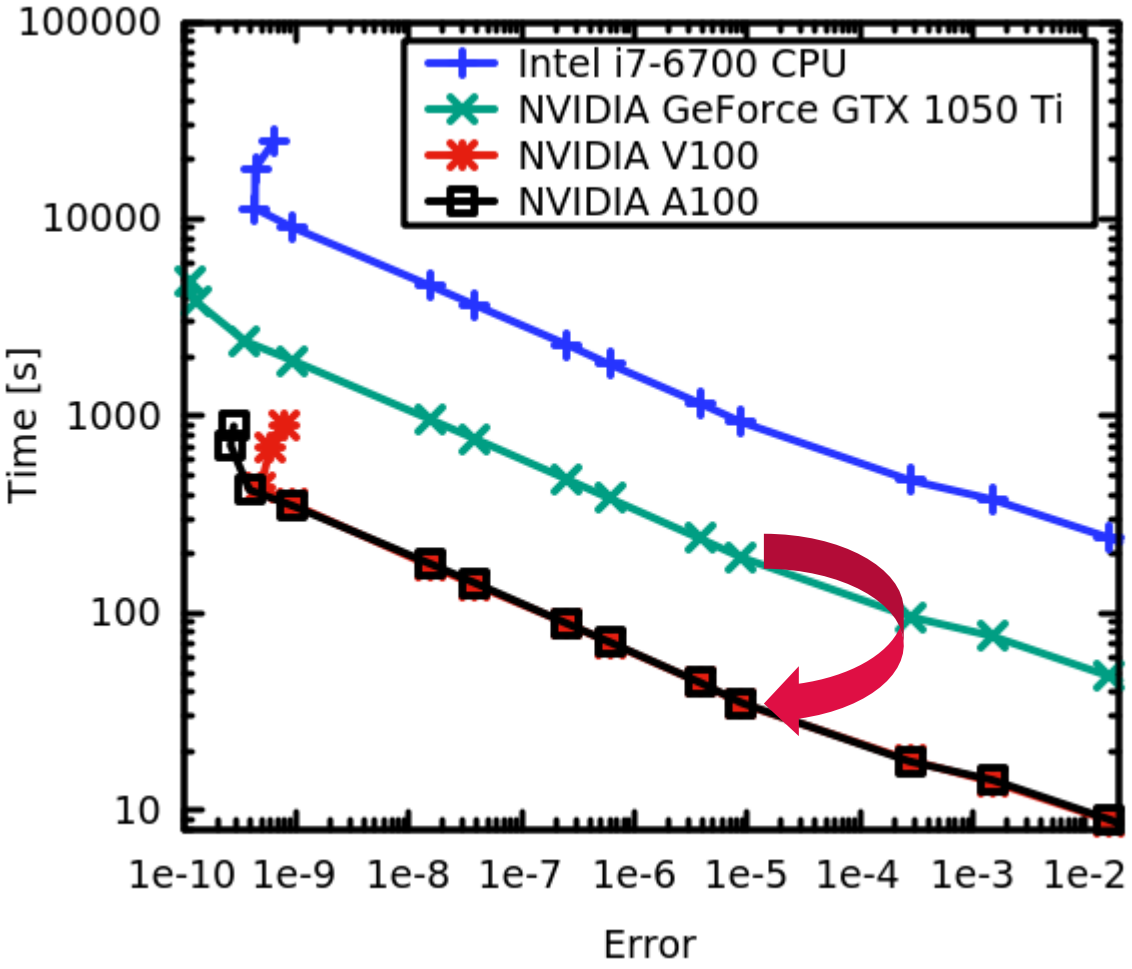
CUDA+OpenACC MPI C++



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

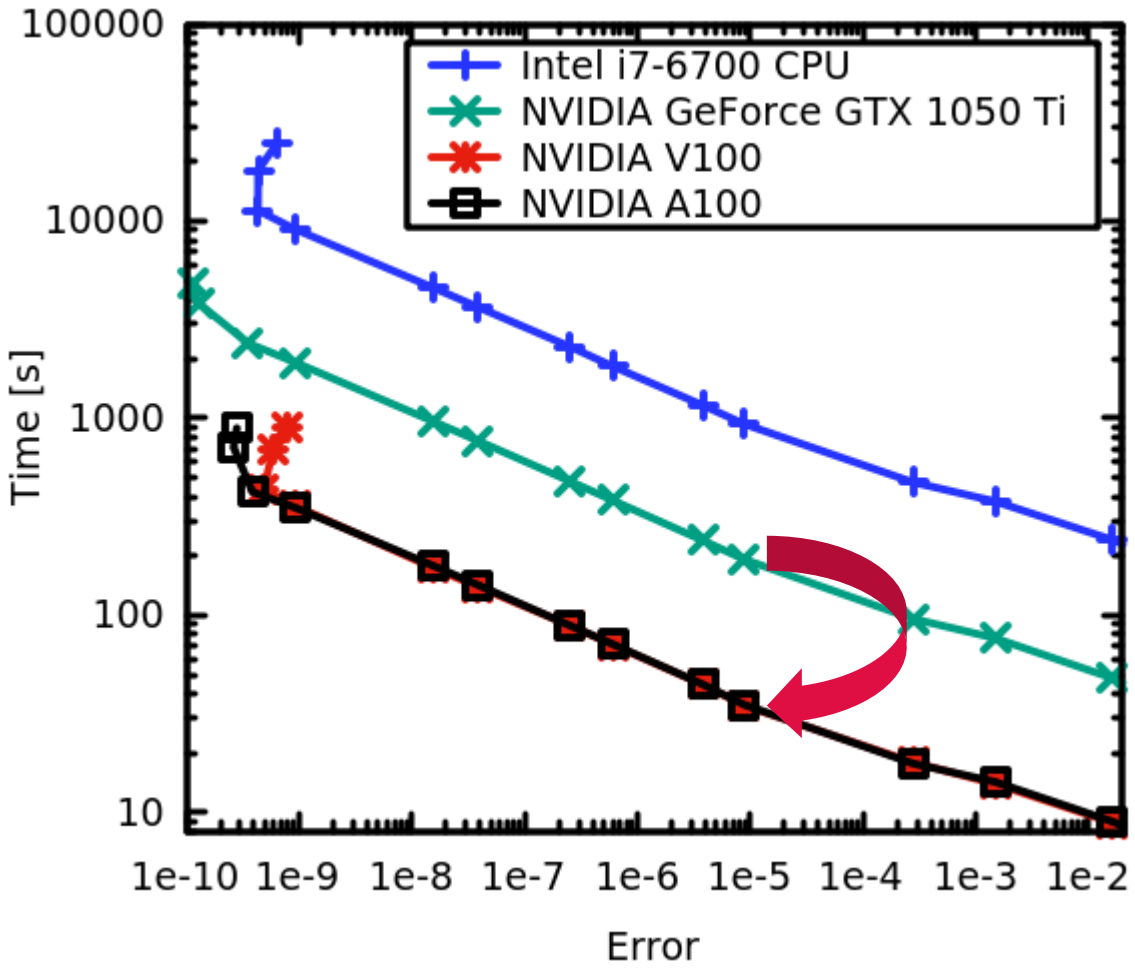


Bottleneck kernels profit from **Tensor Cores**

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++



Bottleneck kernels profit from **Tensor Cores**

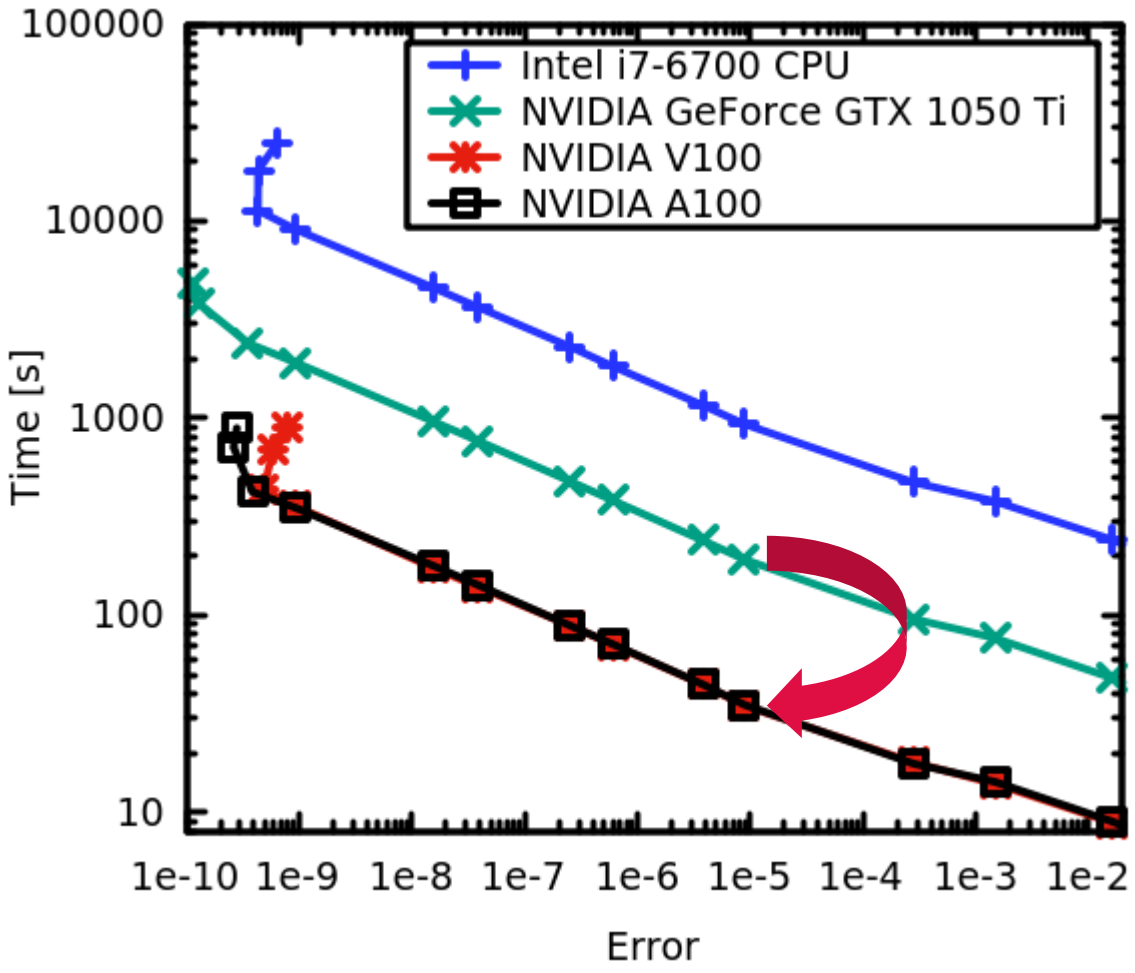
➢ For each k_0 in steps of 2, for each k_1, m_0, m_1

$$\begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix} \leftarrow \begin{pmatrix} c_{k_0,k_1} & s^{-}_{k_0,k_1} \\ s^{+}_{k_0,k_1} & c_{k_0,k_1} \end{pmatrix} \begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix}$$

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++



Bottleneck kernels profit from **Tensor Cores**

➢ For each k_0 in steps of 2, for each k_1, m_0, m_1

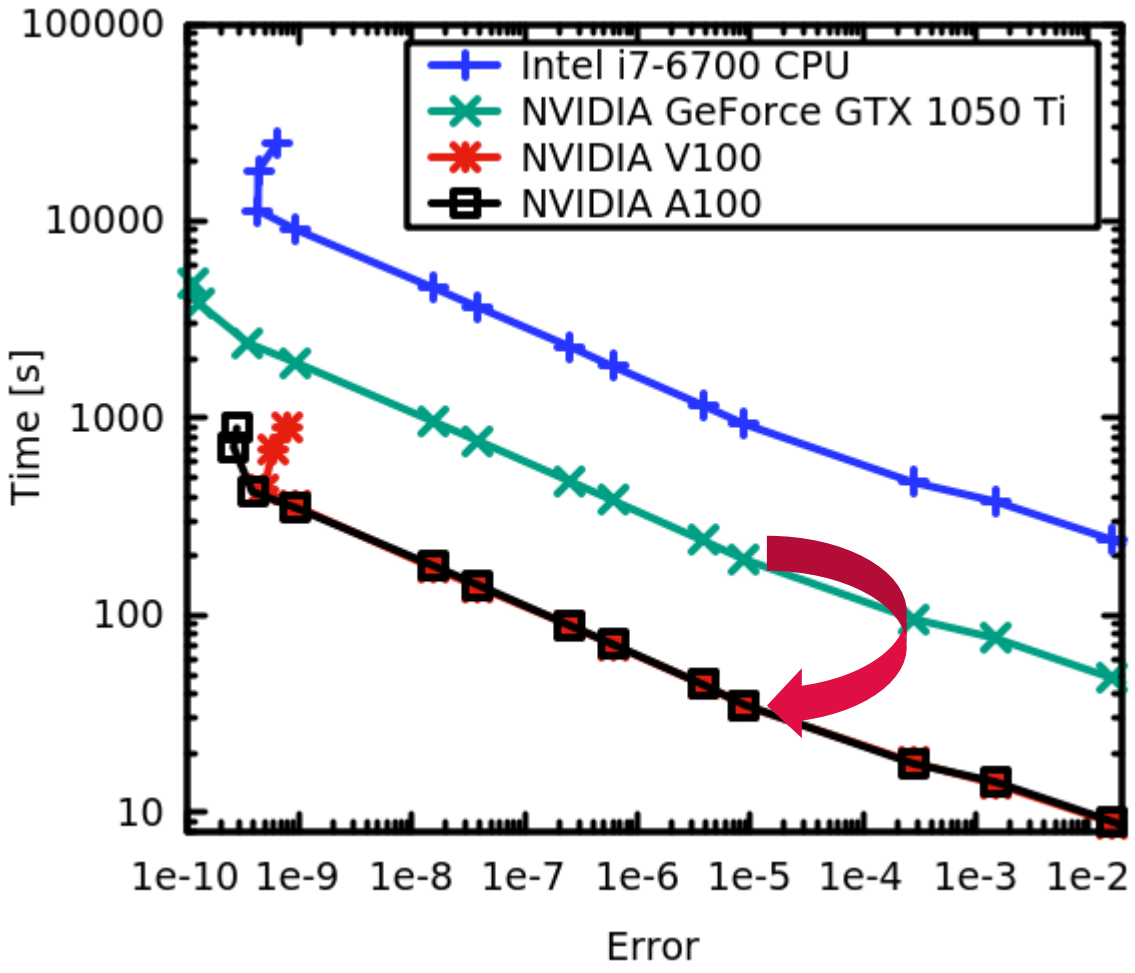
$$\begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix} \leftarrow \begin{pmatrix} c_{k_0,k_1} & s^{-}_{k_0,k_1} \\ s^{+}_{k_0,k_1} & c_{k_0,k_1} \end{pmatrix} \begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix}$$

→ **2-component updates**

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++



Bottleneck kernels profit from **Tensor Cores**

- For each k_0 in steps of 2, for each k_1, m_0, m_1

$$\begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix} \leftarrow \begin{pmatrix} c_{k_0,k_1} & s^-_{k_0,k_1} \\ s^+_{k_0,k_1} & c_{k_0,k_1} \end{pmatrix} \begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix}$$

→ **2-component updates**

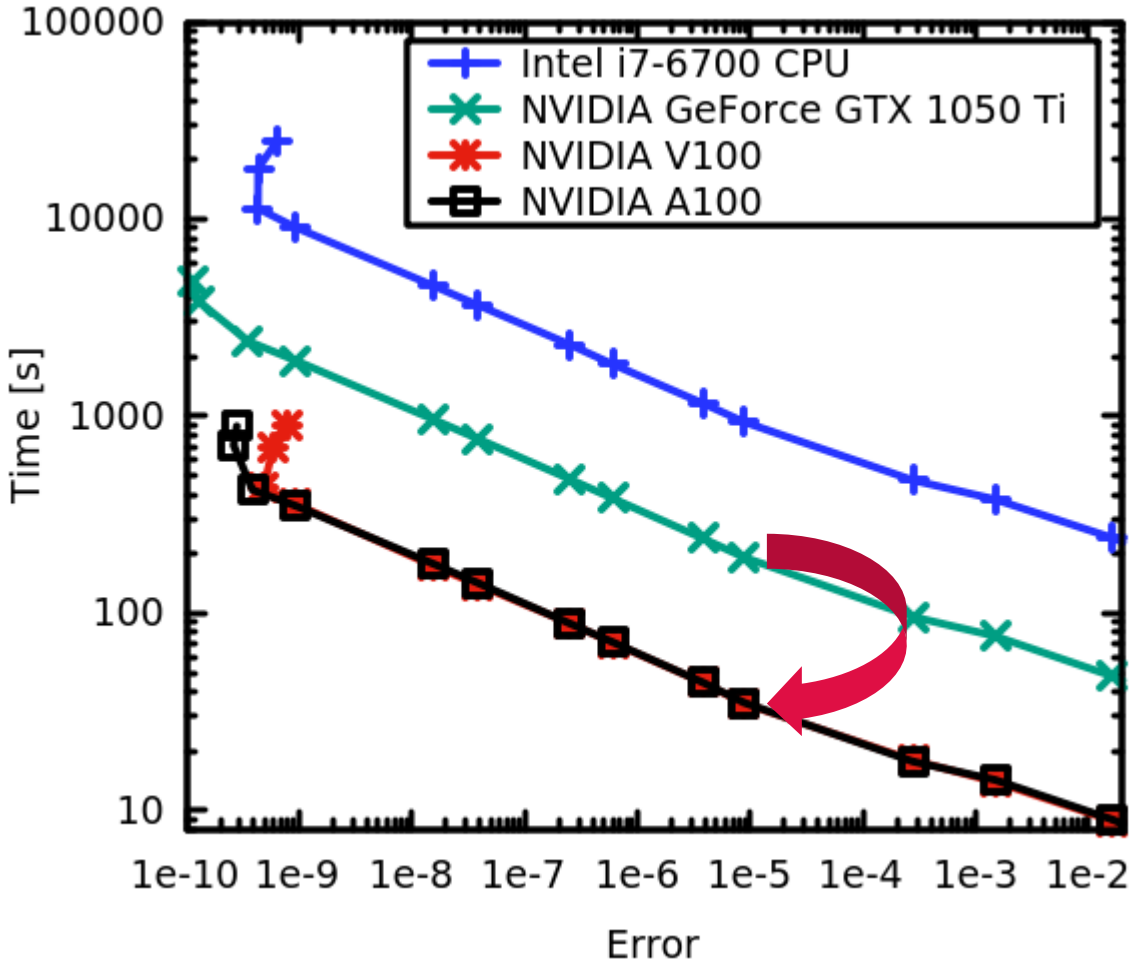
- For each k_0, k_1, m_0

$$\rho_{k_0 m_0 k_1 m_1} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \, \rho_{k_0 m_0' k_1 m_1}$$

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++



Bottleneck kernels profit from **Tensor Cores**

- For each k_0 in steps of 2, for each k_1, m_0, m_1

$$\begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix} \leftarrow \begin{pmatrix} c_{k_0,k_1} & s^-_{k_0,k_1} \\ s^+_{k_0,k_1} & c_{k_0,k_1} \end{pmatrix} \begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix}$$

→ **2-component updates**

- For each k_0, k_1, m_0

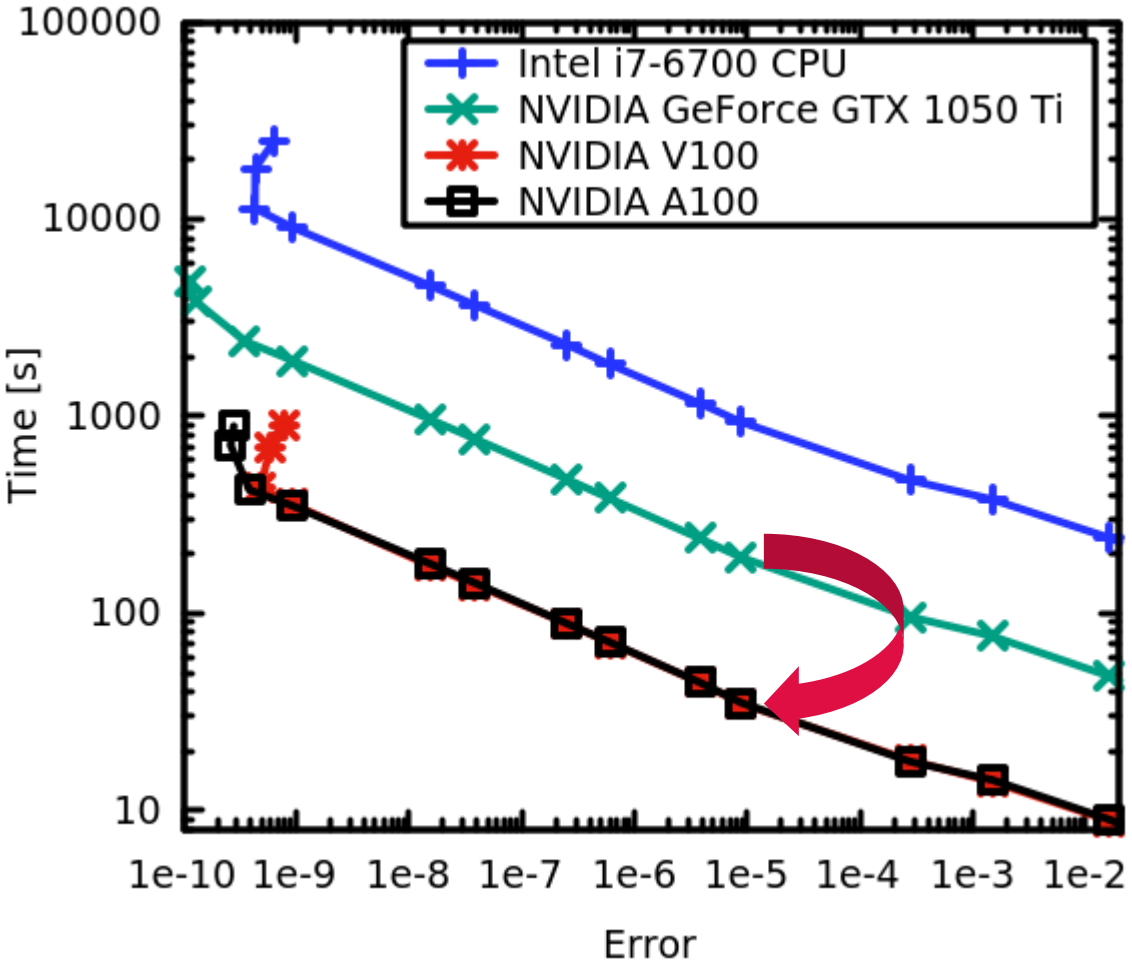
$$\rho_{k_0 m_0 k_1 m_1} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \, \rho_{k_0 m_0' k_1 m_1}$$

→ **M-component updates**

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++



Bottleneck kernels profit from **Tensor Cores**

➢ For each k_0 in steps of 2, for each k_1, m_0, m_1

$$\begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix} \leftarrow \begin{pmatrix} c_{k_0,k_1} & s^-_{k_0,k_1} \\ s^+_{k_0,k_1} & c_{k_0,k_1} \end{pmatrix} \begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix}$$

→ **2-component updates**

➢ For each k_0, k_1, m_0

$$\rho_{k_0 m_0 k_1 m_1} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \, \rho_{k_0 m_0' k_1 m_1}$$

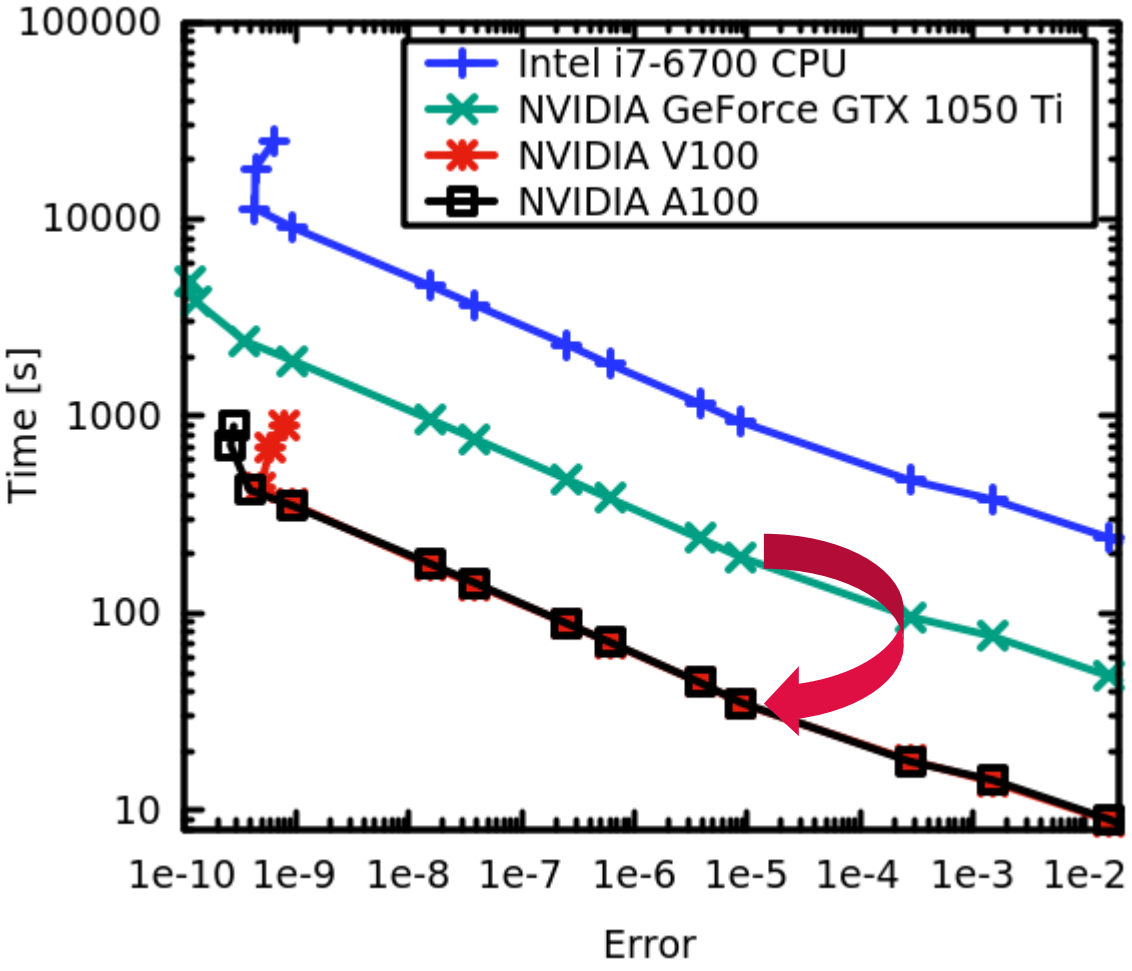
→ **M-component updates**

Small system with **M=4** and **K=100**

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++



Bottleneck kernels profit from **Tensor Cores**

➢ For each k_0 in steps of 2, for each k_1, m_0, m_1

$$\begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix} \leftarrow \begin{pmatrix} c_{k_0,k_1} & s^-_{k_0,k_1} \\ s^+_{k_0,k_1} & c_{k_0,k_1} \end{pmatrix} \begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix}$$

→ **2-component updates**

➢ For each k_0, k_1, m_0

$$\rho_{k_0 m_0 k_1 m_1} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \, \rho_{k_0 m_0' k_1 m_1}$$

→ **M-component updates**

Small system with **M=4** and **K=100**

→ Similar performance for **V100** and **A100**

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

Realistic system with **M=12** and **K=400**

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs CUDA+OpenACC MPI C++

Realistic system with **M=12** and **K=400**

For each k_0, k_1, m_0

$$\rho_{k_0 m_0 k_1 m_1} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \, \rho_{k_0 m_0' k_1 m_1}$$

12-component updates

JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

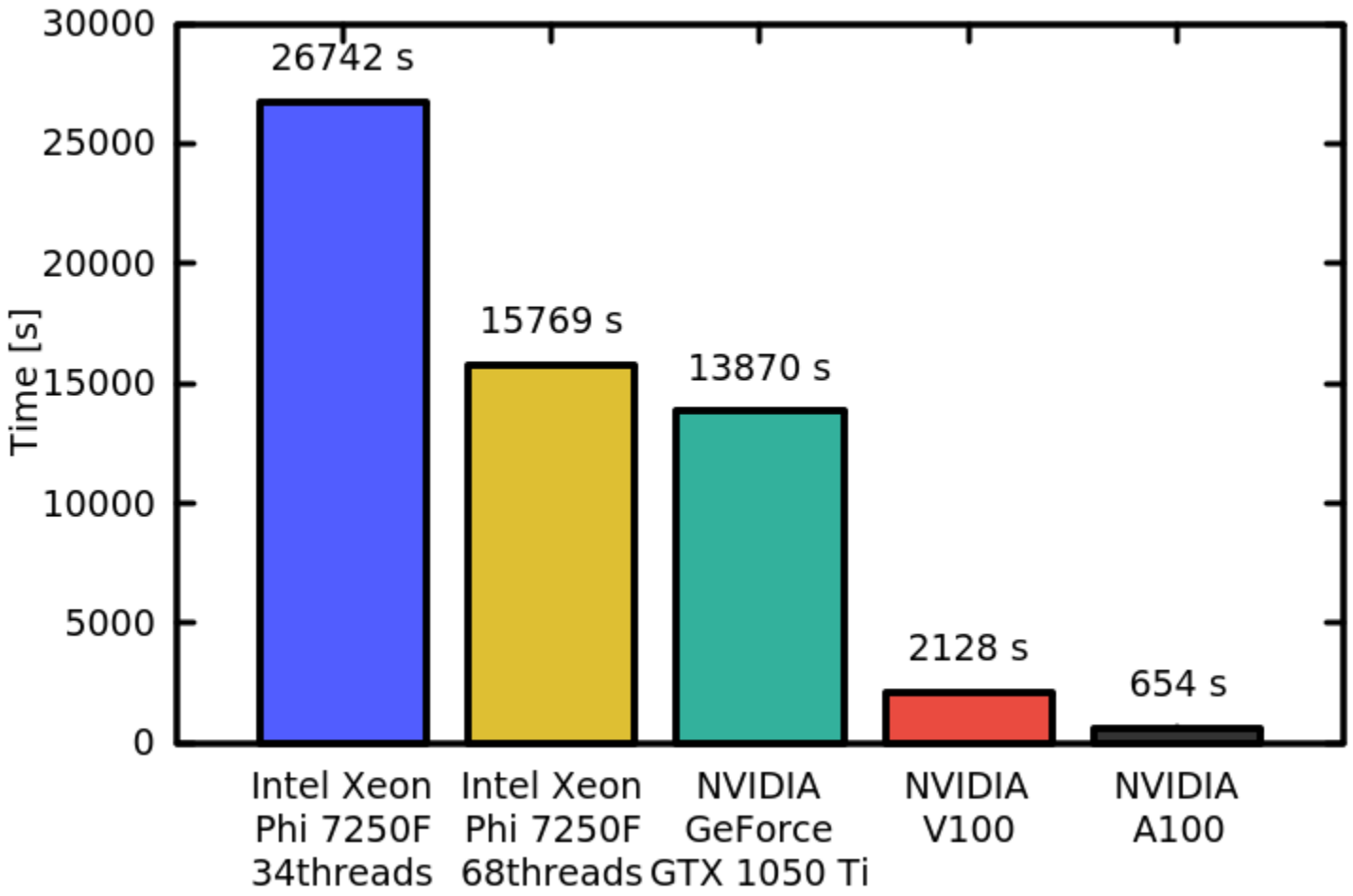
CUDA+OpenACC MPI C++

Realistic system with **M=12** and **K=400**

For each k_0, k_1, m_0

$$\rho_{k_0 m_0 k_1 m_1} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \, \rho_{k_0 m_0' k_1 m_1}$$

12-component updates



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

Simulating physical realizations of quantum computers on GPUs

CUDA+OpenACC MPI C++

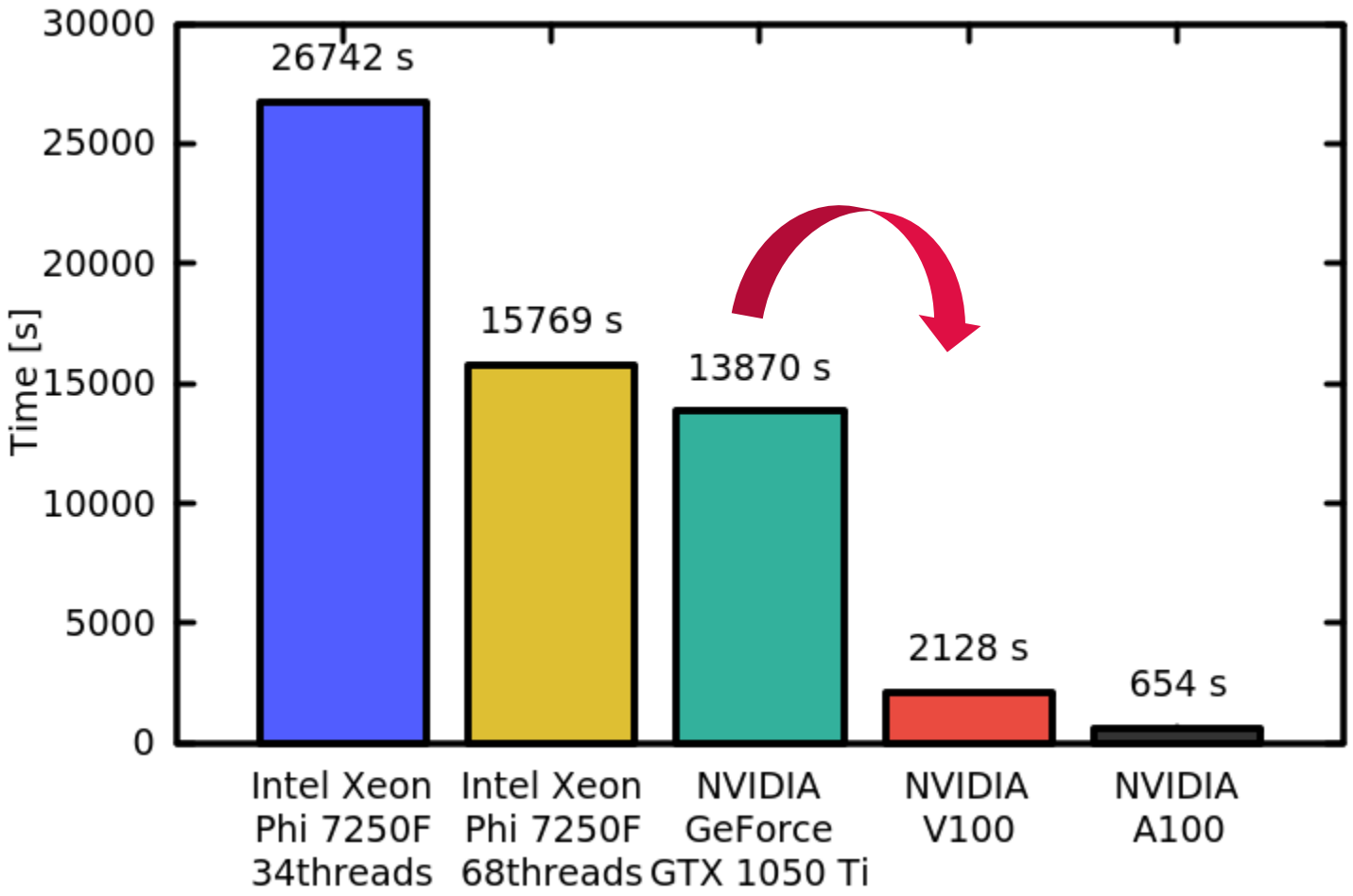
Realistic system with **M=12** and **K=400**

For each k_0, k_1, m_0

$$\rho_{k_0 m_0 k_1 m_1} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \, \rho_{k_0 m_0' k_1 m_1}$$

12-component updates

→ Similar speedup until **V100**



JUQMES: QUANTUM MASTER EQUATION SIMULATOR

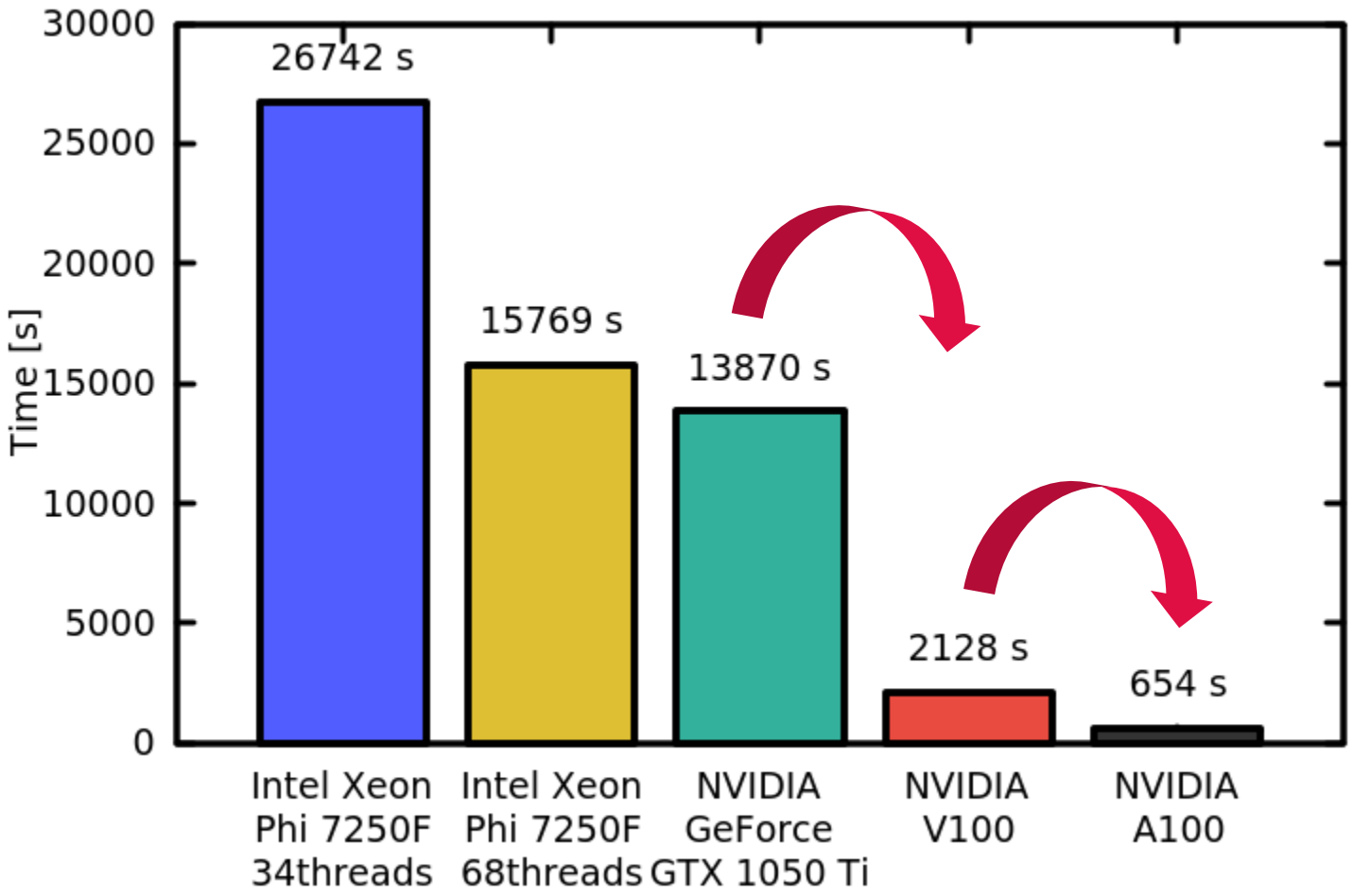
Simulating physical realizations of quantum computers on GPUs CUDA+OpenACC MPI C++

Realistic system with **M=12** and **K=400**

For each k_0, k_1, m_0

$$\rho_{k_0 m_0 k_1 m_1} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \, \rho_{k_0 m_0' k_1 m_1}$$

12-component updates



→ Similar speedup until **V100**

→ Further speedup from **V100** to **A100**

SUMMARY

SUMMARY

- All QC simulations are composed of

  huge **sparse** **complex**

  matrix-vector updates

SUMMARY

- All QC simulations are composed of

  huge **sparse** **complex**

  matrix-vector updates

- Simulating QCs is a versatile approach to benchmark supercomputers

SUMMARY

- All QC simulations are composed of

  huge **sparse** **complex**

  matrix-vector updates
- Simulating QCs is a versatile approach to benchmark supercomputers
  - Memory-, network-, and computation-intensive

SUMMARY

- All QC simulations are composed of

  huge **sparse** **complex**

  matrix-vector updates
- Simulating QCs is a versatile approach to benchmark supercomputers
  - Memory-, network-, and computation-intensive
- Huge speedup from CPU-based simulators to GPU-based simulators

SUMMARY

- All QC simulations are composed of

  huge **sparse** **complex**

  matrix-vector updates
- Simulating QCs is a versatile approach to benchmark supercomputers
  - Memory-, network-, and computation-intensive
- Huge speedup from CPU-based simulators to GPU-based simulators

THANK YOU FOR YOUR ATTENTION

- More information and references:
  - **MPI communication scheme:** De Raedt et al., Comp. Phys. Commun. 176, 121 (2007)
  - **Benchmarking gate-based quantum computers:** Michielsen et al., Comp. Phys. Commun. 220, 44 (2017)
  - **JUQCS:** De Raedt et al., Comp. Phys. Commun. 237, 41 (2019)
  - **Quantum supremacy with JUQCS:** Arute et al., Nature 574, 505 (2019)
  - **Benchmarking supercomputers with JUQCS:** Willsch et al., NIC Series 50, 255 (2020)
  - **GPU-accelerated simulations of QA and the QAOA:** Willsch et al., preprint available on arXiv (2021)

Q & A

What are the fundamental tensor operations in all simulations of this kind?

Q & A

What are the fundamental tensor operations in all simulations of this kind?

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2'} H_{q_2 q_2'} \psi_{\cdots q_3 q_2' q_1 q_0}$$

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2' q_0'} U_{q_2 q_2' q_0 q_0'} \psi_{\cdots q_3 q_2' q_1 q_0'}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \rho_{k_0 m_0' k_1 m_1 \cdots}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{k_0' k_1'} W_{k_0 k_0' k_1 k_1'} \rho_{k_0' m_0 k_1' m_1 \cdots}$$

Q & A

What are the fundamental tensor operations in all simulations of this kind?

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2'} H_{q_2 q_2'} \psi_{\cdots q_3 q_2' q_1 q_0}$$

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2' q_0'} U_{q_2 q_2' q_0 q_0'} \psi_{\cdots q_3 q_2' q_1 q_0'}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \rho_{k_0 m_0' k_1 m_1 \cdots}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{k_0' k_1'} W_{k_0 k_0' k_1 k_1'} \rho_{k_0' m_0 k_1' m_1 \cdots}$$

$$\sum_{k_1'} \begin{pmatrix} 1 & & & & \\ & c_{11} & -is_{12} & & \\ & -is_{21} & c_{22} & & \\ & & & \ddots & \\ & & & c_{(K-2)(K-2)} & -is_{(K-2)(K-1)} \\ & & & -is_{(K-2)(K-1)} & c_{(K-1)(K-1)} \end{pmatrix}_{k_1 k_1'} \rho_{k_0 m_0 k_1' m_1 \cdots}$$

Q & A

What are the fundamental tensor operations in all simulations of this kind?

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2'} H_{q_2 q_2'} \psi_{\cdots q_3 q_2' q_1 q_0}$$

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2' q_0'} U_{q_2 q_2' q_0 q_0'} \psi_{\cdots q_3 q_2' q_1 q_0'}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \rho_{k_0 m_0' k_1 m_1 \cdots}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{k_0' k_1'} W_{k_0 k_0' k_1 k_1'} \rho_{k_0' m_0 k_1' m_1 \cdots}$$

$$\sum_{k_1'} \begin{pmatrix} 1 & & & & \\ & c_{11} & -is_{12} & & \\ & -is_{21} & c_{22} & & \\ & & & \ddots & \\ & & & c_{(K-2)(K-2)} & -is_{(K-2)(K-1)} \\ & & & -is_{(K-2)(K-1)} & c_{(K-1)(K-1)} \end{pmatrix}_{k_1 k_1'} \rho_{k_0 m_0 k_1' m_1 \cdots}$$

For each k_0 in steps of 2, for each k_1, m_0, m_1

$$\begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix} \leftarrow \begin{pmatrix} c_{k_0,k_1} & s^{-}_{k_0,k_1} \\ s^{+}_{k_0,k_1} & c_{k_0,k_1} \end{pmatrix} \begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix}$$

Q & A

What are the fundamental tensor operations in all simulations of this kind?

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2'} H_{q_2 q_2'} \psi_{\cdots q_3 q_2' q_1 q_0}$$

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2' q_0'} U_{q_2 q_2' q_0 q_0'} \psi_{\cdots q_3 q_2' q_1 q_0'}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \rho_{k_0 m_0' k_1 m_1 \cdots}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{k_0' k_1'} W_{k_0 k_0' k_1 k_1'} \rho_{k_0' m_0 k_1' m_1 \cdots}$$

$$\sum_{k_1'} \begin{pmatrix} 1 & & & & \\ & c_{11} & -is_{12} & & \\ & -is_{21} & c_{22} & & \\ & & & \ddots & \\ & & & c_{(K-2)(K-2)} & -is_{(K-2)(K-1)} \\ & & & -is_{(K-2)(K-1)} & c_{(K-1)(K-1)} \end{pmatrix}_{k_1 k_1'} \rho_{k_0 m_0 k_1' m_1 \cdots}$$

For each k_0 in steps of 2, for each k_1, m_0, m_1

$$\begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix} \leftarrow \begin{pmatrix} c_{k_0,k_1} & s^{-}_{k_0,k_1} \\ s^{+}_{k_0,k_1} & c_{k_0,k_1} \end{pmatrix} \begin{pmatrix} \rho_{k_0 m_0 k_1 m_1} \\ \rho_{(k_0+1) m_0 (k_1+1) m_1} \end{pmatrix}$$

→ **In-place double complex 2D SPMV updates**

Q & A

What dedicated library functions would be useful for quantum computer simulations?

Q & A

What dedicated library functions would be useful for quantum computer simulations?

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2'} H_{q_2 q_2'} \psi_{\cdots q_3 q_2' q_1 q_0}$$

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2' q_0'} U_{q_2 q_2' q_0 q_0'} \psi_{\cdots q_3 q_2' q_1 q_0'}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \rho_{k_0 m_0' k_1 m_1 \cdots}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{k_0' k_1'} W_{k_0 k_0' k_1 k_1'} \rho_{k_0' m_0 k_1' m_1 \cdots}$$

Q & A

What dedicated library functions would be useful for quantum computer simulations?

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2'} H_{q_2 q_2'} \psi_{\cdots q_3 q_2' q_1 q_0}$$

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2' q_0'} U_{q_2 q_2' q_0 q_0'} \psi_{\cdots q_3 q_2' q_1 q_0'}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \rho_{k_0 m_0' k_1 m_1 \cdots}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{k_0' k_1'} W_{k_0 k_0' k_1 k_1'} \rho_{k_0' m_0 k_1' m_1 \cdots}$$

Generic 1-component and 2-component in-place double complex 2D SPMV updates

Q & A

What dedicated library functions would be useful for quantum computer simulations?

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2'} H_{q_2 q_2'} \psi_{\cdots q_3 q_2' q_1 q_0}$$

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2' q_0'} U_{q_2 q_2' q_0 q_0'} \psi_{\cdots q_3 q_2' q_1 q_0'}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \rho_{k_0 m_0' k_1 m_1 \cdots}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{k_0' k_1'} W_{k_0 k_0' k_1 k_1'} \rho_{k_0' m_0 k_1' m_1 \cdots}$$

Generic 1-component and 2-component in-place double complex 2D SPMV updates

Q & A

What dedicated library functions would be useful for quantum computer simulations?

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2'} H_{q_2 q_2'} \psi_{\cdots q_3 q_2' q_1 q_0}$$

$$\psi_{\cdots q_3 q_2 q_1 q_0} \leftarrow \sum_{q_2' q_0'} U_{q_2 q_2' q_0 q_0'} \psi_{\cdots q_3 q_2' q_1 q_0'}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{m_0'} V_{m_0 m_0'} \rho_{k_0 m_0' k_1 m_1 \cdots}$$

$$\rho_{k_0 m_0 k_1 m_1 \cdots} \leftarrow \sum_{k_0' k_1'} W_{k_0 k_0' k_1 k_1'} \rho_{k_0' m_0 k_1' m_1 \cdots}$$

Generic 1-component and 2-component in-place double complex 2D SPMV updates

Q & A

Do you make use of cuBLAS or other dedicated libraries in your code?

Q & A

Do you make use of cuBLAS or other dedicated libraries in your code?

Yes, but:

Q & A

Do you make use of cuBLAS or other dedicated libraries in your code?

Yes, but:

- **cuBLAS:**

Q & A

Do you make use of cuBLAS or other dedicated libraries in your code?

Yes, but:

- **cuBLAS:**

`cublasZgemm:` $C = \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$

Q & A

Do you make use of cuBLAS or other dedicated libraries in your code?

Yes, but:

- **cuBLAS:**

`cublasZgemm:` $C = \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$

`cublasGemmStridedBatchedEx:`

$$C + i * \mathrm{strideC} = \alpha \mathrm{op}(A + i * \mathrm{strideA})\mathrm{op}(B + i * \mathrm{strideB}) + \beta(C + i * \mathrm{strideC})$$

Q & A

Do you make use of cuBLAS or other dedicated libraries in your code?

Yes, but:

- **cuBLAS:**

  `cublasZgemm:` $C = \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$

  `cublasGemmStridedBatchedEx:`

  $$C + i * \mathrm{strideC} = \alpha \mathrm{op}(A + i * \mathrm{strideA})\mathrm{op}(B + i * \mathrm{strideB}) + \beta(C + i * \mathrm{strideC})$$

- **cuTENSOR: possible, but typically support for in-place operations missing**

Q & A

Do you make use of cuBLAS or other dedicated libraries in your code?

Yes, but:

- **cuBLAS:**

  `cublasZgemm:` $C = \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$

  `cublasGemmStridedBatchedEx:`

  $$C + i * \mathrm{strideC} = \alpha \mathrm{op}(A + i * \mathrm{strideA})\mathrm{op}(B + i * \mathrm{strideB}) + \beta(C + i * \mathrm{strideC})$$

- **cuTENSOR: possible, but typically support for in-place operations missing**
- **Other libraries: often support for <double> times <double complex> missing**

Q & A

Do you make use of cuBLAS or other dedicated libraries in your code?

Yes, but:

- **cuBLAS:**

  `cublasZgemm:` $C = \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$

  `cublasGemmStridedBatchedEx:`

  $$C + i * \mathrm{strideC} = \alpha \mathrm{op}(A + i * \mathrm{strideA})\mathrm{op}(B + i * \mathrm{strideB}) + \beta(C + i * \mathrm{strideC})$$

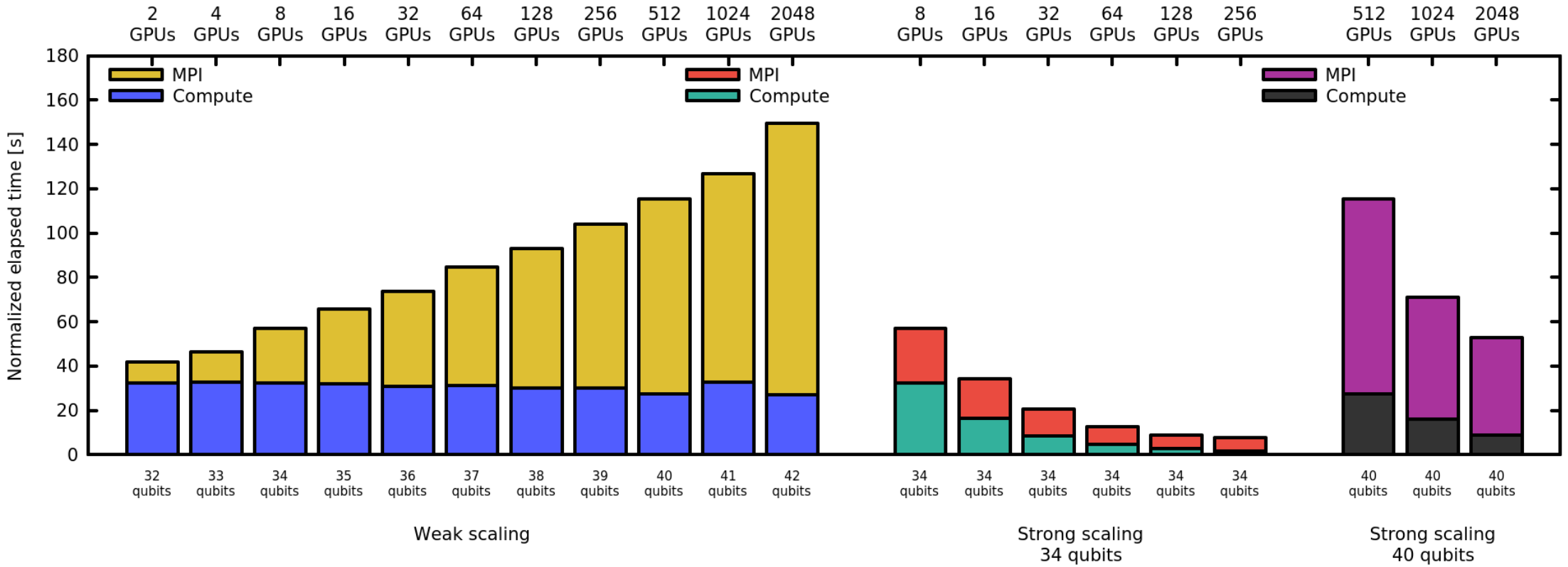
- **cuTENSOR: possible, but typically support for in-place operations missing**
- **Other libraries: often support for <double> times <double complex> missing**
- **Our custom CUDA kernels were faster**

Q & A

What is the bottleneck that limits the performance of JUQCS and are there ideas to improve it?

Q & A

What is the bottleneck that limits the performance of JUQCS and are there ideas to improve it?

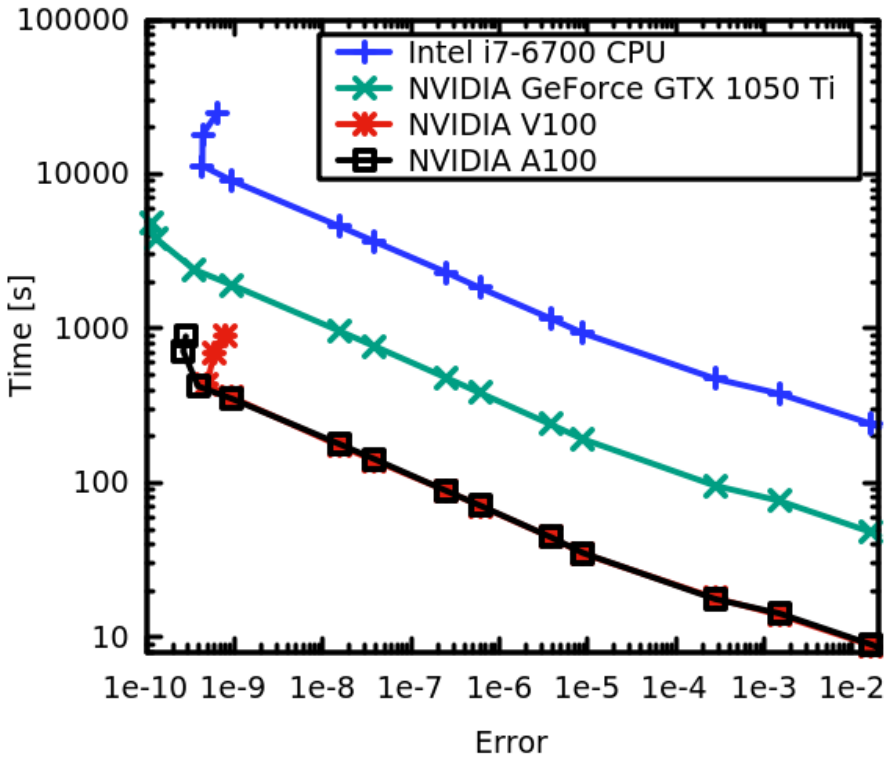


Q & A

Why did V100 and A100 perform equally well in the first JUQMES result?

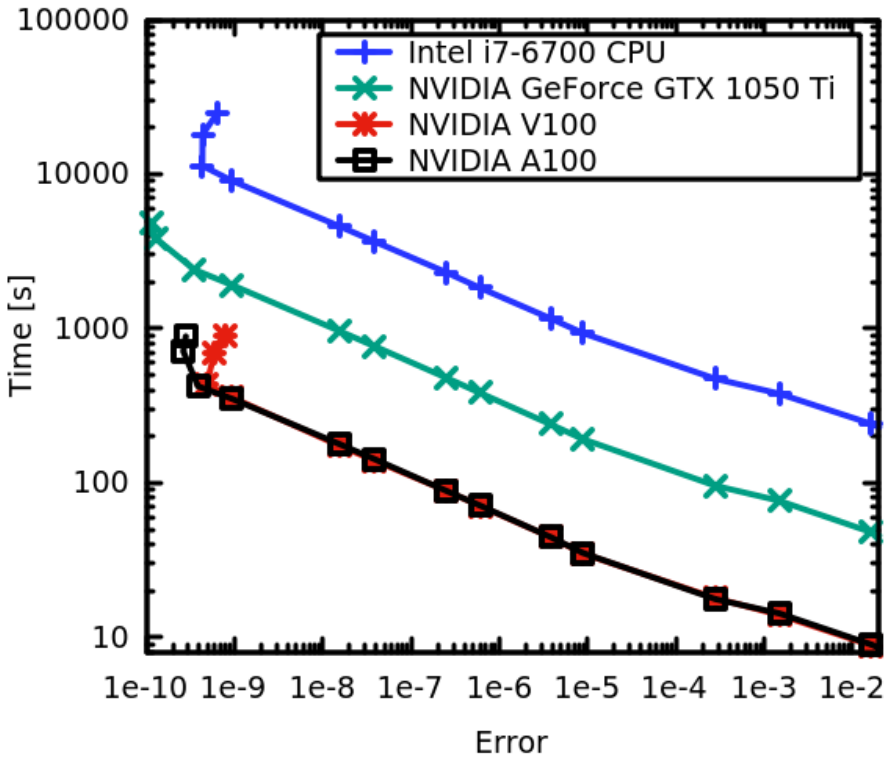
Q & A

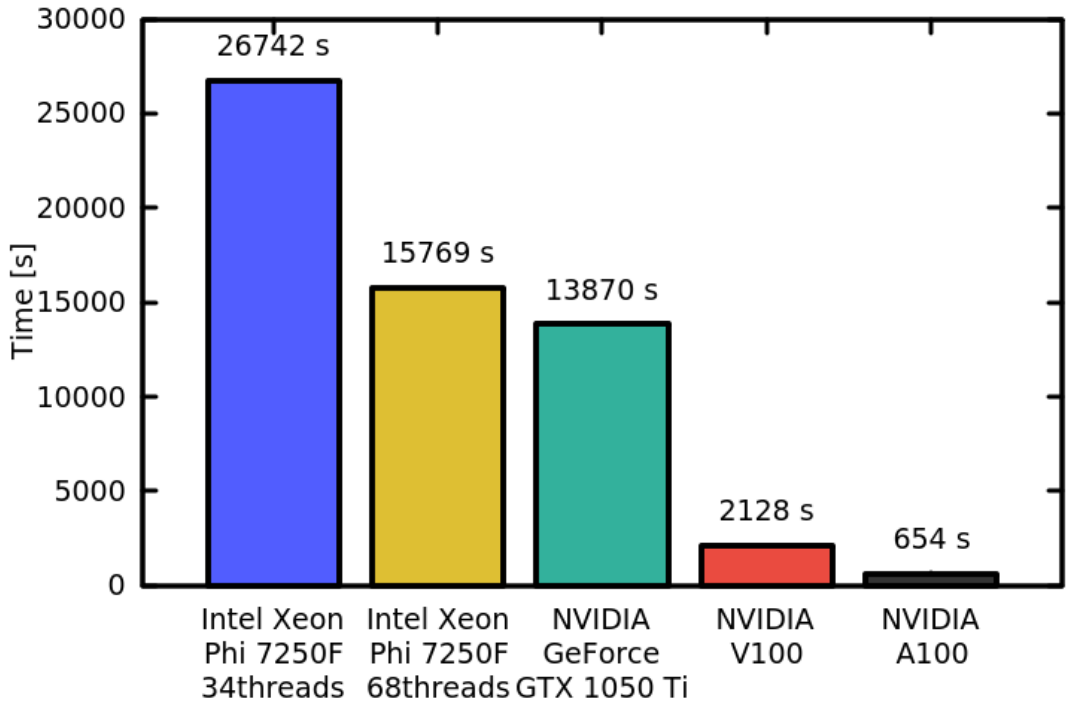
Why did V100 and A100 perform equally well in the first JUQMES result?



Q & A

Why did V100 and A100 perform equally well in the first JUQMES result?

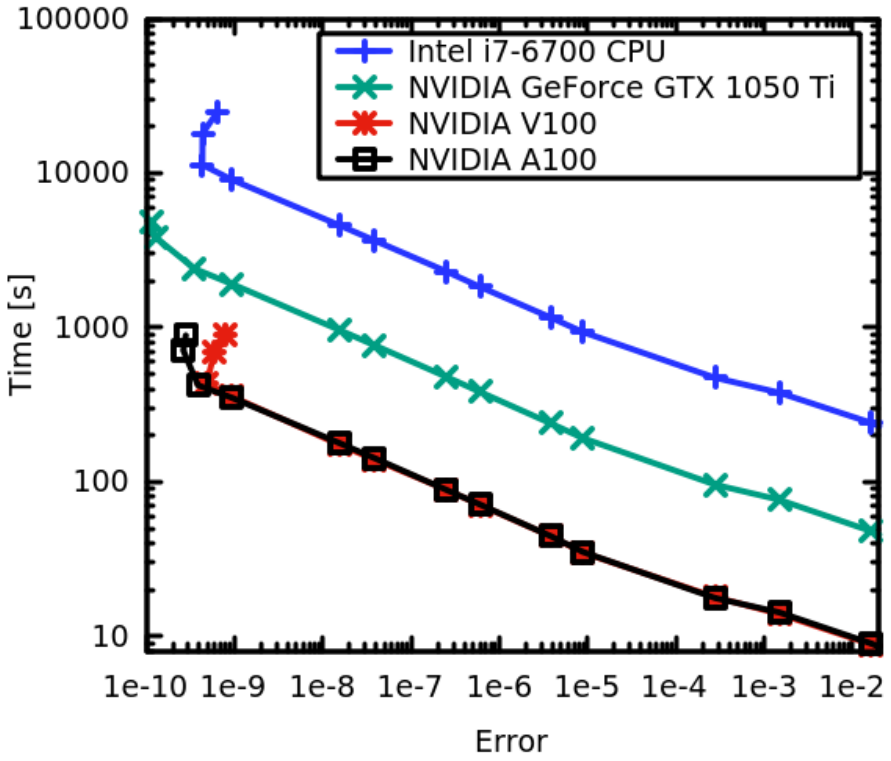


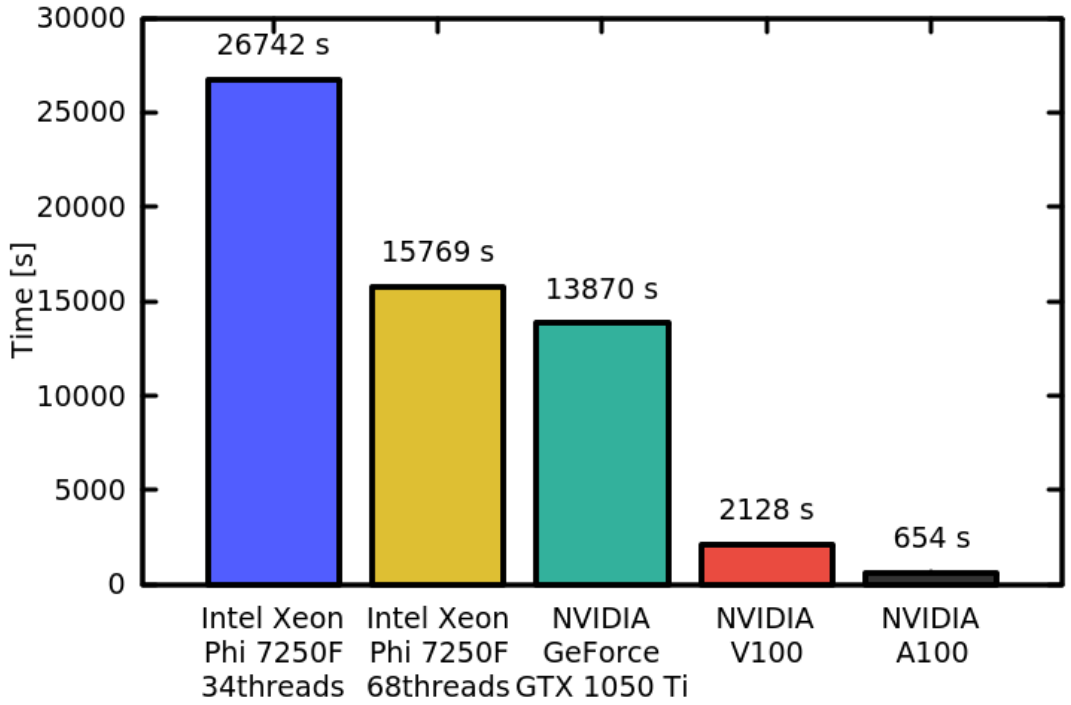


Q & A

$$\rho = (\rho_{k_0 m_0 k_1 m_1}) \quad \longrightarrow \quad \text{Memory: } K \times M \times K \times M$$

Why did V100 and A100 perform equally well in the first JUQMES result?

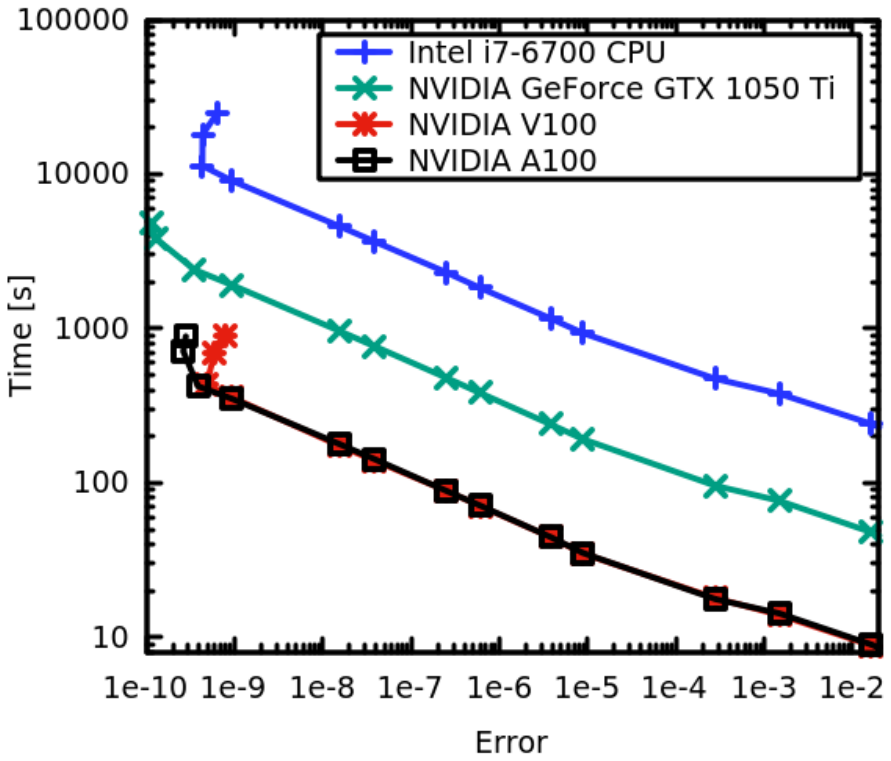


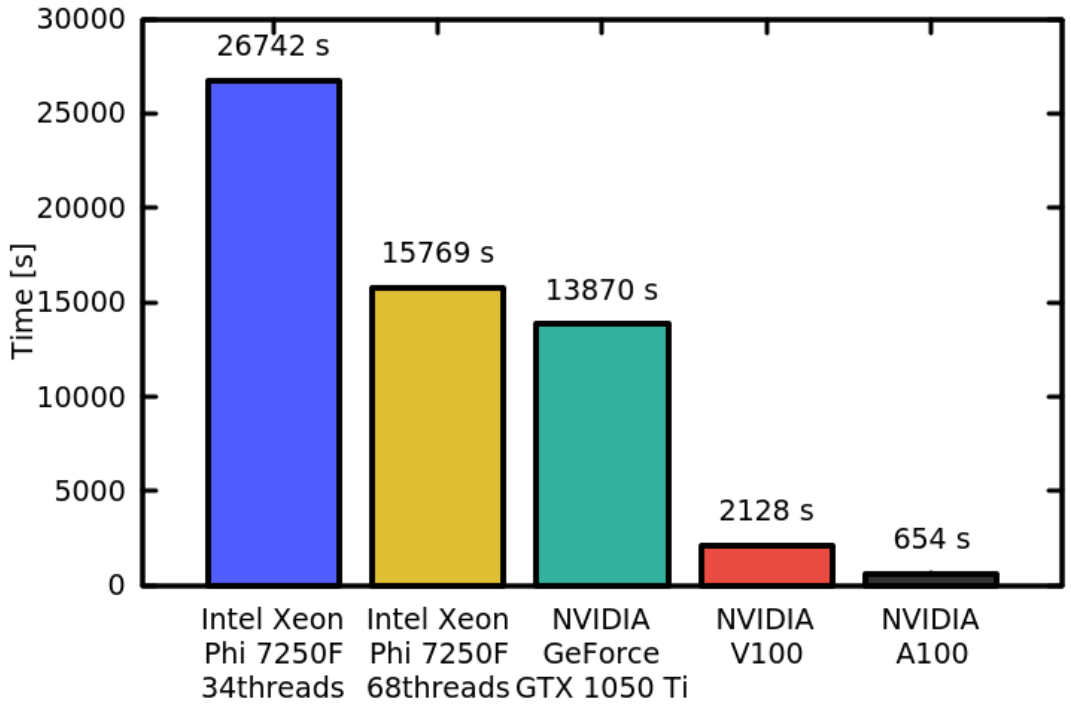


Q & A

$$\rho = (\rho_{k_0 m_0 k_1 m_1}) \quad \longrightarrow \quad \text{Memory: } K \times M \times K \times M$$

Why did V100 and A100 perform equally well in the first JUQMES result?



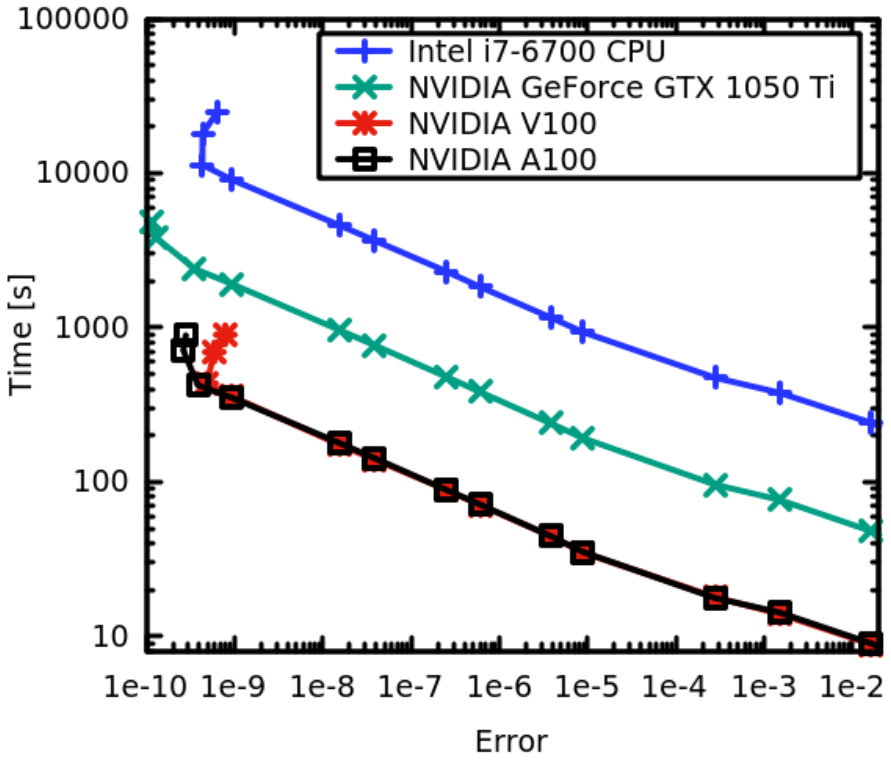


Small system with **M=4** and **K=100**

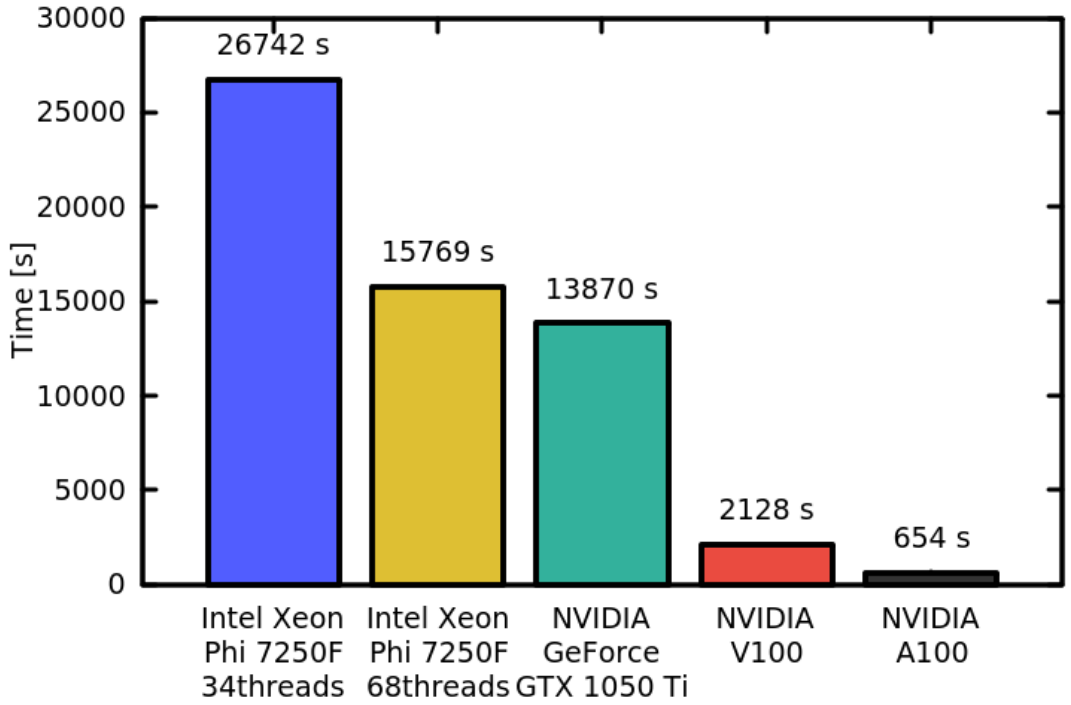
Q & A

$$\rho = (\rho_{k_0 m_0 k_1 m_1}) \quad \longrightarrow \quad \text{Memory: } K \times M \times K \times M$$

Why did V100 and A100 perform equally well in the first JUQMES result?



Small system with **M=4** and **K=100**



Realistic system with **M=12** and **K=400**