Implementing a Blazingly Fast Quantum State Simulator in Rust

Charlee Stefanski[1] Saveliy Yusufov[1]

[1]Advanced Computing, Wells Fargo

RustConf
September 2023
Albuquerque, NM, USA

Why does Wells Fargo care about quantum computing?

"If you are not in the game in some meaningful way, it is unlikely you will be a leader in the future."

Professor Dirk Englund, MIT in "Quantum Computing @ MIT: The Past, Present, and Future of the Second Revolution in Computing (arXiv:2002.05559)

Example focus areas and applications:

Optimization	→	Portfolio optimization, risk management (VaR)
Quantum reservoir computing	→	Models for financial time series
Stochastic models (hidden quantum Markov models)	→	limit order book price prediction
State preparation and random sampling	→	Sampling from classically intractable distributions, differential privacy

Example focus areas and applications:

Optimization	→	Portfolio optimization, risk management (VaR)
Quantum reservoir computing	→	Models for financial time series
Stochastic models (hidden quantum Markov models)	→	limit order book price prediction
State preparation and random sampling	→	Sampling from classically intractable distributions, differential privacy

Implementation and Learning of Quantum Hidden Markov Models

Vanio Markov,[1] Vladimir Rastunkov,[2] Amol Deshmukh,[2] Daniel Fry,[2] and Charlee Stefanski[1]

[1] *Wells Fargo*
[2] *IBM Quantum, IBM Research*
(Dated: July 7, 2023)

Quantum deep Q learning with distributed prioritized experience replay*

Samuel Yen-Chi Chen[1][0000−0003−0114−4826]

Wells Fargo, New York NY 10017, USA

Optimizing Quantum Noise-induced Reservoir Computing for Nonlinear and Chaotic Time Series Prediction

Daniel Fry,[1] Amol Deshmukh,[1] Samuel Yen-Chi Chen,[2] Vladimir Rastunkov,[1] and Vanio Markov[2]

[1] *IBM Quantum, IBM Research*
[2] *Wells Fargo*

Asynchronous training of quantum reinforcement learning

Samuel Yen-Chi Chen[1]

[1] *Wells Fargo*

(Dated: January 13, 2023)

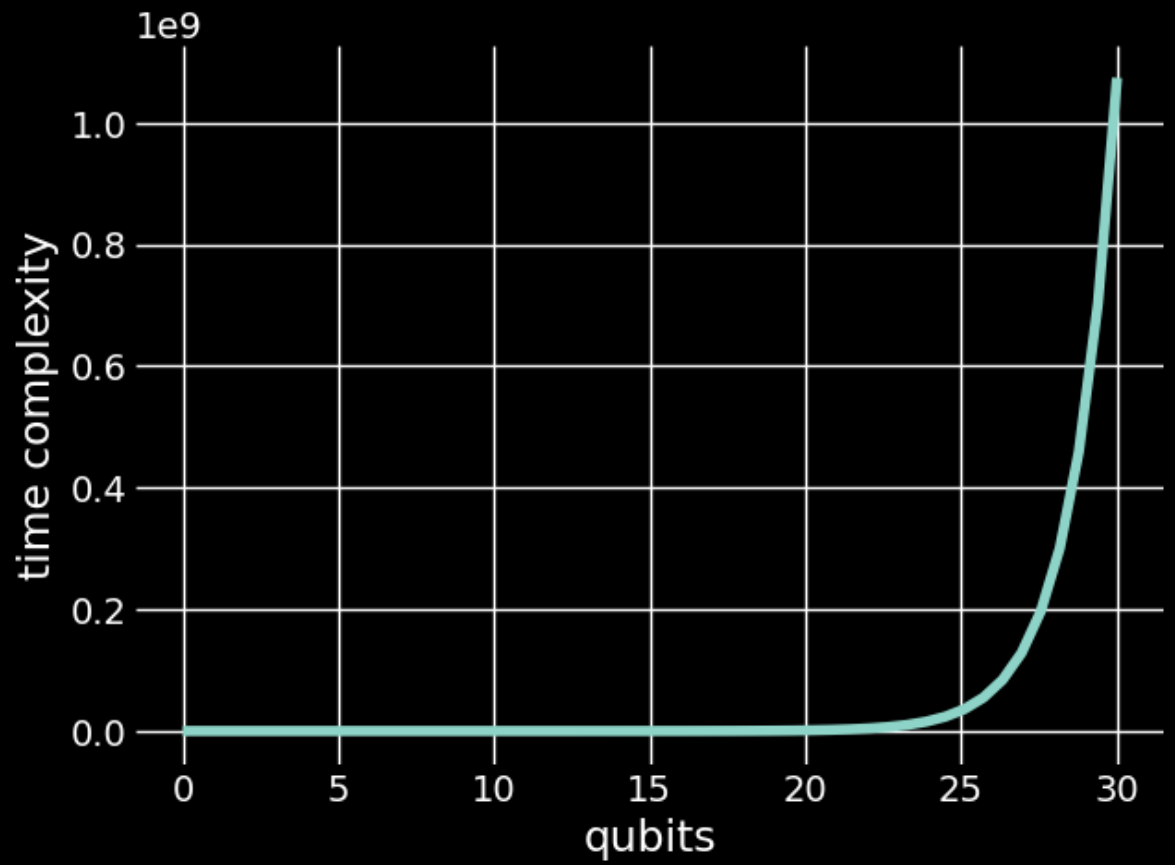
Why do we need a simulator?

Noise

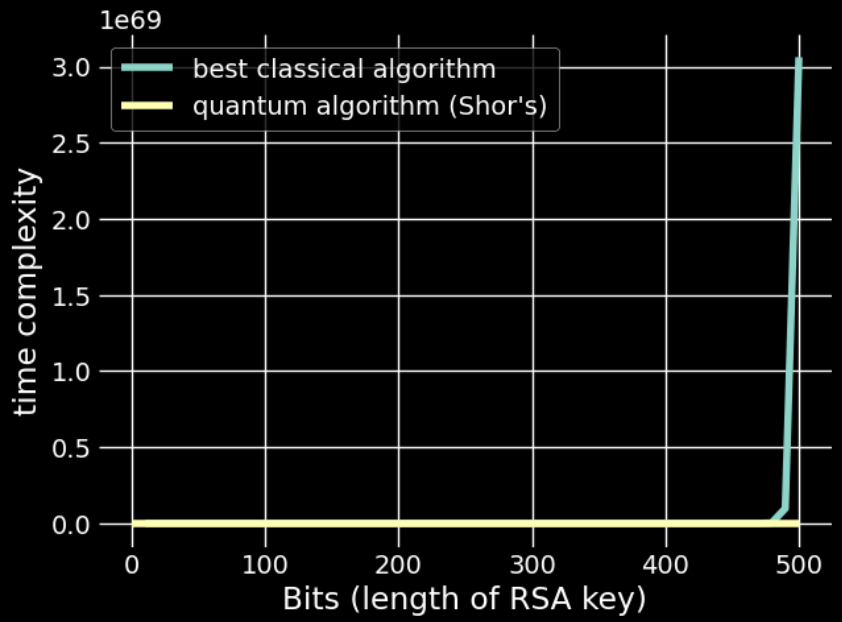
Development

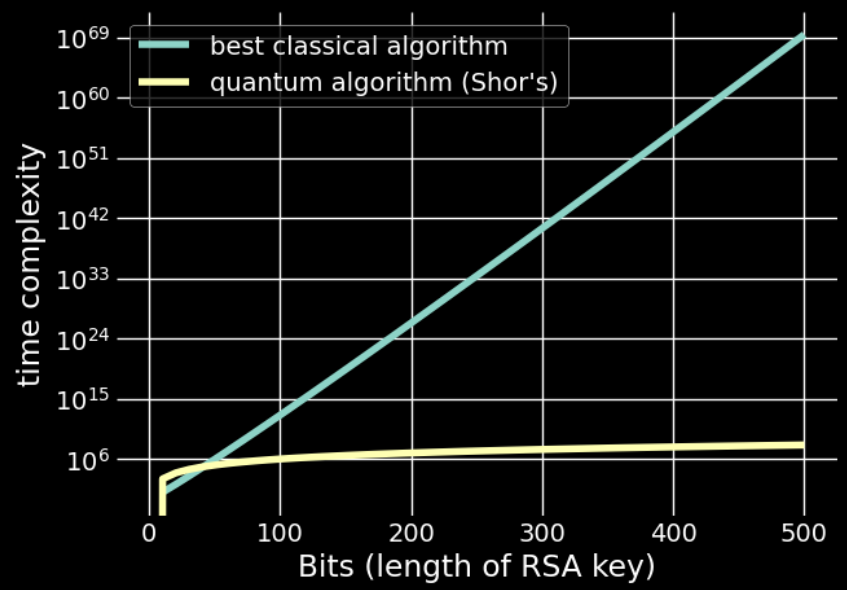
Optimization, Benchmarking, Education...

Computational complexity of simulating quantum computations



Quantum computers can break encryption?





Quantum computers can break encryption?

$\approx 1,000,000$ qubits to crack RSA[1]

Largest quantum computer has 433 qubits (and noise is still a big issue) .

[1] Are quantum computers about to break online privacy? https://www.nature.com/articles/d41586-023-00017-0

Quantum computers can break encryption?

$\approx 1,000,000$ qubits to crack RSA[1]

Largest quantum computer has 433 qubits (and noise is still a big issue) .

Can we just simulate it?

[1] Are quantum computers about to break online privacy? https://www.nature.com/articles/d41586-023-00017-0

Quantum computers can break encryption?

qubits $\rightarrow 10^6$

Quantum computers can break encryption?

qubits $\rightarrow 10^6$

complex numbers $\rightarrow 2^{10^{\mathbf{6}}}$

Quantum computers can break encryption?

qubits $\rightarrow 10^6$

complex numbers $\rightarrow 2^{10^6}$

classical bits required $\approx 2^{10^6} \cdot 128$ bits

Quantum computers can break encryption?

qubits $\rightarrow 10^6$

complex numbers $\rightarrow 2^{10^6}$

classical bits required $\approx 2^{10^6} \cdot 128$ bits

# of atoms in the observable universe $\approx 10^{80} \approx 2^{265.75}$

To simulate one million qubits, we would need more bits than there are atoms in the observable universe

$$2^{265.75}\ \text{bits} < 2^{10^6} \cdot 128\ \text{bits}$$

What can we simulate?

The Exascale-class HPE Cray EX Supercomputer at Oak Ridge National Laboratory

https://www.flickr.com/photos/olcf/52117623798/in/photostream/

What can we simulate?

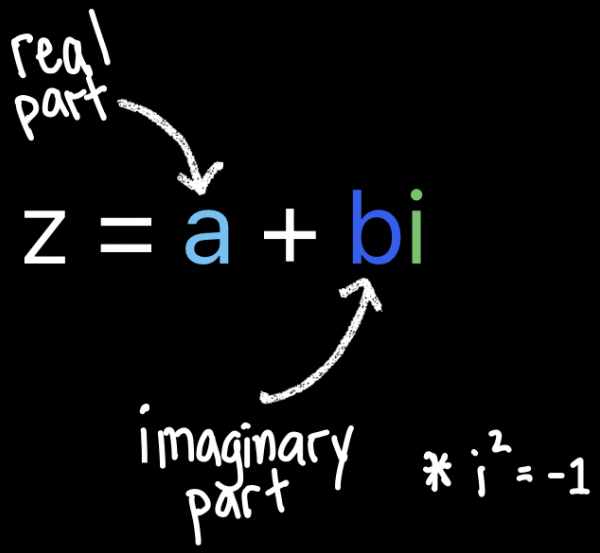
Let's divide the work evenly…

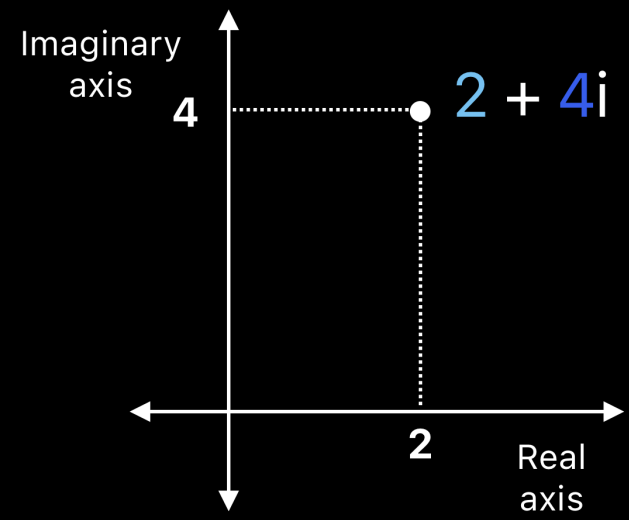
$$\frac{2^x \cdot 128 \text{ bits}}{9472} \leq 4 \text{ TB}$$

$$\approx 51 \text{ qubits}$$

Simulating quantum computations:
what you need to know

Reminder: Complex Numbers

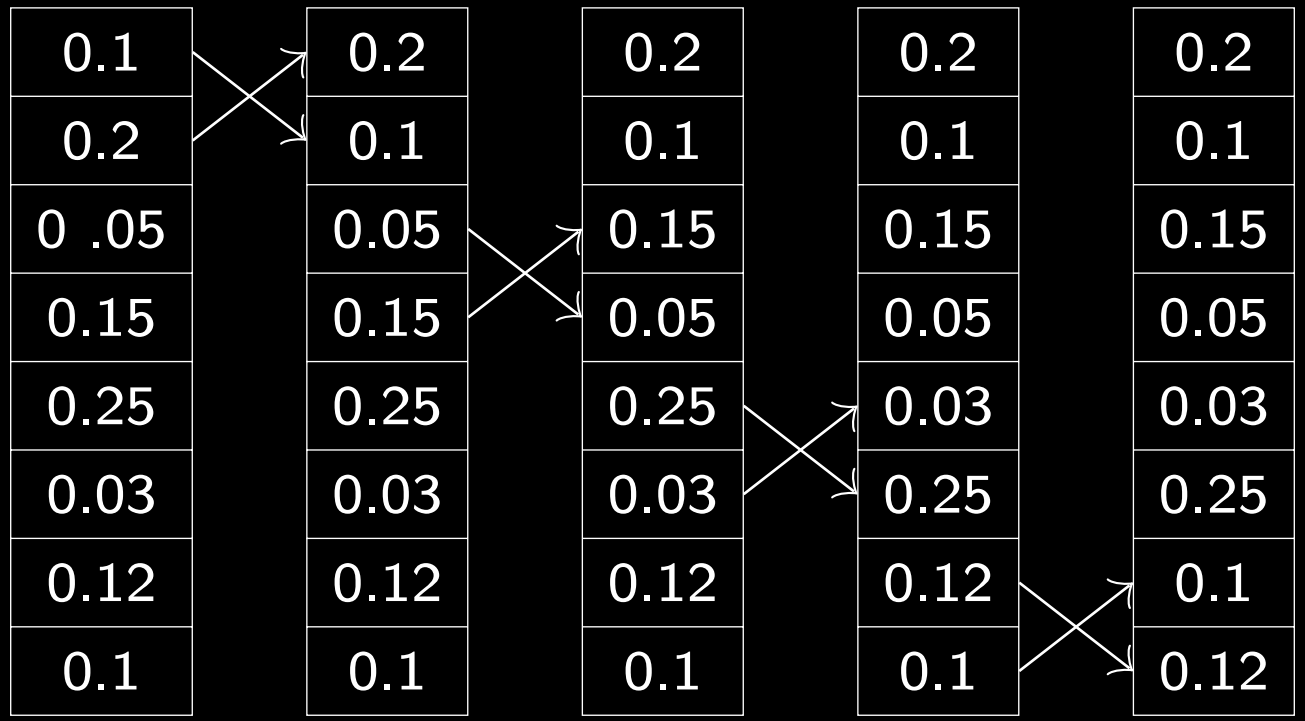


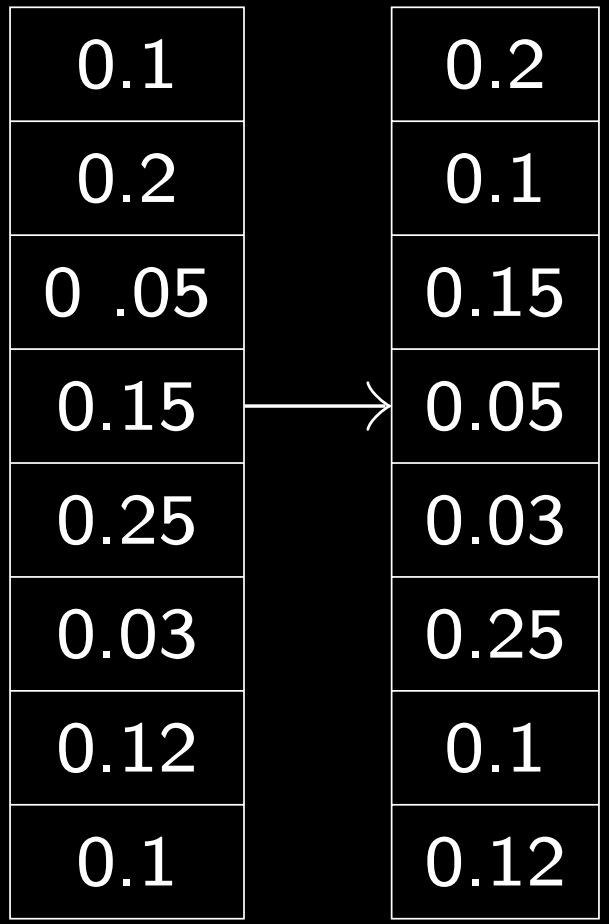


Representation in Rust:

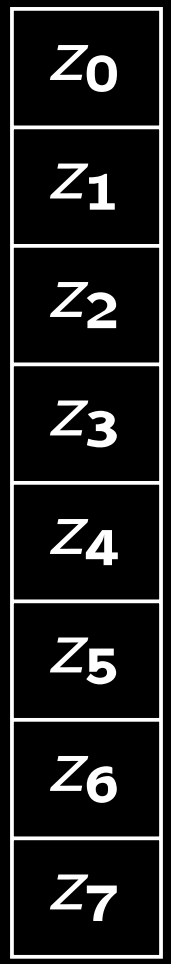
```
struct ComplexNum {
    real: f64,
    imaginary: f64,
}
```

Attribute	Value
0	0.1
1	0.2
2	0.05
3	0.15
4	0.25
5	0.03
6	0.12
7	0.1


0.1
0.2
0 .05
0.15
0.25
0.03
0.12
0.1
0.2
0.1
0.05
0.15
0.25
0.03
0.12
0.1
0.2
0.1
0.15
0.05
0.25
0.03
0.12
0.1
0.2
0.1
0.15
0.05
0.03
0.25
0.1
0.12


0.1
0.2
0 .05
0.15
0.25
0.03
0.12
0.1
0.2
0.1
0.15
0.05
0.03
0.25
0.1
0.12

State Vectors



Quantum Gates

$$\begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix}$$

Pairs

Avoid allocating a *huge* $2^n \cdot 2^n = 2^{2n}$ array…use pairs!

Basic quantum gates can be represented by a 2×2 matrix:

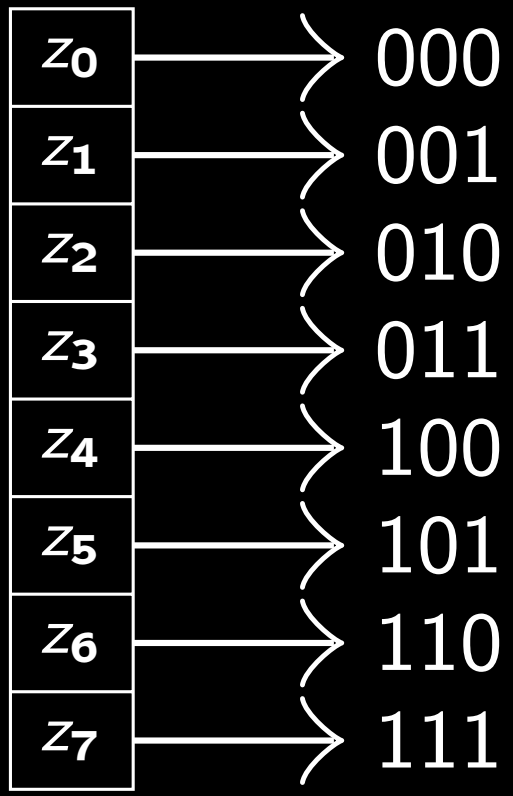
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Pairs

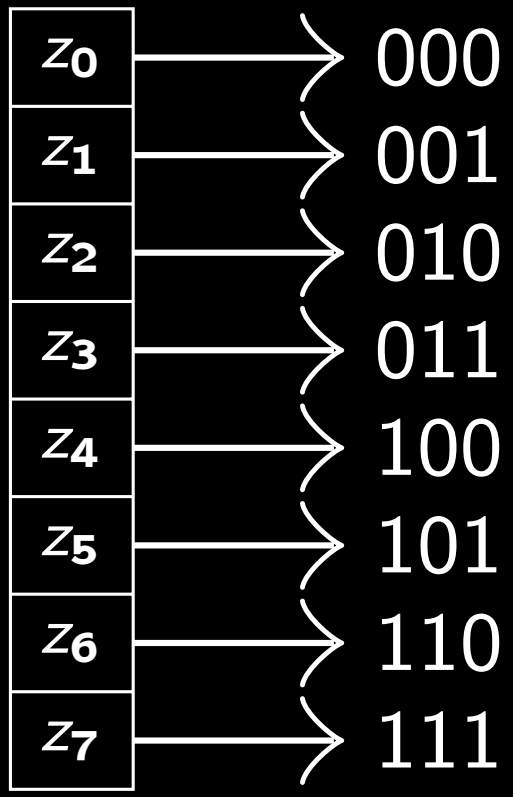
For n qubits, there are 2^n possible binary outcomes.

n = 2	n = 3	n = 4
00	000	0000 1000
01	001	0001 1001
10	010	0010 1010
11	011	0011 1011
	100	0100 1100
	101	0101 1101
	110	0110 1110
	111	0111 1111

Each amplitude in the state corresponds to an outcome.



Each amplitude in the state corresponds to an outcome.



Note: The integer form of each binary number is the index of the corresponding amplitude (i.e., 000 is 0, 001 is 1).

Pairs

Pairs of amplitudes correspond to the outcomes that differ only in position t.
Note: Qubits are indexed from the right (this is a convention in quantum computing).

$n = 3$ qubits, $t = 0$

$$\boxed{000}\ \boxed{001}\ \boxed{010}\ \boxed{011}\ \boxed{100}\ \boxed{101}\ \boxed{110}\ \boxed{111}$$

$n = 4$ qubits, $t = 1$

0000 and 0010

0001 and 0011

0100 and 0110

0101 and 0111

1000 and 1010

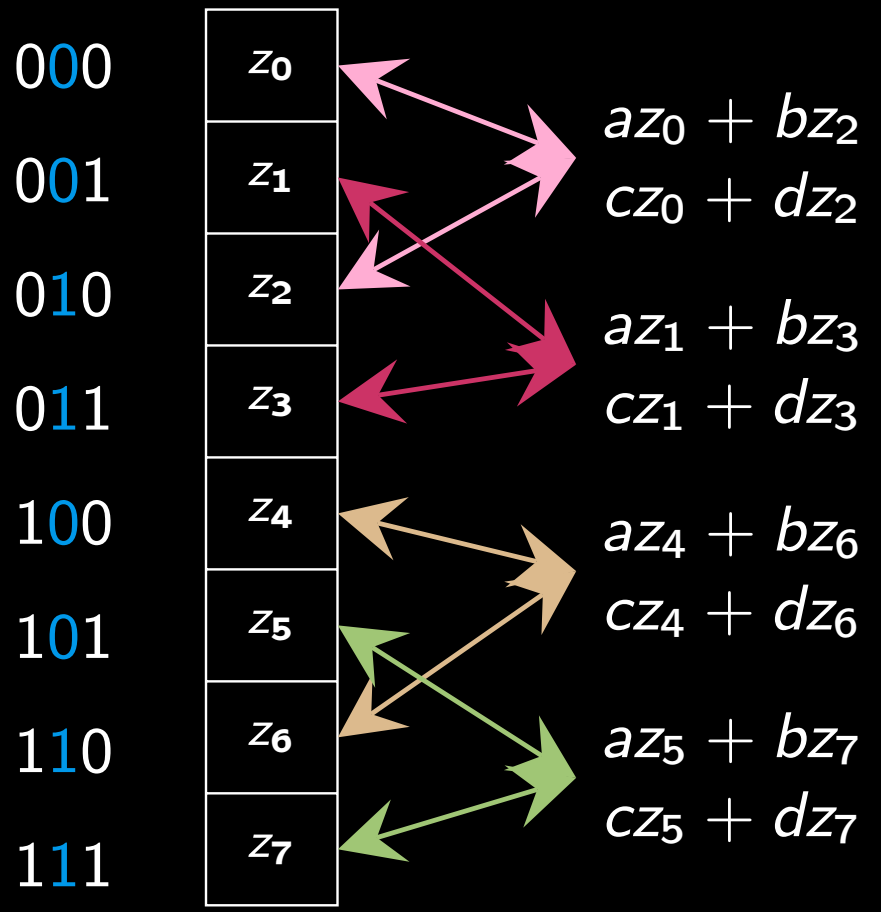
1001 and 1011

1100 and 1110

1101 and 1111

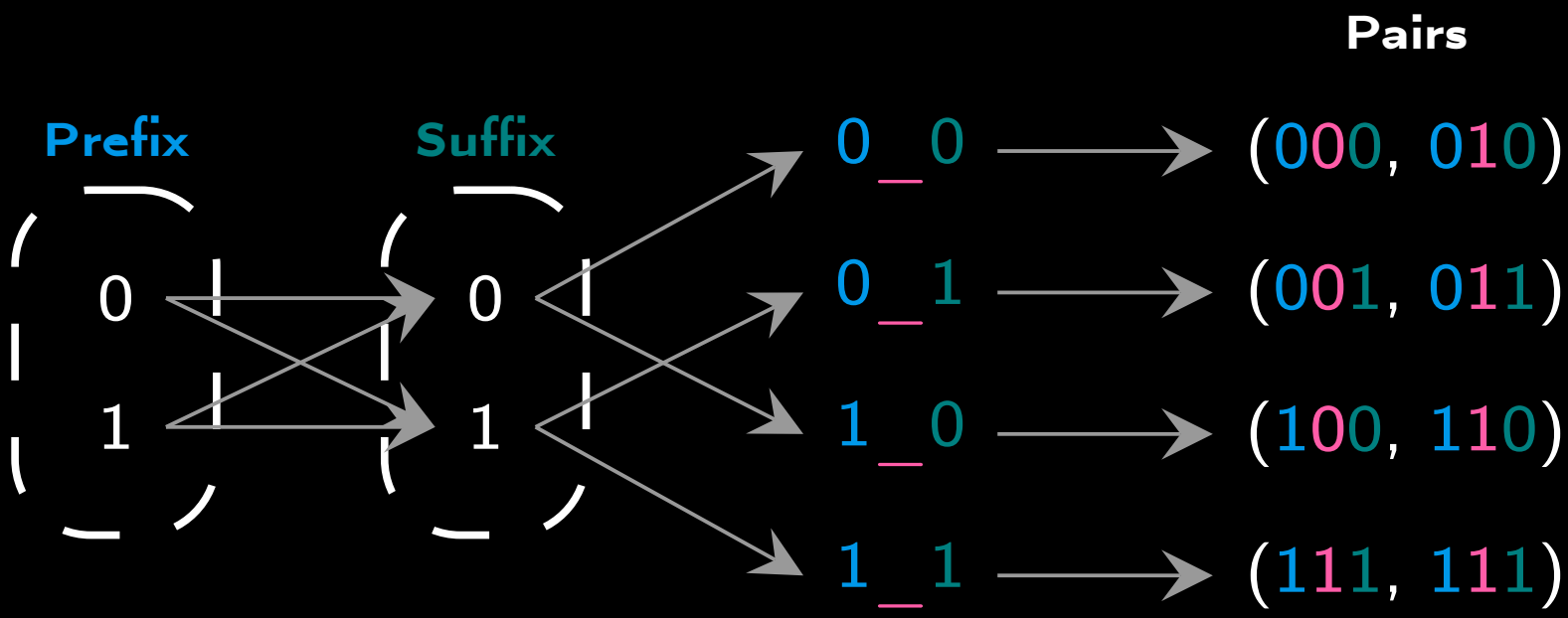
Applying a gate with pairs

Applying a general single-qubit gate $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ to a three-qubit state ($n = 3$) with target qubit 1 ($t = 1$):



Strategy 3 — Insertion

Generate the prefix & suffix together, then *insert* the target



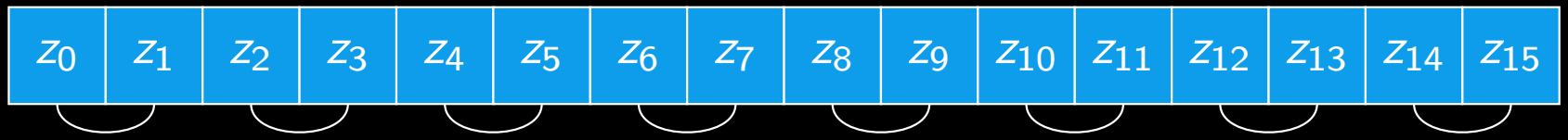
Strategy 3 — Insertion

```rust
pub fn apply_strategy3(state: &mut State, target: usize, diag_matrix: &[Amplitude; 2]) {
    let distance = 1 << target;
    let num_pairs = state.len() >> 1;

    for i in 0..num_pairs {
        let s0 = i + ((i >> target) << target);
        let s1 = s0 + distance;
        recombine(state, s0, s1, diag_matrix);
    }
}
```

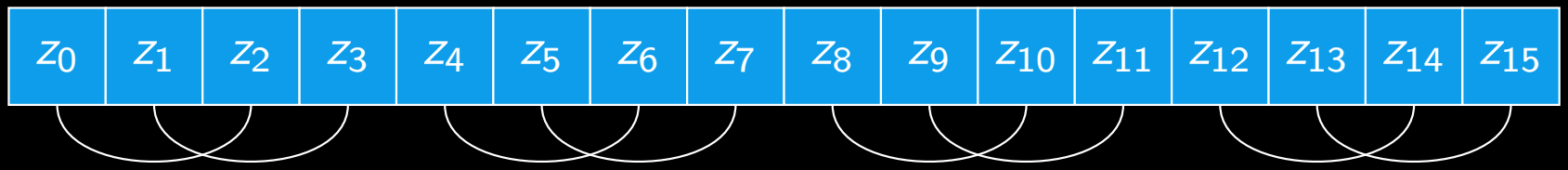
n = 4

target = 0



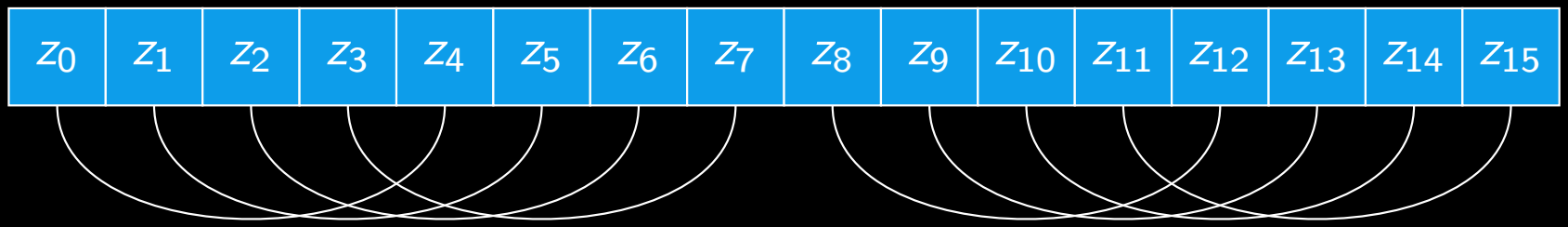
n = 4

target = 1



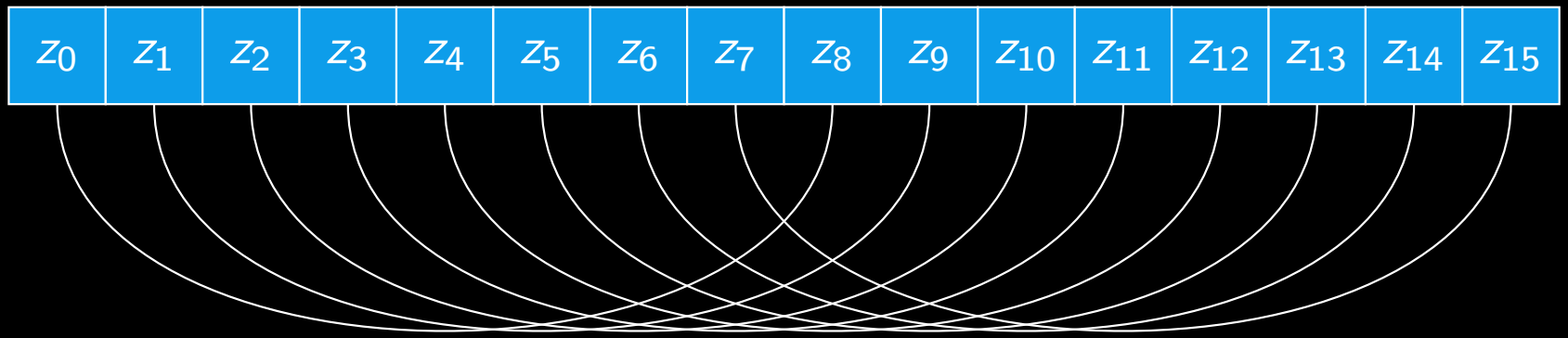
n = 4

target = 2

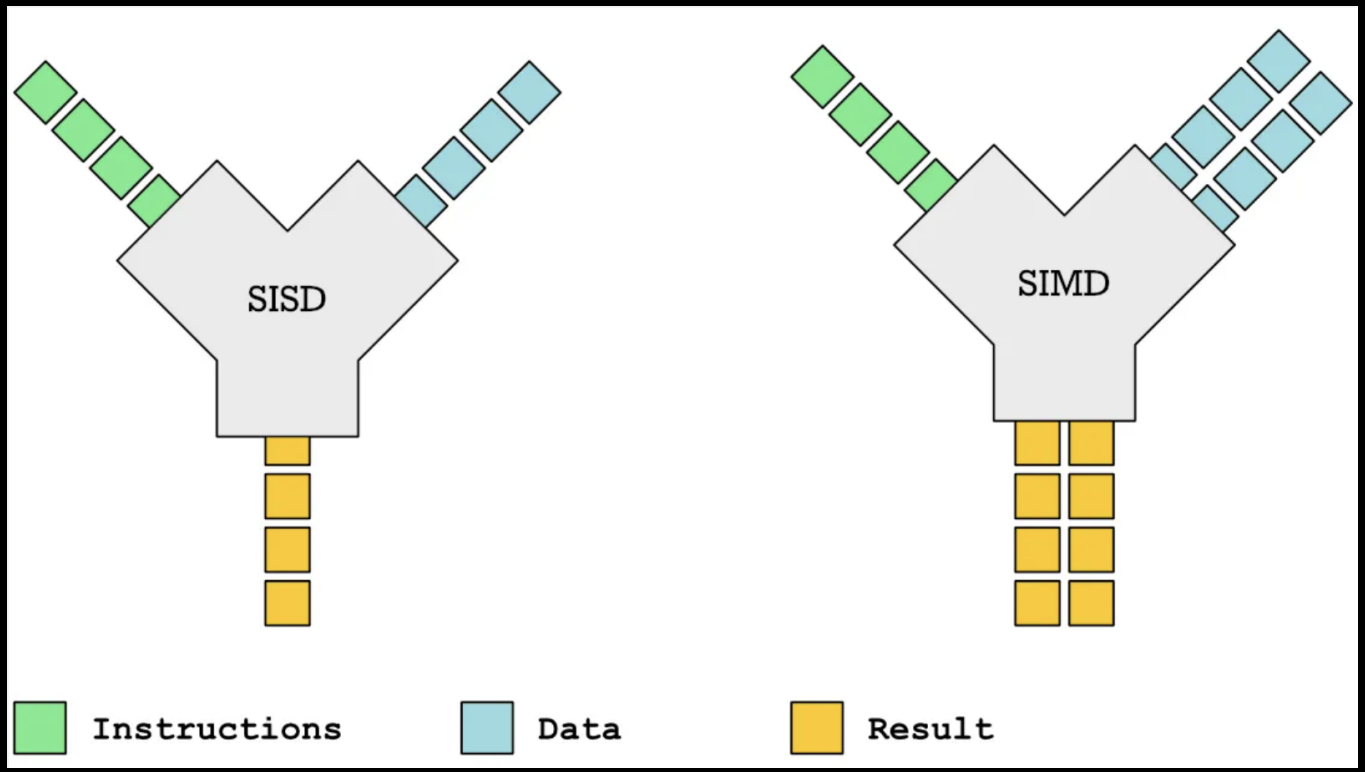


n = 4

target = 3



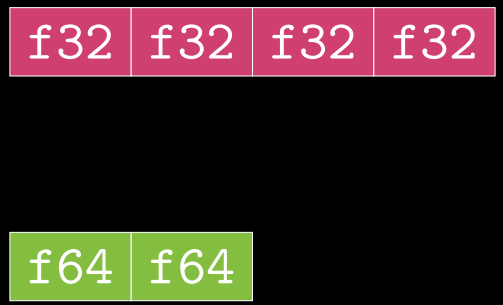
SIMD



https://miro.medium.com/v2/format:webp/1*VineM-3sao5qn76JYeWhMQ.png

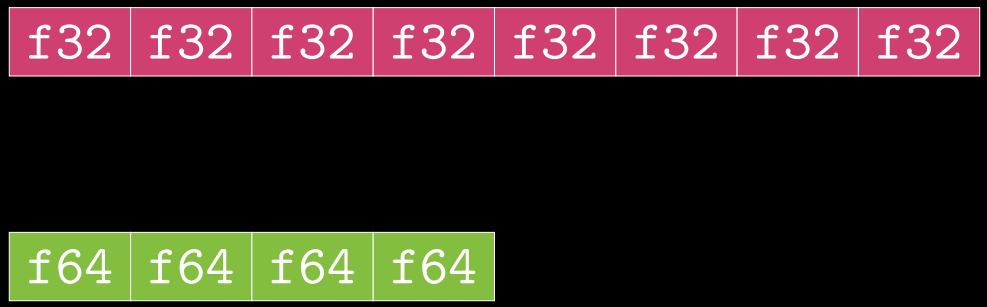
SIMD

128 bit



SIMD

256 bit



Insertion Strategy ASM

```
vmovupd xmm4, xmmword ptr [r9 + rax]
vmulpd  xmm5, xmm4, xmm3
vshufpd xmm5, xmm5, xmm5, 1
vmulpd  xmm4, xmm4, xmm2
vaddpd  xmm6, xmm4, xmm5
vsubpd  xmm4, xmm4, xmm5
vblendpd        xmm4, xmm4, xmm6, 1
vmovupd xmmword ptr [r9 + rax], xmm4
shl     r10, 4
vmovupd xmm4, xmmword ptr [r9 + r10]
vmulpd  xmm5, xmm4, xmm1
vshufpd xmm5, xmm5, xmm5, 1
vmulpd  xmm4, xmm4, xmm0
vaddpd  xmm6, xmm4, xmm5
vsubpd  xmm4, xmm4, xmm5
vblendpd        xmm4, xmm4, xmm6, 1
vmovupd xmmword ptr [r9 + r10], xmm4
```

Insertion Strategy ASM

`c2-standard-16` machine instruction set extensions:

Intel® SSE4.2, Intel® AVX, Intel® AVX2, Intel® AVX-512

---

use `lscpu` on *nix systems to find available cpu instructions

Insertion Strategy ASM

`c2-standard-16` machine instruction set extensions:

Intel® SSE4.2, Intel® AVX, Intel® AVX2, Intel® AVX-512

No :(

---

use `lscpu` on *nix systems to find available cpu instructions

3 levels of understanding how SIMD works

1. Compilers are smart! They will auto-vectorize all the code!

---

matklad, *Can You Trust a Compiler to Optimize Your Code?*, `https://matklad.github.io/2023/04/09/can-you-trust-a-compiler-to-optimize-your-code.html`

3 levels of understanding how SIMD works

1. Compilers are smart! They will auto-vectorize all the code!
2. Compilers are dumb . . . It's always better to manually write explicit SIMD instructions.

---

matklad, *Can You Trust a Compiler to Optimize Your Code?*, `https://matklad.github.io/2023/04/09/can-you-trust-a-compiler-to-optimize-your-code.html`

3 levels of understanding how SIMD works

1. Compilers are smart! They will auto-vectorize all the code!
2. Compilers are dumb . . . It's always better to manually write explicit SIMD instructions.
3. Writing SIMD by hand is really hard . . . Compilers can vectorize code if it's written in a form that's amenable-to-vectorization.

---

matklad, *Can You Trust a Compiler to Optimize Your Code?*, `https://matklad.github.io/2023/04/09/can-you-trust-a-compiler-to-optimize-your-code.html`

More chunks, less branching

- express the algorithm in terms of processing chunks of elements

---

matklad, *Can You Trust a Compiler to Optimize Your Code?*, `https://matklad.github.io/2023/04/09/can-you-trust-a-compiler-to-optimize-your-code.html`

More chunks, less branching

- express the algorithm in terms of processing chunks of elements
- within each chunk, ensure no branching and all elements are processed similarly

---

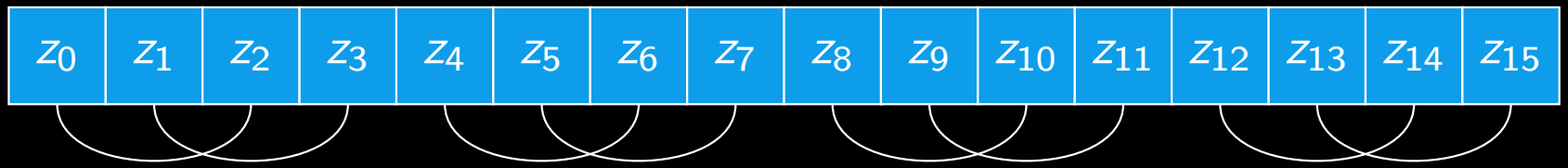
matklad, *Can You Trust a Compiler to Optimize Your Code?*, `https://matklad.github.io/2023/04/09/can-you-trust-a-compiler-to-optimize-your-code.html`

Where are the chunks?

Recall

`n = 4`

`target = 1`

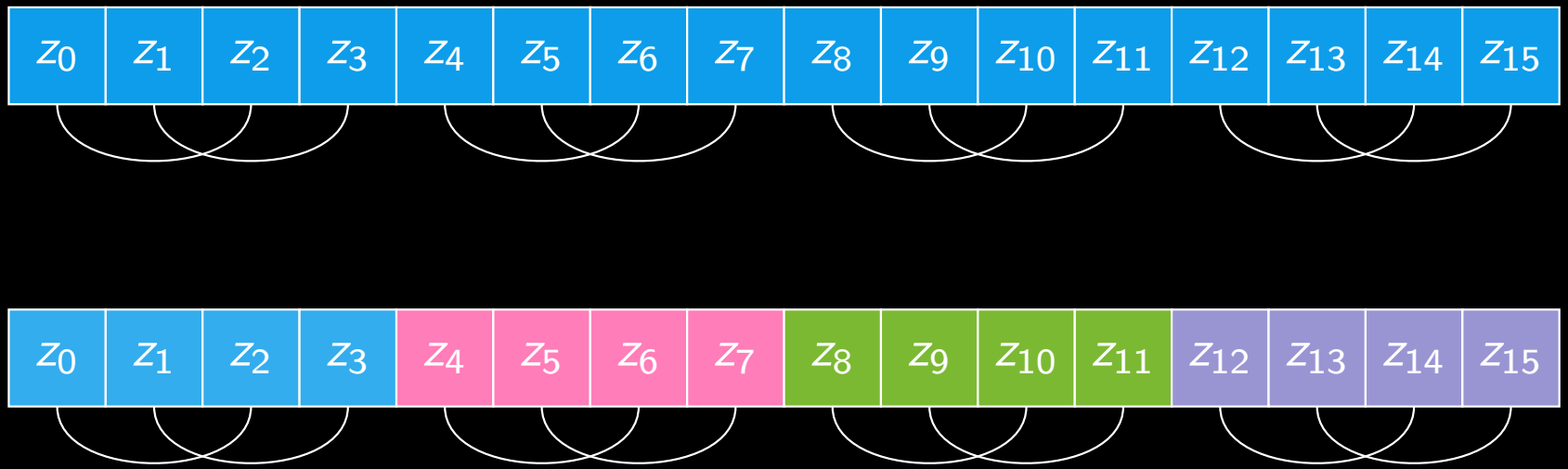


Where are the chunks?

Recall

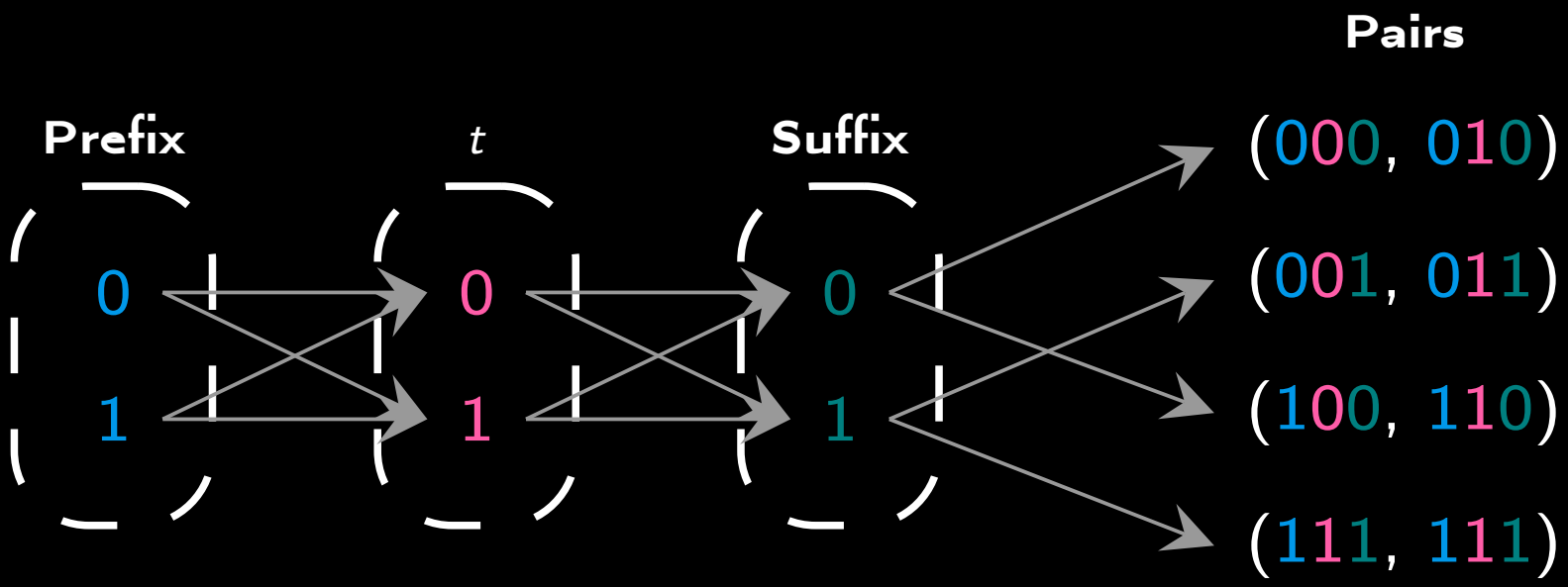
`n = 4`

`target = 1`



Strategy 2 – Concatenation

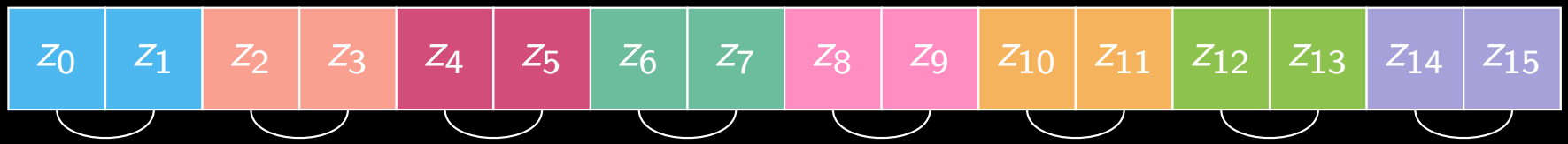
Generate prefix, append target, then generate suffix



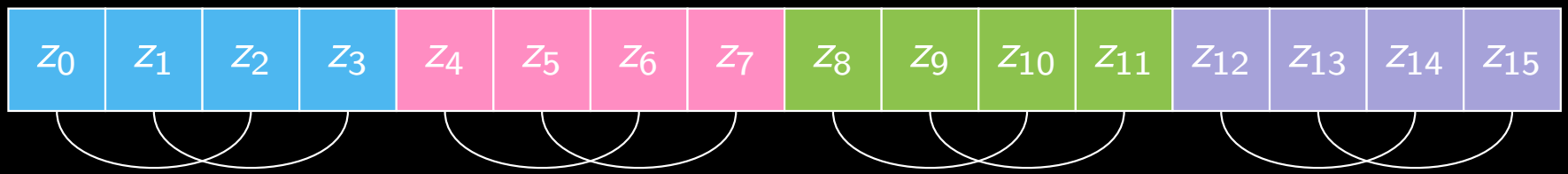
Strategy 2 — Concatenation

```rust
pub fn apply_strategy2(state: &mut State, target: usize, diag_matrix: &[Amplitude; 2]) {
    let chunks = state.len() >> (target + 1);
    (0..chunks).for_each(|chunk| {
        let dist = 1 << target;
        let base = (2 * chunk) << target;
        for i in 0..dist {
            let s0 = base + i;
            let s1 = s0 + dist;
            recombine(state, s0, s1, diag_matrix);
        }
    });
}
```

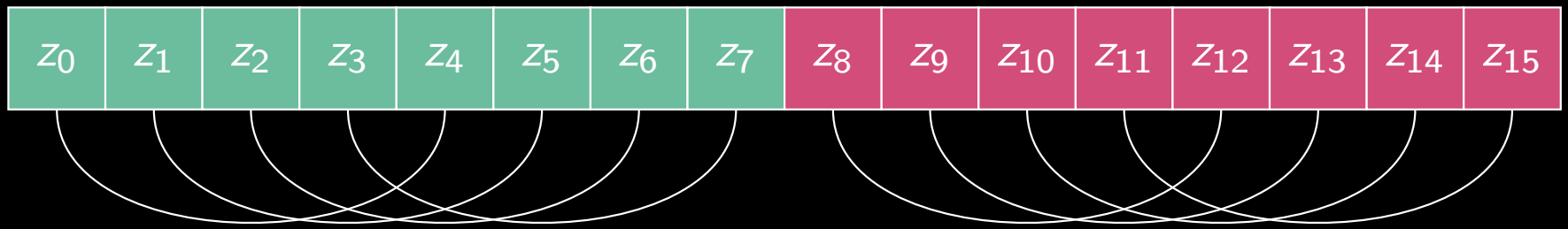
```
n = 4
target = 0
num_chunks = state.len() / 2^(target+1) = 8
```



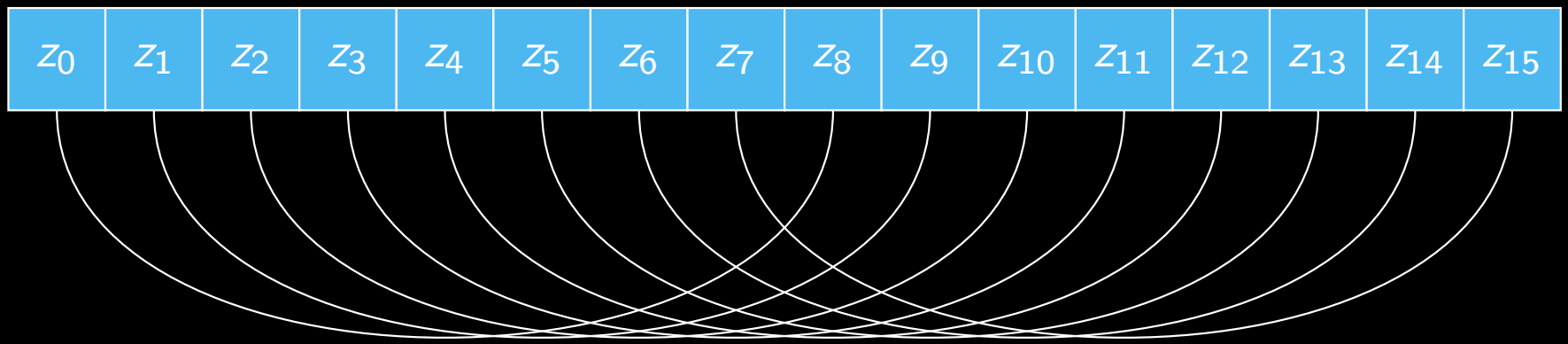
```
n = 4
target = 1
num_chunks = state.len() / 2^(target+1) = 4
```



```
n = 4
target = 2
num_chunks = state.len() / 2^(target+1) = 2
```



```
n = 4
target = 3
num_chunks = state.len() / 2^(target+1) = 1
```



Concatenation strategy generated ASM

```
vmovupd ymm13, ymmword ptr [r10 + rdi]
vmovupd ymm14, ymmword ptr [r10 + rdi + 32]
vperm2f128      ymm15, ymm13, ymm14, 49
vperm2f128      ymm13, ymm13, ymm14, 32
vunpcklpd       ymm14, ymm13, ymm15
vunpckhpd       ymm13, ymm13, ymm15
add     rdx, r9
shl     rdx, 4
vmovupd ymm15, ymmword ptr [r10 + rdx]
vmovupd ymm16, ymmword ptr [r10 + rdx + 32]
vshuff64x2      ymm17, ymm15, ymm16, 3
vshuff64x2      ymm15, ymm15, ymm16, 0
vunpcklpd       ymm16, ymm15, ymm17
vunpckhpd       ymm15, ymm15, ymm17
vmulpd  ymm17, ymm13, ymm2
vmulpd  ymm18, ymm14, ymm3
vsubpd  ymm17, ymm17, ymm18
vmulpd  ymm13, ymm13, ymm3
vmulpd  ymm14, ymm14, ymm2
vaddpd  ymm13, ymm14, ymm13
vinsertf64x4    zmm14, zmm17, ymm17, 1
vinsertf64x4    zmm13, zmm13, ymm13, 1
vpermt2pd       zmm13, zmm9, zmm14
vmovupd zmmword ptr [r10 + rdi], zmm13
vmulpd  ymm13, ymm15, ymm4
vmulpd  ymm14, ymm16, ymm5
vsubpd  ymm13, ymm13, ymm14
vmulpd  ymm14, ymm15, ymm5
vmulpd  ymm15, ymm16, ymm4
vaddpd  ymm14, ymm15, ymm14
vinsertf64x4    zmm13, zmm13, ymm13, 1
vinsertf64x4    zmm14, zmm14, ymm14, 1
vpermt2pd       zmm14, zmm9, zmm13
vmovupd zmmword ptr [r10 + rdx], zmm14
vpaddq  ymm12, ymm12, ymm10
```

Lower Latency

- SIMD isn’t always simple

Lower Latency

- SIMD isn’t always simple
- Memory layout may require a lot of permutations (expensive)

Let’s go back to the drawing board

Memory Layout

```rust
struct ComplexNum {
    real: f64,
    imaginary: f64,
}
```

```rust
pub struct State {
    amps: Vec<ComplexNum>,
}
```

Memory Layout

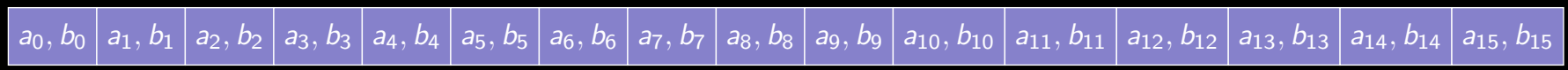
```rust
struct ComplexNum {
    real: f64,
    imaginary: f64,
}
```

```rust
pub struct State {
    amps: Vec<ComplexNum>,
}
```

Memory Layout

```rust
struct ComplexNum {
    real: f64,
    imaginary: f64,
}
```

```rust
pub struct State {
    amps: Vec<ComplexNum>,
}
```

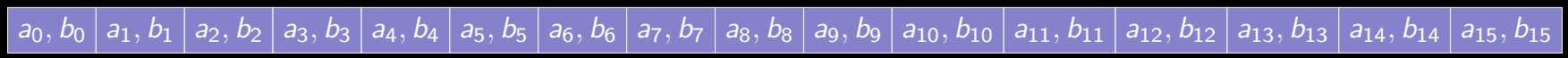


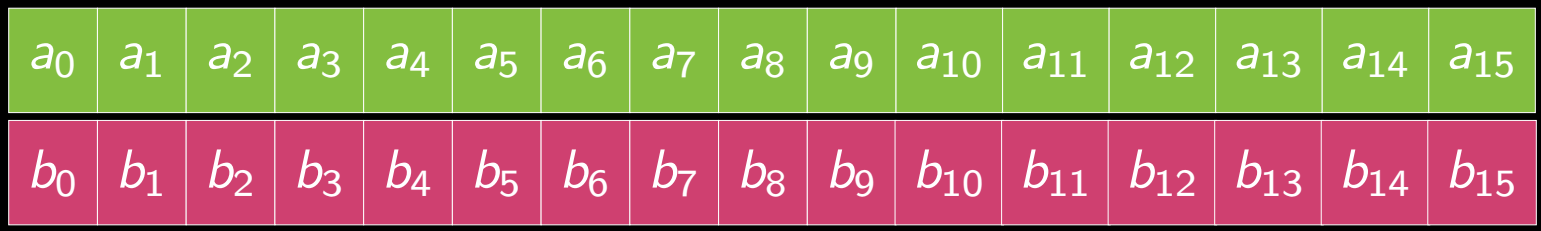
Memory Layout

a_0, b_0	a_1, b_1	a_2, b_2	a_3, b_3	a_4, b_4	a_5, b_5	a_6, b_6	a_7, b_7	a_8, b_8	a_9, b_9	a_{10}, b_{10}	a_{11}, b_{11}	a_{12}, b_{12}	a_{13}, b_{13}	a_{14}, b_{14}	a_{15}, b_{15}

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}

Memory Layout






```
pub struct State {
    reals: Vec<f64>,
    imags: Vec<f64>,
}
```

Concatenation Strategy

```rust
pub fn apply_strategy2(state: &mut State, target: usize, diag_matrix: &[Amplitude; 2]) {
    let chunks = state.len() >> (target + 1);
    (0..chunks).for_each(|chunk| {
        let dist = 1 << target;
        let base = (2 * chunk) << target;
        for i in 0..dist {
            let s0 = base + i;
            let s1 = s0 + dist;
            recombine(state, s0, s1, diag_matrix);
        }
    });
}
```

ASM for Concatenation Strategy

```
vpaddq  xmm12, xmm11, xmm10
vmovq   rdx, xmm12
vmovupd ymm12, ymmword ptr [rbx + 8*rdx]
vmovupd ymm13, ymmword ptr [r14 + 8*rdx]
lea     rdi, [rdx + r11]
vmovupd ymm14, ymmword ptr [rbx + 8*rdi]
vmovupd ymm15, ymmword ptr [r14 + 8*rdi]
vmulpd  ymm16, ymm12, ymm4
vmulpd  ymm17, ymm13, ymm5
vsubpd  ymm16, ymm16, ymm17
vmovupd ymmword ptr [rbx + 8*rdx], ymm16
vmulpd  ymm12, ymm12, ymm5
vmulpd  ymm13, ymm13, ymm4
vaddpd  ymm12, ymm13, ymm12
vmovupd ymmword ptr [r14 + 8*rdx], ymm12
vmulpd  ymm12, ymm14, ymm6
vmulpd  ymm13, ymm15, ymm7
vsubpd  ymm12, ymm12, ymm13
vmovupd ymmword ptr [rbx + 8*rdi], ymm12
vmulpd  ymm12, ymm14, ymm7
vmulpd  ymm13, ymm15, ymm6
vaddpd  ymm12, ymm13, ymm12
vmovupd ymmword ptr [r14 + 8*rdi], ymm12
vpaddq  ymm11, ymm11, ymm9
```

Cache rules everything around me

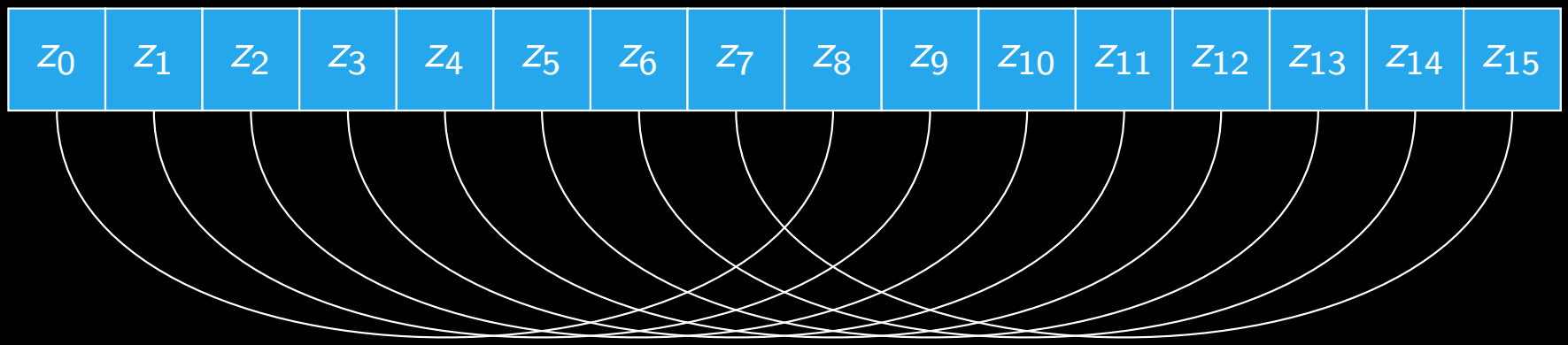
Recall

n = 4

target = 3

dist = $2^{target} = 8$

num_chunks = state.len() / $2^{target+1} = 1$

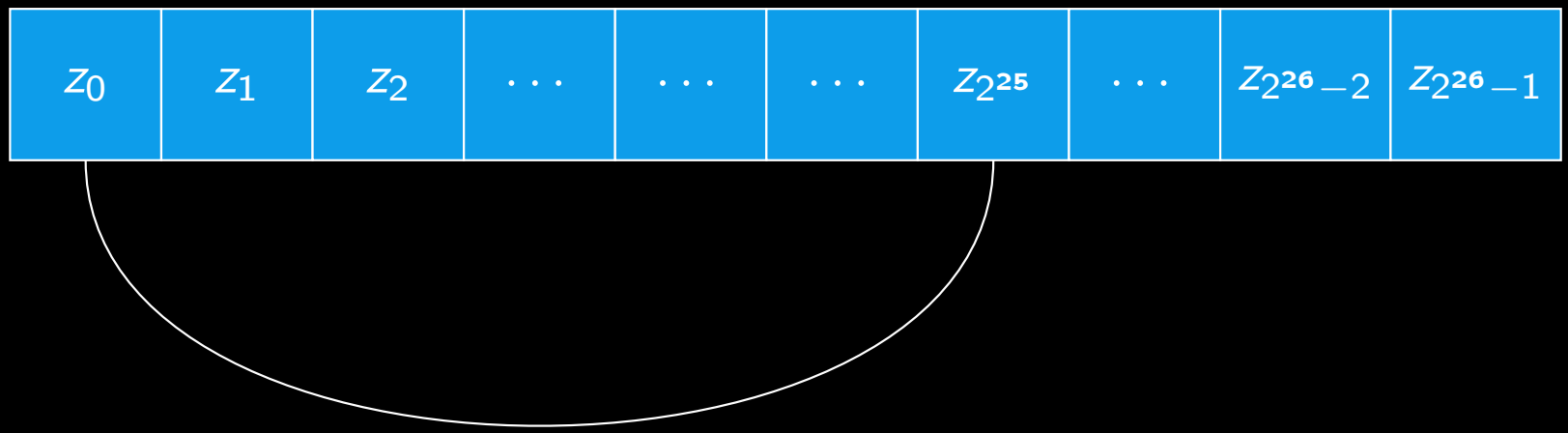


Consider: `n = 26; target = 25`

$\texttt{target} = 25 \implies \texttt{distance} = 2^{\texttt{target}} = 2^{25} = 33554432$

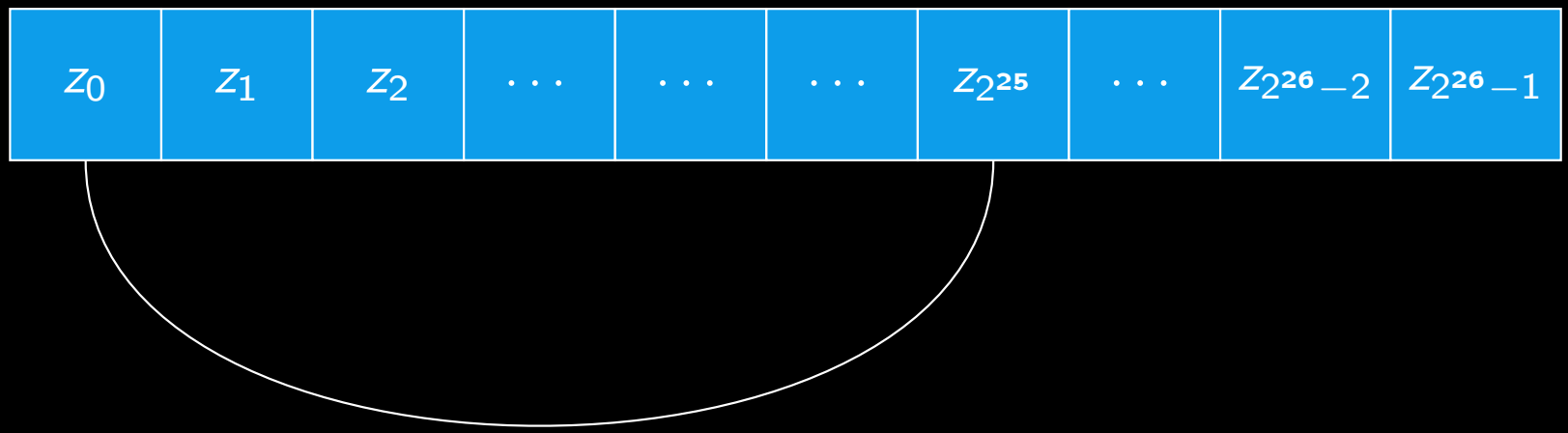
Consider: `n = 26; target = 25`

$\texttt{target} = 25 \implies \texttt{distance} = 2^{\texttt{target}} = 2^{25} = 33554432$



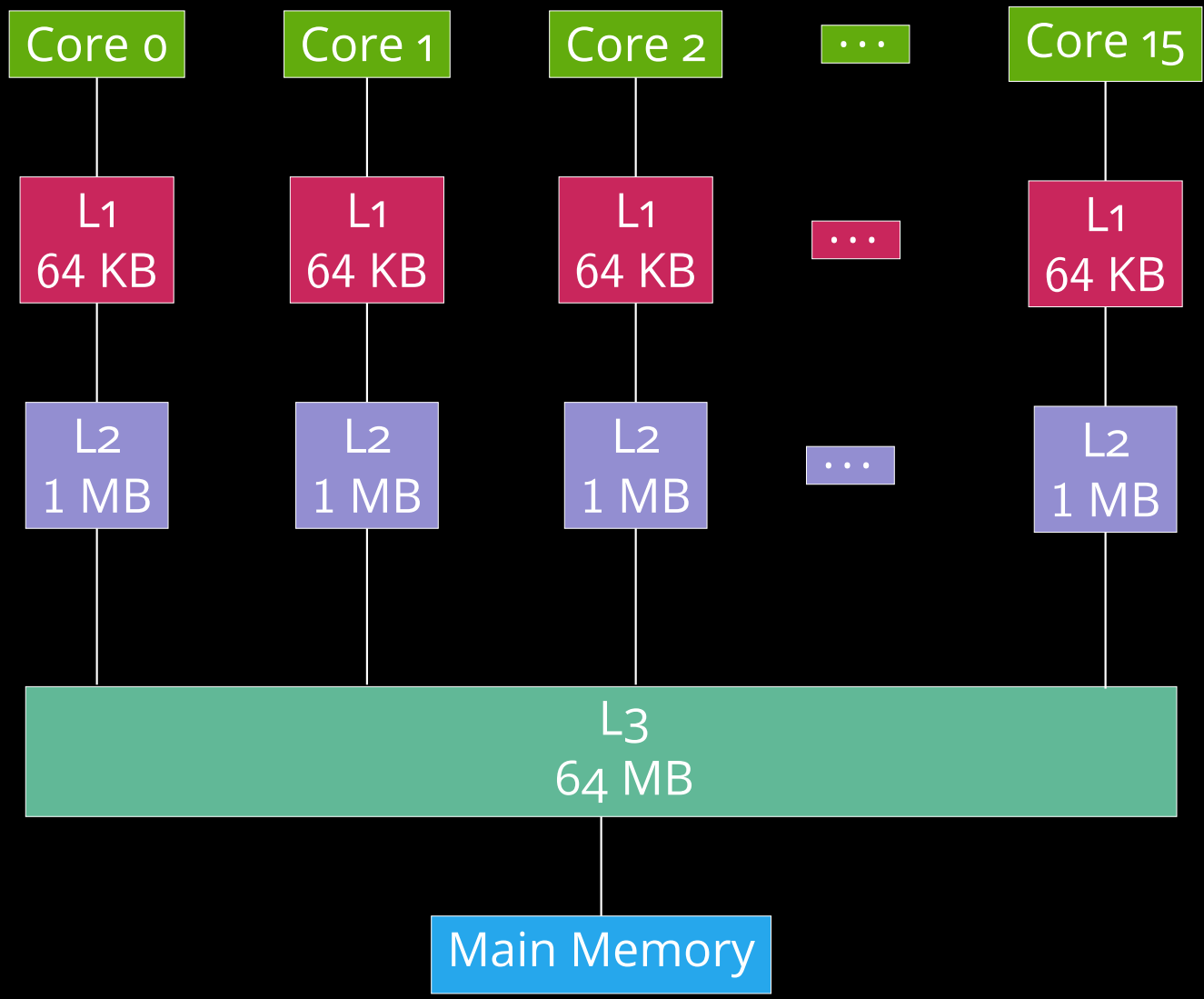
Consider: `n = 26; target = 25`

$\texttt{target} = 25 \implies \texttt{distance} = 2^{\texttt{target}} = 2^{25} = 33554432$



pairs are $33,554,432$ elements apart!

Cache Rules Everything Around Me

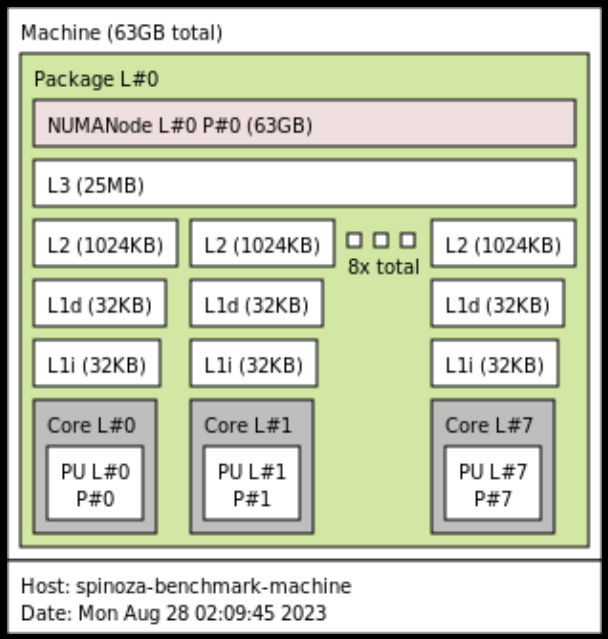


Cache Rules Everything Around Me

Op	Time	Comparison
L1 Cache Reference	0.5 ns	
Branch Mispredict	5 ns	
L2 Cache Reference	7 ns	14× L1 Cache
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	20× L2 Cache, 200× L1 Cache

https://gist.github.com/jboner/2841832

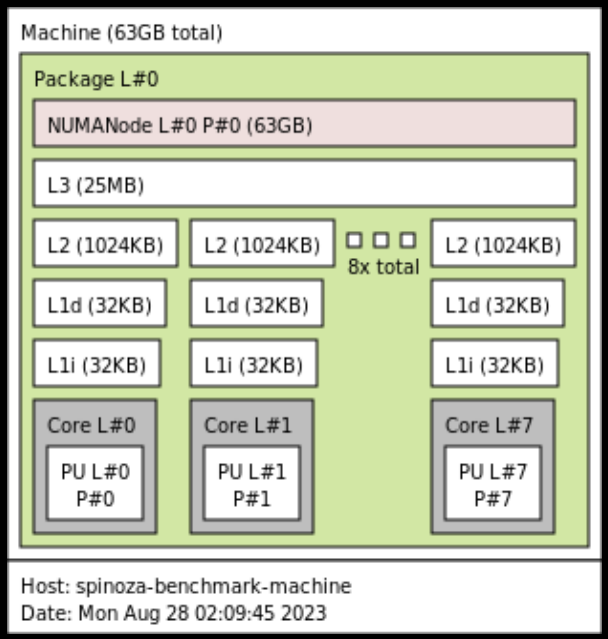
Cache Rules Everything Around Me



Consider:

cache misses as # of qubits increases and as the target qubit increases

Cache Rules Everything Around Me



Consider:

cache misses as # of qubits increases

and as the target qubit increases

- 16 qubits $\implies 2^{16} \cdot 128$ bits ≈ 1.049 MB
- 20 qubits $\implies 2^{20} \cdot 128$ bits ≈ 16.78 MB
- 25 qubits $\implies 2^{25} \cdot 128$ bits ≈ 536.9 MB

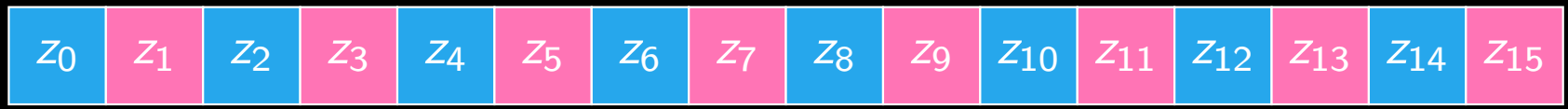
Certain gates are special

No need to have both sides of the pair!

Just scale the elements of the state individually!

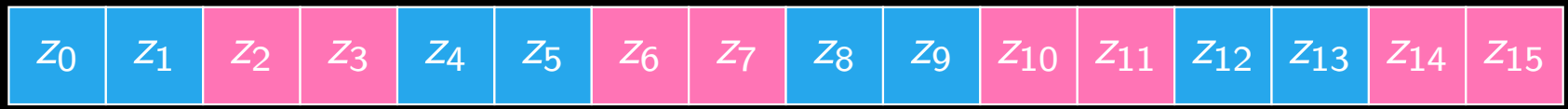
Strategy 1 — Group & Traverse

```
n = 4
target = 0
chunk_size = 1
```



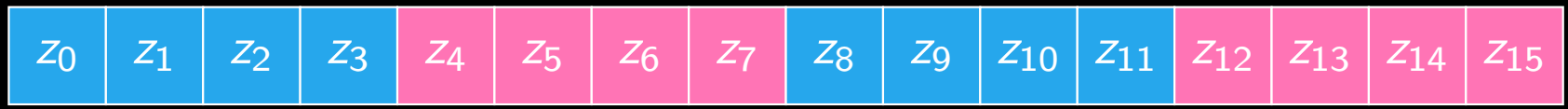
Strategy 1 — Group & Traverse

```
n = 4
target = 1
chunk_size = 2
```



Strategy 1 — Group & Traverse

```
n = 4
target = 2
chunk_size = 4
```



Strategy 1 — Group & Traverse

```
n = 4
target = 3
chunk_size = 2
```

Strategy 1 — Group & Traverse

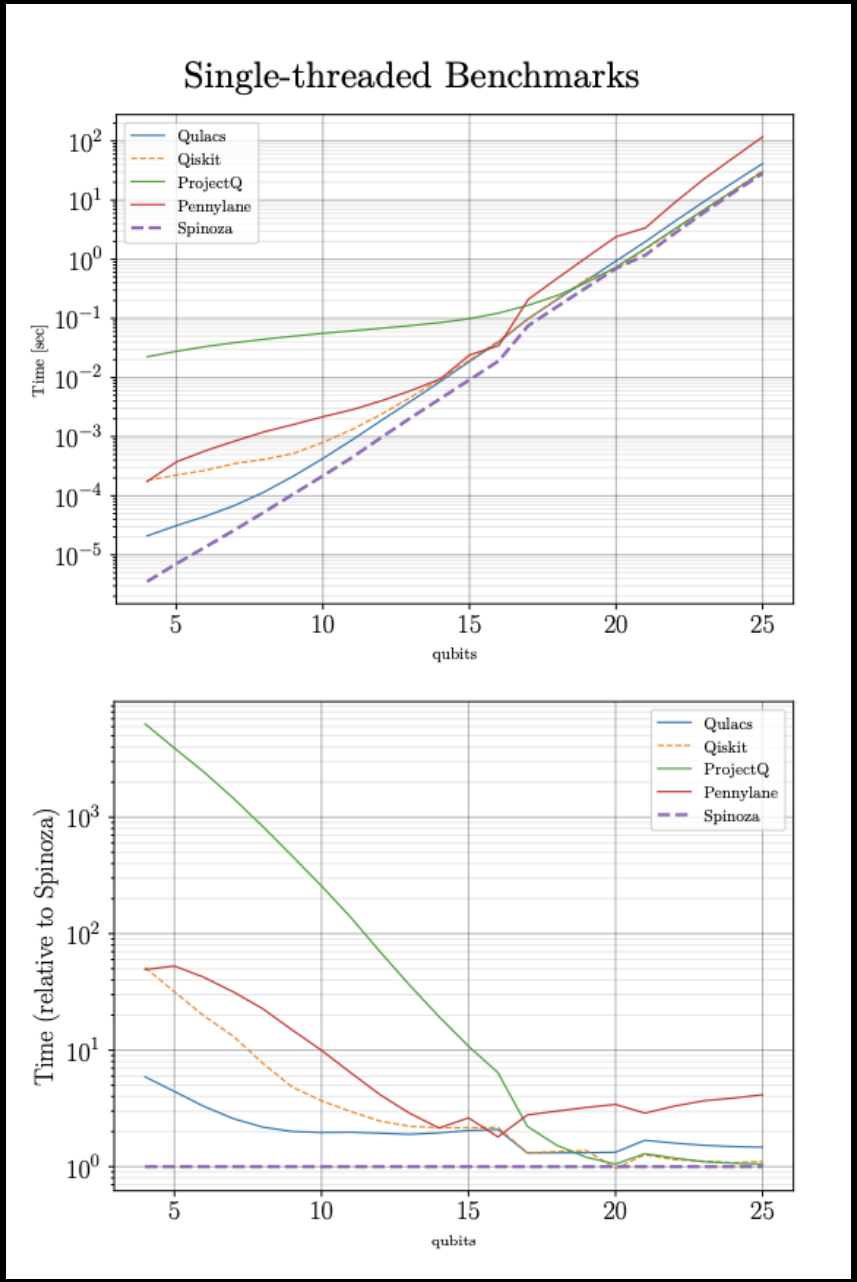
```rust
fn apply_strategy1(state: &mut State, target: usize, diag_matrix: &[Amplitude; 2]) {
    let chunk_size = 1 << target;
    state
        .reals
        .chunks_exact_mut(chunk_size)
        .zip(state.imags.chunks_exact_mut(chunk_size))
        .enumerate()
        .for_each(|(i, (c0, c1))| {
            c0.iter_mut().zip(c1.iter_mut()).for_each(|(a, b)| {
                let m = diag_matrix[i & 1];
                let c = *a;
                let d = *b;

                *a = c * m.re - d * m.im;
                *b = c * m.im + d * m.re;
            });
        });
}
```

Strategy 1 — Group & Traverse

```
vmovupd ymm3, ymmword ptr [r14 + 8*r12 - 32]
vmovupd ymm4, ymmword ptr [r14 + 8*r12]
vmovupd ymm5, ymmword ptr [r15 + 8*r12 - 32]
vmovupd ymm6, ymmword ptr [r15 + 8*r12]
vmulpd  ymm7, ymm1, ymm3
vmulpd  ymm8, ymm1, ymm4
vmulpd  ymm9, ymm2, ymm5
vsubpd  ymm7, ymm7, ymm9
vmulpd  ymm9, ymm2, ymm6
vsubpd  ymm8, ymm8, ymm9
vmovupd ymmword ptr [r14 + 8*r12 - 32], ymm7
vmovupd ymmword ptr [r14 + 8*r12], ymm8
vmulpd  ymm3, ymm2, ymm3
vmulpd  ymm4, ymm2, ymm4
vmulpd  ymm5, ymm1, ymm5
vaddpd  ymm3, ymm3, ymm5
vmulpd  ymm5, ymm1, ymm6
vaddpd  ymm4, ymm4, ymm5
vmovupd ymmword ptr [r15 + 8*r12 - 32], ymm3
vmovupd ymmword ptr [r15 + 8*r12], ymm4
```

Benchmarks


Single-threaded Benchmarks
Qulacs
Qiskit
ProjectQ
Pennylane
Spinoza
Time [sec]
qubits
Time (relative to Spinoza)
qubits

Acknowledgements

Constantin Gonciulea
Orson R. L. Peters
Shnatsel
Vitaliy Dorum
Many others!
Rust community

github.com/QuState/spinoza