Performance analysis of a state vector quantum circuit simulation on A64FX processor

Miwako Tsuji, Mitsuhisa Sato

RIKEN Center for Computational Science

Introduction

- Quantum computers (QCs) are systems that use the properties of quantum physics
- QCs are expected to perform the computations where classical computers (CCs) should show little growth
- Quantum computer simulators are important to
  - verify quantum computers
  - develop quantum applications
  - computational costs to simulate quantum computers will increase exponentially for the number of “qubits”
- Important to understand the features of quantum computer simulators
- Performance analysis of a quantum simulator “qulacs” on A64FX processor

Background

- Our target : Quantum computer simulators simulate quantum logic gates model QC
- a quantum bit (qubit) is a unit of quantum information, like a bit in CCs
  - a qubit represents a quantum state
  - an n-qubit state is represented by a vector composed of 2^n basis vectors
    - an array of 2^n complex numbers in QC simulators
- The quantum logic gates operate on the quantum state
- In the quantum computer simulators, each quantum logic gate operation scans and updates an array of 2^n complex numbers

- For A64FX, an Armv8.2-A with the scalable vector extinction (SVE),
  - important to make use of the 512-bit SIMD for quantum logic gate operations
  - some insights about an array of complex numbers on the A64FX

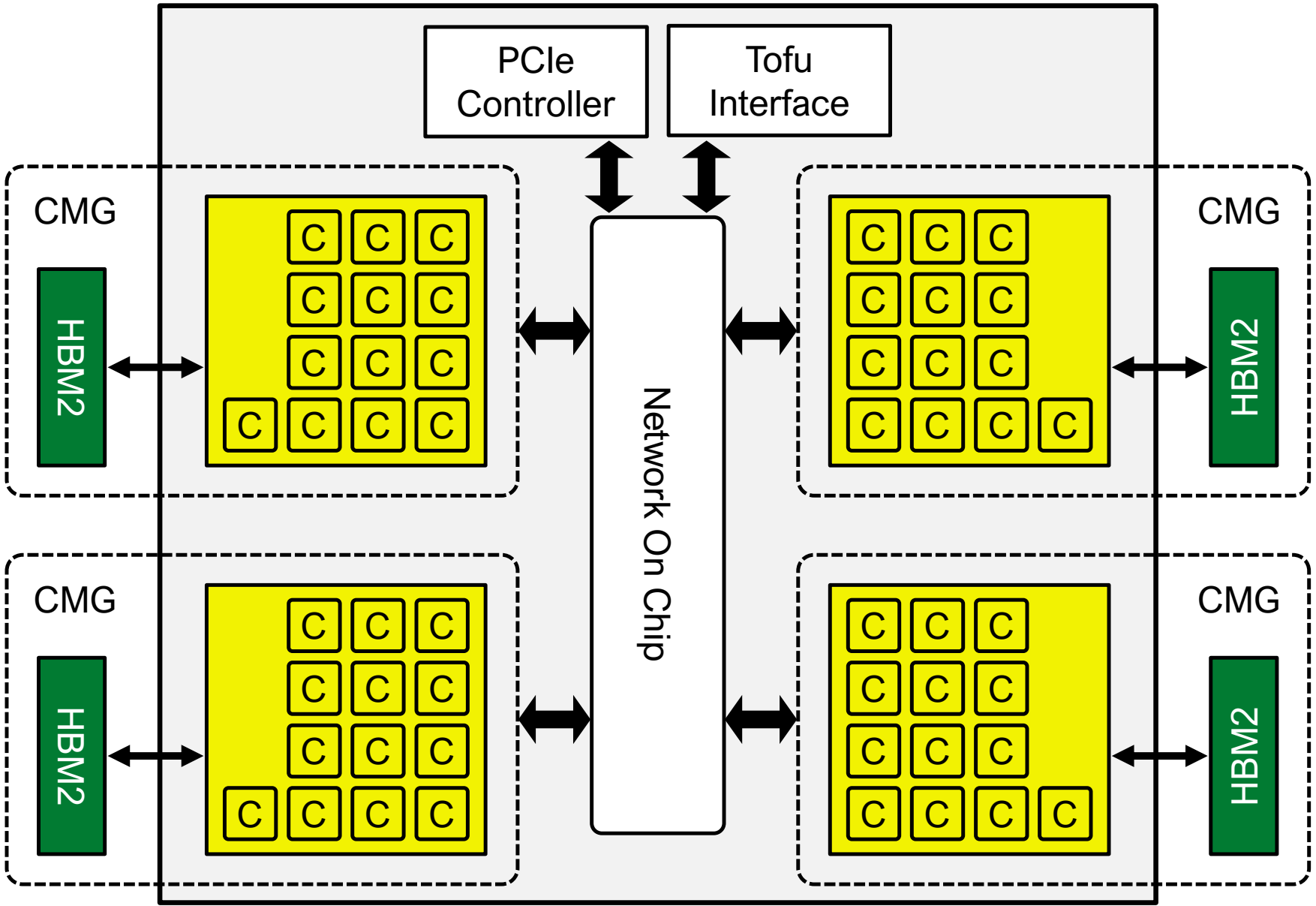
This talk focuses on..

- The performance of A64FX Armv8.2-a+SVE for quantum computer simulators
- Qulacs
  - a quantum circuit simulator
  - Implemented in C/C++ and with Python interface
  - GPU, **OpenMP**, x86 Built-in functions for AVX2
  - MPI: Imamura et. al. “mpiQulacs: A Distributed Quantum Computer Simulator for A64FX-based Cluster Systems (in Japanese)”, IPSJ, SIG HPC Technical Reports , 2022-HPC-185
  - https://github.com/qulacs/qulacs

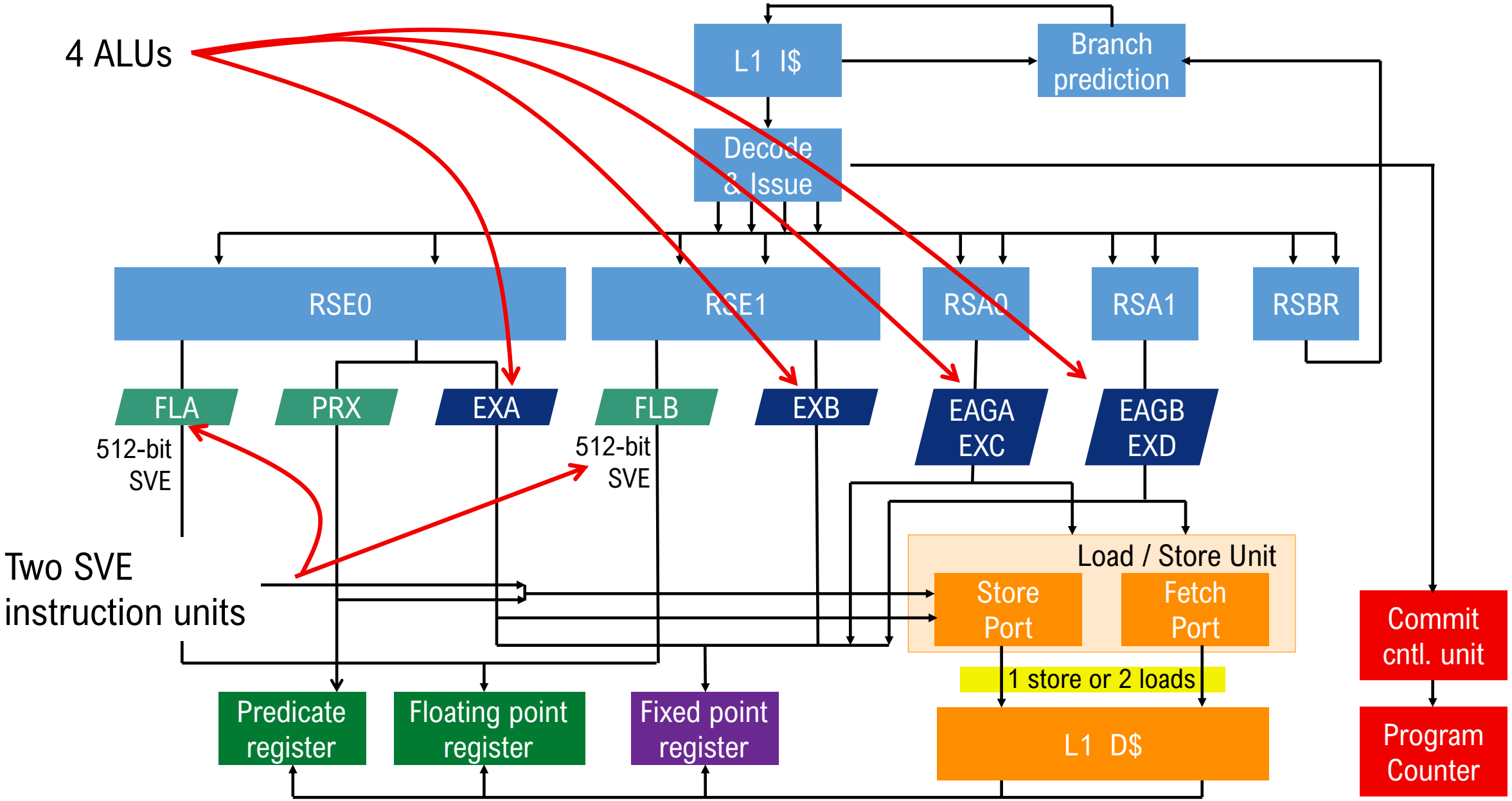
A64FX: Node Overview

C: Core
CMG: Core Memory Group
HBM: High Bandwidth Memory

- 4 CMG in a Node
  - 48 + 2/4 core
- SVE 512-bit wide SIMD
- FP64/FP32/FP16
- 12+1 cores in a CMG
  - an assistant core in each CMG
- HBM2 32GiB/Node
  - 8GiB/CMG
  - 1024 GB/s



A64FX: Core Overview



The gates used in our Benchmarks

gate	characteristic
H	single qubit gate
X	single qubit gate
Y	single qubit gate
Z	single qubit gate
RX	single-qubit rotation gate
RZ	single-qubit rotation gate
CNOT	two qubit gate
CZ	two qubit gate
SWAP	two qubit gate

- Single qubit gates act on a single qubit
- Single qubit rotation gates are the analog rotation matrices in three Cartesian axes of the 3D rotation group
- Two qubit gates act on two qubits

Benchmarks

- 11-hold repetition of a scan of each qubit of a gate
- Single qubit gate (ex. Hadamard gate and RZ gate)

```
for(int iter=0; iter<11; iter++){
  for(int i=0; i<nqubits; i++){
    circuit.add_H_gate(i);
  }
 }
 circuit.update_quantum_state(st);
```

```
for(int iter=0; iter<11; iter++){
  for(int i=0; i<nqubits; i++){
    circuit.add_RZ_gate(angle,i);
  }
 }
 circuit.update_quantum_state(st);
```

- Two qubit gate (ex. CZ gate)

```
ictr = number_of_qubits/2;
for(int iter=0; iter<11; iter++){
 for(int i=0; i<nqubits; i++){
   if(i!=ictr)
   circuit.add_CZ_gate(i, ictr);
 }
}
circuit.update_quantum_state(st);
```

An example of gate operations in QC simulation

```
void H_gate(UINT target_qubit_index, CTYPE *state, ITYPE dim) {
  const ITYPE loop_dim = dim / 2;
  const ITYPE mask = (1ULL << target_qubit_index);
  const ITYPE lmask = mask - 1;  //
  const ITYPE hmask = ~mask_low;
  const double sqrt2inv = 1. / sqrt(2.);
  if (target_qubit_index == 0) {
#pragma omp parallel for
    for (ITYPE idx = 0; idx < dim; idx += 2) {
      CTYPE temp0 = state[idx];
      CTYPE temp1 = state[idx+1];
      state[idx] = (temp0 + temp1)*sqrt2inv;
      state[idx + 1] = (temp0 - temp1)*sqrt2inv;
    }
  }else {
#pragma omp parallel for
    for (ITYPE idx = 0; idx < loop_dim; idx += 2) {
      ITYPE bidx_0 = (idx&lmask) + ((idx&hmask) << 1);
      ITYPE bidx_1 = bidx_0 + mask;
      CTYPE temp_a0 = state[bidx_0];     CTYPE temp_a1 = state[bidx_1];
      CTYPE temp_b0 = state[bidx_0 + 1]; CTYPE temp_b1 = state[bidx_1 + 1];
      state[bidx_0] = (temp_a0 + temp_a1)*sqrt2inv;
      state[bidx_1] = (temp_a0 - temp_a1)*sqrt2inv;
      state[bidx_0 + 1] = (temp_b0 + temp_b1)*sqrt2inv;
      state[bidx_1 + 1] = (temp_b0 - temp_b1)*sqrt2inv;
```

2^qubits

an array of 2^qubits
complex number
2*2^qubits*sizeof(double)

```
lmask = mask - 1;
// 000……000111 // target_qubit 1s from low
hmask = ~mask_low;
// 111……111000 // target_qubit 0s from low
```

An example of gate operations in QC simulation

```
void H_gate(UINT target_qubit_index, CTYPE *state, ITYPE dim) {
  const ITYPE loop_dim = dim / 2;
  const ITYPE mask = (1ULL << target_qubit_index);
  const ITYPE lmask = mask - 1;
  const ITYPE hmask = ~mask_low;
  const double sqrt2inv = 1. / sqrt(2.);
  if (target_qubit_index == 0) {
#pragma omp parallel for
    for (ITYPE idx = 0; idx < dim
      CTYPE temp0 = state[idx];
      CTYPE temp1 = state[idx+1]
      state[idx] = (temp0 + temp1
      state[idx + 1] = (temp0 -
    }
  }else {
#pragma omp parallel for
    for (ITYPE idx = 0; idx < loop_dim; idx += 2) {
      ITYPE bidx_0 = (idx&lmask) + ((idx&hmask) << 1);
      ITYPE bidx_1 = bidx_0 + mask;
      CTYPE temp_a0 = state[bidx_0];     CTYPE temp_a1 = state[bidx_1];
      CTYPE temp_b0 = state[bidx_0 + 1]; CTYPE temp_b1 = state[bidx_1 + 1];
      state[bidx_0] = (temp_a0 + temp_a1)*sqrt2inv;
      state[bidx_1] = (temp_a0 - temp_a1)*sqrt2inv;
      state[bidx_0 + 1] = (temp_b0 + temp_b1)*sqrt2inv;
      state[bidx_1 + 1] = (temp_b0 - temp_b1)*sqrt2inv;
```

2^qubits

an array of 2^qubits complex number
2*2^qubits*sizeof(double)

FCC -Kfast: ”SIMD conversion is not applied to this loop because the uncertain order of the definition and reference to variable 'state' may cause different results from serial execution.”

not vectorized

An example of gate operations in QC simulation

```
void H_gate(UINT target_qubit_index, CTYPE * restrict state, ITYPE dim) {
  const ITYPE loop_dim = dim / 2;
..
```

This may also help

Declares array for which the loop slice optimization can be performed.

Note: omp simd also enhances the vectorization, but other optimizations may not be applied

```
  }else {
#pragma omp parallel for
#ocl loop norecurrence
    for (ITYPE idx = 0; idx < loop_dim; idx += 2) {
      ITYPE bidx_0 = (idx&lmask) + ((idx&hmask) << 1);
      ITYPE bidx_1 = bidx_0 + mask;
      CTYPE temp_a0 = state[bidx_0];     CTYPE temp_a1 = state[bidx_1];
      CTYPE temp_b0 = state[bidx_0 + 1]; CTYPE temp_b1 = state[bidx_1 + 1];
      state[bidx_0] = (temp_a0 + temp_a1)*sqrt2inv;
      state[bidx_1] = (temp_a0 - temp_a1)*sqrt2inv;
      state[bidx_0 + 1] = (temp_b0 + temp_b1)*sqrt2inv;
      state[bidx_1 + 1] = (temp_b0 - temp_b1)*sqrt2inv;
```

ACLE Implementations

- ACLE: Arm C Language Extension
  - specifies source language extensions and implementation choices that C/C++ compilers can implement in order to explicitly implement vectorization and parallelization
  - whereas ACEL supports SVE features, we focus on the 512-bit implementation here

ACLE Implementations, general case

```
1  const ITYPE loop_dim = dim / 2;
2  const ITYPE mask = (1ULL << target_qubit_index);
3  const ITYPE lmask = mask - 1;
4  const ITYPE hmask = ~lmask;
5  const double sqrt2inv = 1. / sqrt(2.);
6  if (target_qubit_index == 0) {
7    for (idx = 0; idx < dim; idx += 2) {
8       CTYPE temp0  = state[idx];
9       CTYPE temp1  = state[idx+1];
10      state[idx]   = (temp0 + temp1)*sqrt2inv;
11      state[idx+1] = (temp0 - temp1)*sqrt2inv;  }
12 } else {
13   for (idx = 0; idx < loop_dim; idx += 2) {
14      // compute base indexes from the idx and masks
15      ITYPE bidx_0 = (idx&lmask)+((idx&hmask)<< 1);
16      ITYPE bidx_1 = bidx_0 + mask;
17      CTYPE a0 = state[bidx_0];
18      CTYPE a1 = state[bidx_1];
19      CTYPE b0 = state[bidx_0 + 1];
20      CTYPE b1 = state[bidx_1 + 1];
21      state[bidx_0] = (a0 + a1)*sqrt2inv;
22      state[bidx_1] = (a0 - a1)*sqrt2inv;
23      state[bidx_0 + 1] = (b0 + b1)*sqrt2inv;
24      state[bidx_1 + 1] = (b0 - b1)*sqrt2inv;  }}
```

```
// p1 = svptrue_b64(), all active predicate
uint64_t istate = 0;
uint64_t jstate = istate+mask;
// copy a mask value to 8 elements in a register
svuint64_t svlmask = svdup_u64(mask_low);
svuint64_t svhmask = svdup_u64(mask_high);
for (uint64_t idx = 0; idx < loop_dim; idx += 8) {
  double *ptr0 = (double*)&(state[istate]);
  double *ptr1 = (double*)&(state[jstate]);
  svuint64_t sidx0 = svindex_u64(idx, 1);
  svuint64_t sidx1 = svand_x(p1, sidx0, svlmask);
  svuint64_t sidx2 = svand_x(p1, sidx0, svhmask);
  svuint64_t sidx3 = svlsl_x(p1, sidx2, 1);
  svuint64_t sidx4 = svadd_x(p1, sidx1, sidx3);
  // 2 * sizeof(double)
  svuint64_t sidx5 = svlsl_x(p1, sidx4, 4);
  // load 8 real numbers
  svfloat64_t vstate0r
        = svld1_gather_offset(p1, ptr0,   sidx5);
  // load 8 imag numbers
  svfloat64_t vstate0i
        = svld1_gather_offset(p1, ptr0+1, sidx5);
  ....
```

ACLE Implementations, general case

```
1  const ITYPE loop_dim = dim / 2;
2  const ITYPE mask = (1ULL << target_qubit_index);
3  const ITYPE lmask = mask - 1;
4  const ITYPE hmask = ~lmask;
5  const double sqrt2inv = 1. / sqrt(2.);
6  if (target_qubit_index == 0) {
7    for (idx = 0; idx < dim; idx += 2) {
8       CTYPE temp0  = state[idx];
9       CTYPE temp1  = state[idx+1];
10      state[idx]   = (temp0 + temp1)*sqrt2inv;
11      state[idx+1] = (temp0 - temp1)*sqrt2inv;  }
12 } else {
13   for (idx = 0; idx < loop_dim; idx += 2) {
14     // compute base indexes from the idx and masks
15     ITYPE bidx_0 = (idx&lmask)+((idx&hmask)<< 1);
16     ITYPE bidx_1 = bidx_0 + mask;
17     CTYPE a0 = state[bidx_0];
18     CTYPE a1 = state[bidx_1];
19     CTYPE b0 = state[bidx_0 + 1];
20     CTYPE b1 = state[bidx_1 + 1];
21     state[bidx_0] = (a0 + a1)*sqrt2inv;
22     state[bidx_1] = (a0 - a1)*sqrt2inv;
23     state[bidx_0 + 1] = (b0 + b1)*sqrt2inv;
24     state[bidx_1 + 1] = (b0 - b1)*sqrt2inv;  }}
```

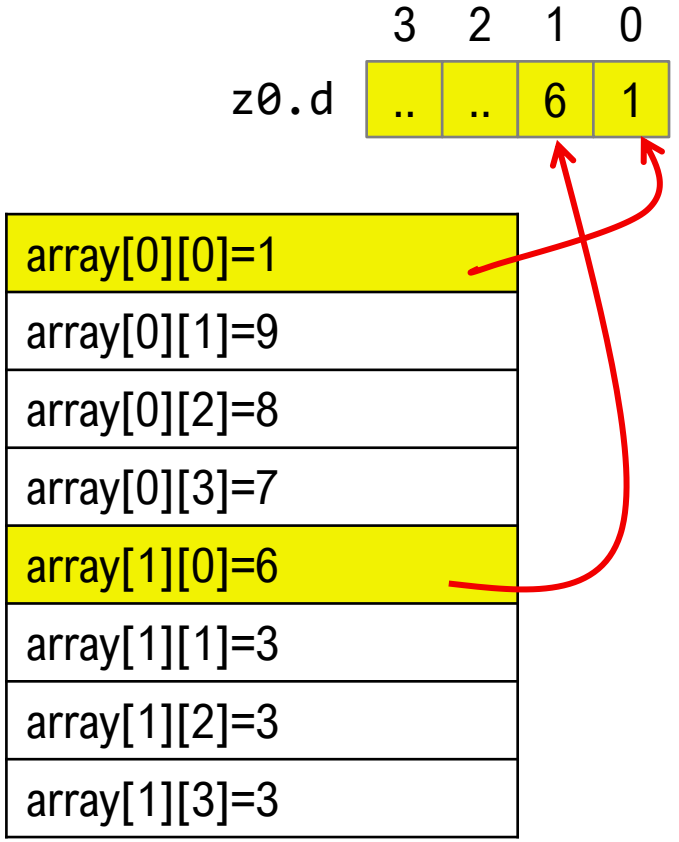
```
// p1 = svptrue_b64(), all active predicate
uint64_t istate = 0;
uint64_t jstate = istate+mask;
// copy a mask value to 8 elements in a register
svuint64_t svlmask = svdup_u64(mask_low);
svuint64_t svhmask = svdup_u64(mask_high);
for (uint64_t idx = 0; idx < loop_dim; idx += 8) {
  double *ptr0 = (double*)&(state[istate]);
  double *ptr1 = (double*)&(state[jstate]);
  svuint64_t sidx0 = svindex_u64(idx, 1);
  svuint64_t sidx1 = svand_x(p1, sidx0, svlmask);
  svuint64_t sidx2 = svand_x(p1, sidx0, svhmask);
  svuint64_t sidx3 = svlsl_x(p1, sidx2, 1);
  svuint64_t sidx4 = svadd_x(p1, sidx1, sidx3);
  // 2 * sizeof(double)
  svuint64_t sidx5 = svlsl_x(p1, sidx4, 4);
  // load 8 real numbers
  svfloat64_t vstate0r
        = svld1_gather_offset(p1, ptr0,   sidx5);
  // load 8 imag numbers
  svfloat64_t vstate0i
        = svld1_gather_offset(p1, ptr0+1, sidx5);
  ....
```

ACLE Implementations, gather load / scatter store

- Arm8.2+SVE supports gather load/scatter store
  - 1 instruction can read/write a list of non-continuous array

```
svld1_gather_offset(predicate, *base_address, offsets)
```

list of addresses



- latency is 15+ cycles
  - 9+ for contiguous load
- 1 SVE register is used to maintain a list of array
- 1 SVE instruction unit is used to compute the address of the elements

ACLE Implementations, special case

- special case, the operations are vectorized under a certain condition
  - the order of qubits and the corresponding quantum state are always in an ascending order
  - “regularity” can be used for the vectorization
  - especially, 4 complex values of a state[] vector can be contiguous
    - 8 real numbers (512-bit!)

```
for (ITYPE idx = 0; idx < loop_dim; idx+= 4) {
    ITYPE bidx_0 = ((idx  )&lmask) + (((idx  )&hmask) << 1);
    ITYPE bidx_1 = ((idx+1)&lmask) + (((idx+1)&hmask) << 1);
    ITYPE bidx_2 = ((idx+2)&lmask) + (((idx+2)&hmask) << 1);
    ITYPE bidx_3 = ((idx+3)&lmask) + (((idx+3)&hmask) << 1);
    CTYPE temp_a0 = state[bidx_0];
    CTYPE temp_a1 = state[bidx_1];
    CTYPE temp_a2 = state[bidx_2];
    CTYPE temp_a3 = state[bidx_3];
```

contiguous if target qubit>=2

```
lmask = mask - 1;
// 000……000111 // target_qubit 1s from low
hmask = ~mask_low;
// 111……111000 // target_qubit 0s from low
```

  - this is not the case for distributed parallel implementations, since the order of qubits should be changed to avoid redundant communication
  - using SVE feature makes it difficult to utilize the regularity ☹

ACLE Implementations, special case, target_qubit>=2

```
1  const ITYPE loop_dim = dim / 2;
2  const ITYPE mask = (1ULL << target_qubit_index);
3  const ITYPE lmask = mask - 1;
4  const ITYPE hmask = ~lmask;
5  const double sqrt2inv = 1. / sqrt(2.);
6  if (target_qubit_index == 0) {
7    for (idx = 0; idx < dim; idx += 2) {
8       CTYPE temp0  = state[idx];
9       CTYPE temp1  = state[idx+1];
10      state[idx]   = (temp0 + temp1)*sqrt2inv;
11      state[idx+1] = (temp0 - temp1)*sqrt2inv;  }
12 } else {
13   for (idx = 0; idx < loop_dim; idx += 2) {
14      // compute base indexes from the idx and masks
15      ITYPE bidx_0 = (idx&lmask)+((idx&hmask)<< 1);
16      ITYPE bidx_1 = bidx_0 + mask;
17      CTYPE a0 = state[bidx_0];
18      CTYPE a1 = state[bidx_1];
19      CTYPE b0 = state[bidx_0 + 1];
20      CTYPE b1 = state[bidx_1 + 1];
21      state[bidx_0] = (a0 + a1)*sqrt2inv;
22      state[bidx_1] = (a0 - a1)*sqrt2inv;
23      state[bidx_0 + 1] = (b0 + b1)*sqrt2inv;
24      state[bidx_1 + 1] = (b0 - b1)*sqrt2inv;  }}
```

```
// ‘‘p1’’ is a predicate indicating all active
for (idx = 0; idx < loop_dim; idx += 4) {
  ITYPE bidx_0=( idx   &lmask)+(( idx    &hmask)<< 1);
  ITYPE bidx_1=bidx_0 + mask;
  svfloat64_t st0, st1, st2, st3, st4, st5;
  // if target qubit >= 2, we can load 4 cont.
  // complex numbers (8 values) simultaneously
  double *ptr0 = (double *)&(state[bidx_0]);
  double *ptr1 = (double *)&(state[bidx_1]);
  st0 = svld1(p1, ptr0);
  st1 = svld1(p1, ptr1);
  st2 = svadd_x(p1, st0, st1);
  st3 = svsub_x(p1, st0, st1);
  st4 = svmul_x(p1, st2, sqrt2inv);
  st5 = svmul_x(p1, st3, sqrt2inv);
  svst1(p1, ptr0, st4);
  svst1(p1, ptr1, st5);
}
```

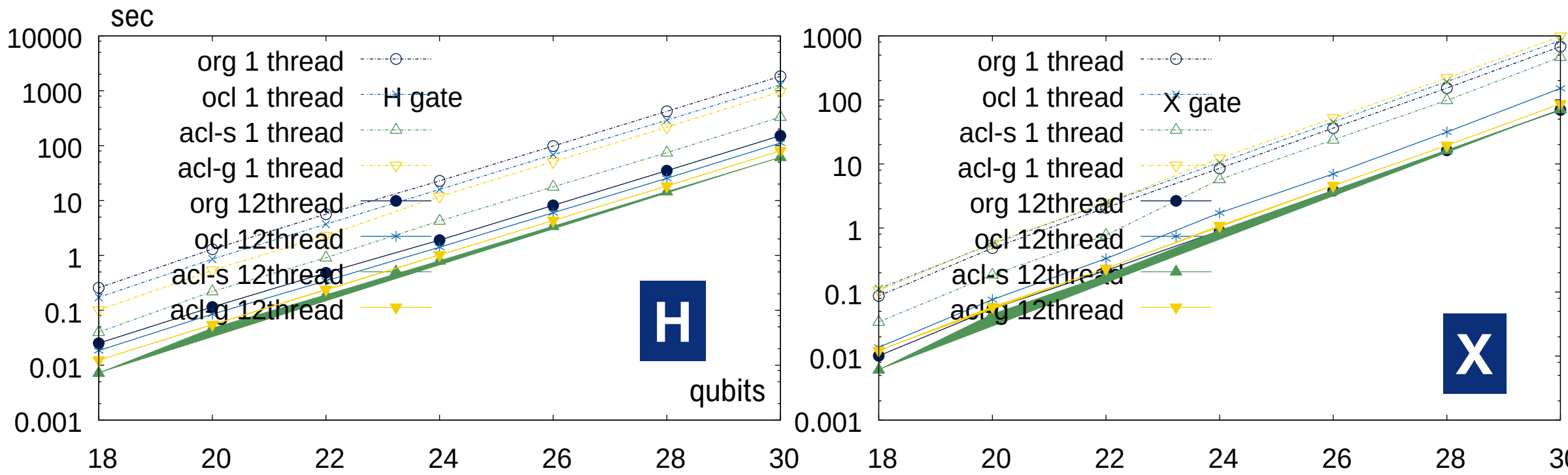
Environments

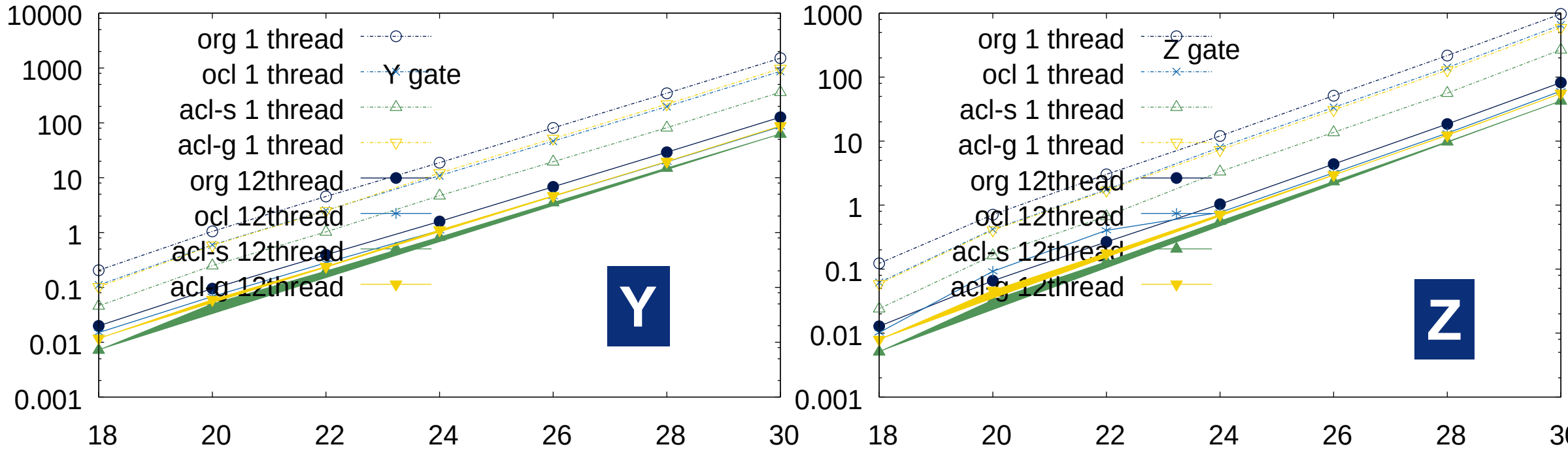
- Fujitsu C compiler for the gate operations, g++ for the other source files
  - fccpx 4.8.0 20220202, lang/tcsds-1.2.35
  - g++ 11.2.0
  - 1 thread, 12 thread
  - 18 to 30 qubits
- Comparison:
  - as-is
  - ocl (optimization control line) directive

    ```
    #ocl loop norecurrence
    ```
  - acle special case
  - acle general case

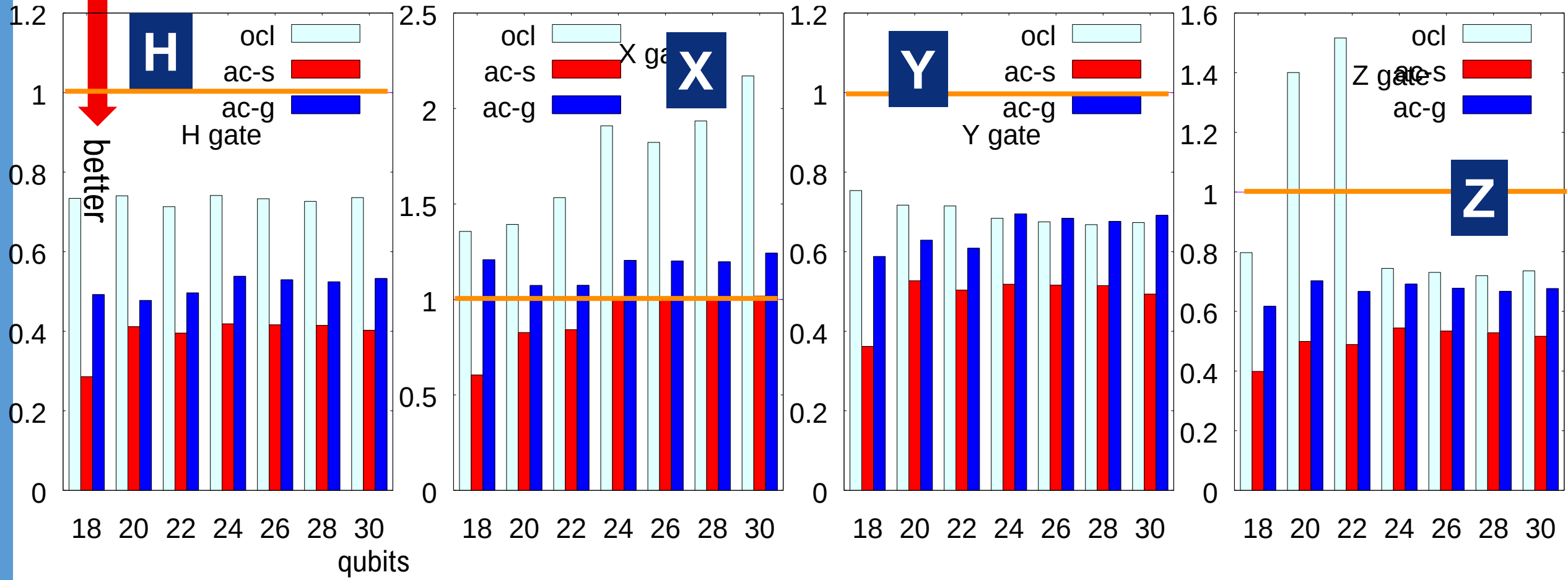
Exec. time : single qubit gates

better





Relative Exec. time : single qubit gates, **12threads** baseline:original code



- ocl & acle are almost always better than original code except X gate
- ACLE (special case) is the best
- ACLE (general case) and OCL are equivalent in Y&Z, but not in H & X
- vectorization does not work well for the X gate

of instructions in single qubit gates, 22qubits **1thread**

		Effective Instructions	Fp operations	SIMD (%) SIMD/Insts	SVE (%) SVE/fp
H	Original	15047480462	8120184180	26.24	4.55
	OCL	2394516308	8120184180	87.75	99.99
	ACLE, special	2127832675	4244647284	48.95	99.99
	ACLE, general	1664275862	4060097908	83.89	99.99
X	Original	6144333483	20	0.2418	0.00
	OCL	1104892771	20	75.33	0.00
	ACLE, special	1104892771	20	31.43	0.00
	ACLE, general	1147464709	20	77.44	0.00
Y	Original	14035070382	6090129428	20.9801	4.55
	OCL	1698667353	6090129428	84.2249	100.0
	ACLE, special	2341701275	2214592532	53.3387	100.0
	ACLE, general	1266949882	1015021588	80.1503	100.0
Z	Original	7656719259	2952790036	19.2132	1.563
	OCL	940681782	1015021588	72.7013	100.0
	ACLE, special	1326022461	1015021588	28.7408	100.0
	ACLE, general	1086673920	1015021588	75.9367	100.0

of instructions in single qubit gates, 22qubits **1thread**

		Effective Instructions	Fp operations	SIMD (%) SIMD/Insts	SVE (%) SVE/fp
H	Original	15047480462	8120184180	26.24	4.55
	OCL	2394516308	8120184180	87.75	99.99
	ACLE, special	2127832675	4244647284	48.95	99.99
	ACLE, general	1664275862			99.99
X	Original	6144333483			0.00
	OCL	1104892771			0.00
	ACLE, special	1104892771			0.00
	ACLE, general	1147464709			0.00
Y	Original	14035070382			4.55
	OCL	1698667353	6090129428	84.2249	100.0
	ACLE, special	2341701275	2214592532	53.3387	100.0
	ACLE, general	1266949882	1015021588	80.1503	100.0
Z	Original	7656719259	2952790036	19.2132	1.563
	OCL	940681782	1015021588	72.7013	100.0
	ACLE, special	1326022461	1015021588	28.7408	100.0
	ACLE, general	1086673920	1015021588	75.9367	100.0

Vectorization can reduce # of instructions.
8 instructions can be packed into a single SIMD instruction.

gates, 22qubits **1thread**

Hand-writing ACLE implementations can reduce # of (redundant) floating-point operations

Note: a SIMD fp instruction perform 8 fp operations

			Fp operations	SIMD (%) SIMD/Insts	SVE (%) SVE/fp
H			8120184180	26.24	4.55
			8120184180	87.75	99.99
			4244647284	48.95	99.99
			4060097908	83.89	99.99
X			20	0.2418	0.00
			20	75.33	0.00
			20	31.43	0.00
	ACLE, general	1147464709	20	77.44	0.00
Y	Original	14035070382	6090129428	20.9801	4.55
	OCL	1698667353	6090129428	84.2249	100.0
	ACLE, special	2341701275	2214592532	53.3387	100.0
	ACLE, general	1266949882	1015021588	80.1503	100.0
Z	Original	7656719259	2952790036	19.2132	1.563
	OCL	940681782	1015021588	72.7013	100.0
	ACLE, special	1326022461	1015021588	28.7408	100.0
	ACLE, general	1086673920	1015021588	75.9367	100.0

of instructions in single qubit gates, 22qubits **1thread**

		Effective Instructions	Fp operations	SIMD (%) SIMD/Insts	SVE (%) SVE/fp
H	Original	15047480462	8120184180	26.24	4.55
	OCL	2394516308	8120184180	87.75	99.99
	ACLE, special	2127832675	4244647284	48.95	99.99
				83.89	99.99
				0.2418	0.00
				75.33	0.00
				31.43	0.00
				77.44	0.00
				0.9801	4.55
				4.2249	100.0
				3.3387	100.0
				0.1503	100.0
Z	Original	7656719259	2952790036	19.2132	1.563
	OCL	940681782	1015021588	72.7013	100.0
	ACLE, special	1326022461	1015021588	28.7408	100.0
	ACLE, general	1086673920	1015021588	75.9367	100.0

The compiler uses different optimization paths for non-SIMD and SIMD code generations…

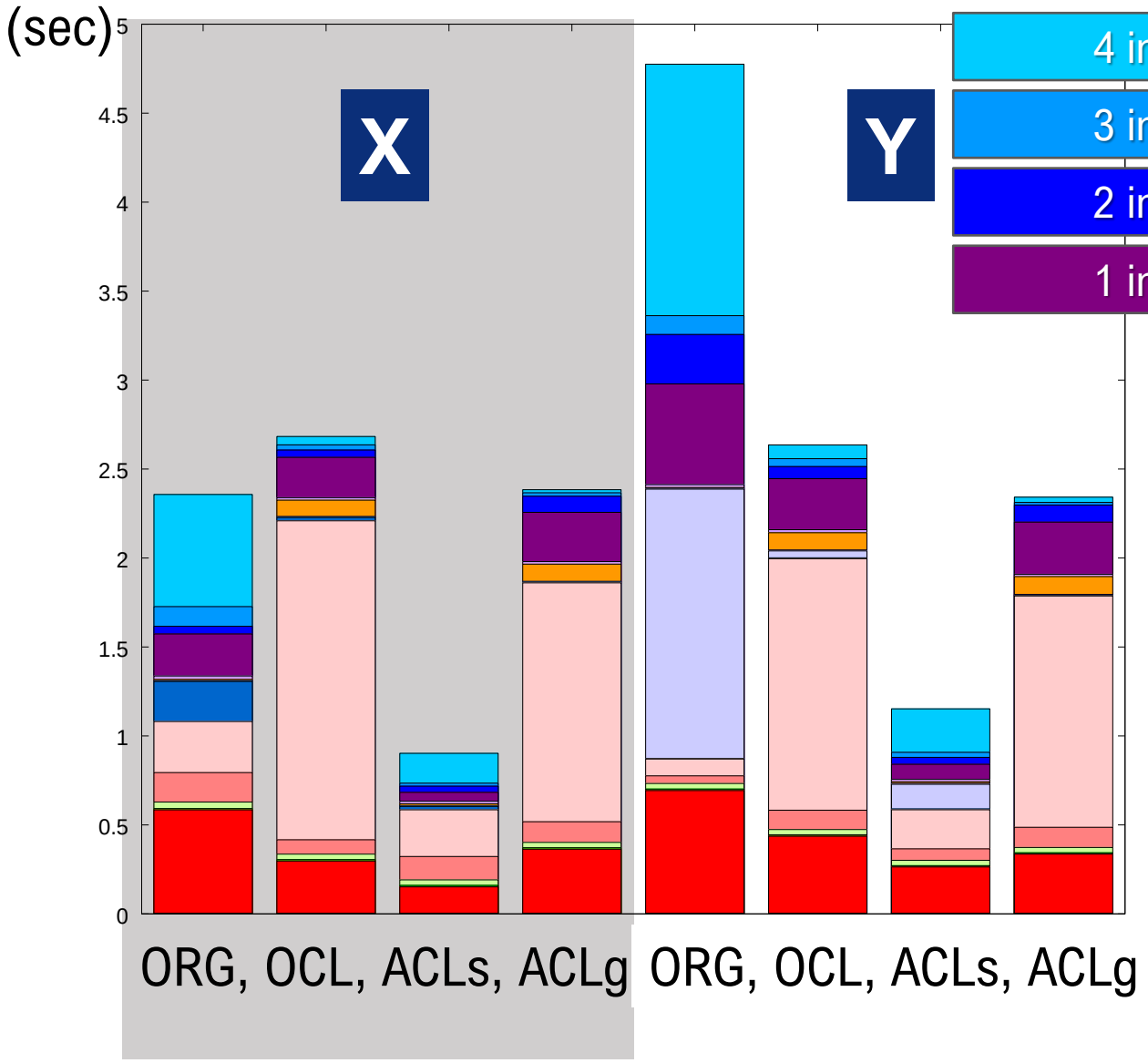
※ In this case, auto-vectorization can reduce # of fp operations, but vice versa

checking the # of instructions/operations is very easy in Fugaku. It is recemented to check

Cycle accounting execution time

- included in an analysis report generated by Fujitsu CPU performance analysis report
- aggregates CPU performance (such as cache misses, # of instructions) measured by multiple executions
- displays a breakdown of program execution time in terms of the events in a processor, such as instruction commit, instruction wait, etc…

Cycle accounting exec. time (22qubits, X&Y gates, 1threads)



4 instruction commit
3 instruction commit
2 instruction commit
1 instruction commit

Fp operation wait
Fp load L1D cache access
Fp load L2 cache access
Fp load memory access

- For X&Y gates, the vectorization can reduce the number of instructions, and the time to commit 1-4 instructions can be reduced.
- For X&Y gates, the time to wait L1D access increases after the vectorization, since the latency of the vector load is longer than that of the scaler load
- Tradeoff
  - # of instructions
  - L1 wait

Performance analysis of the **X** gate

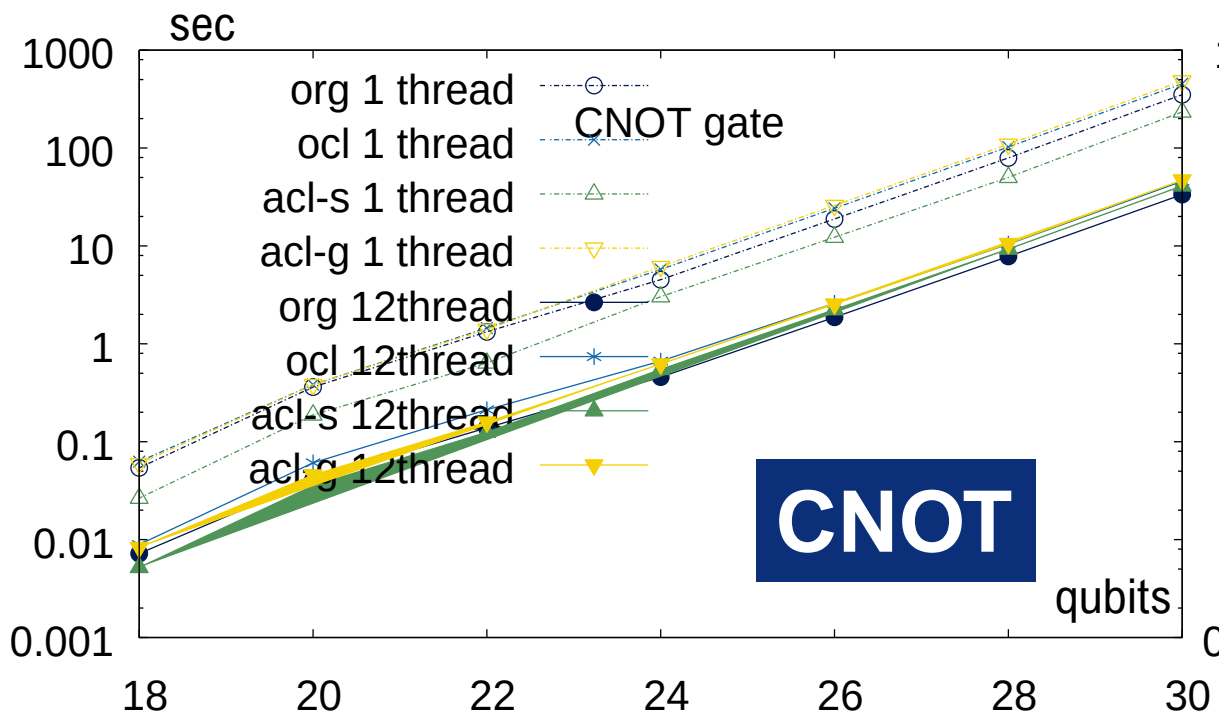
- X gate
  - X gate does not include any Arithmetic Instruction
  - Vectorization should not be beneficial
  - Vectorization simply increase the time to wait L1D access

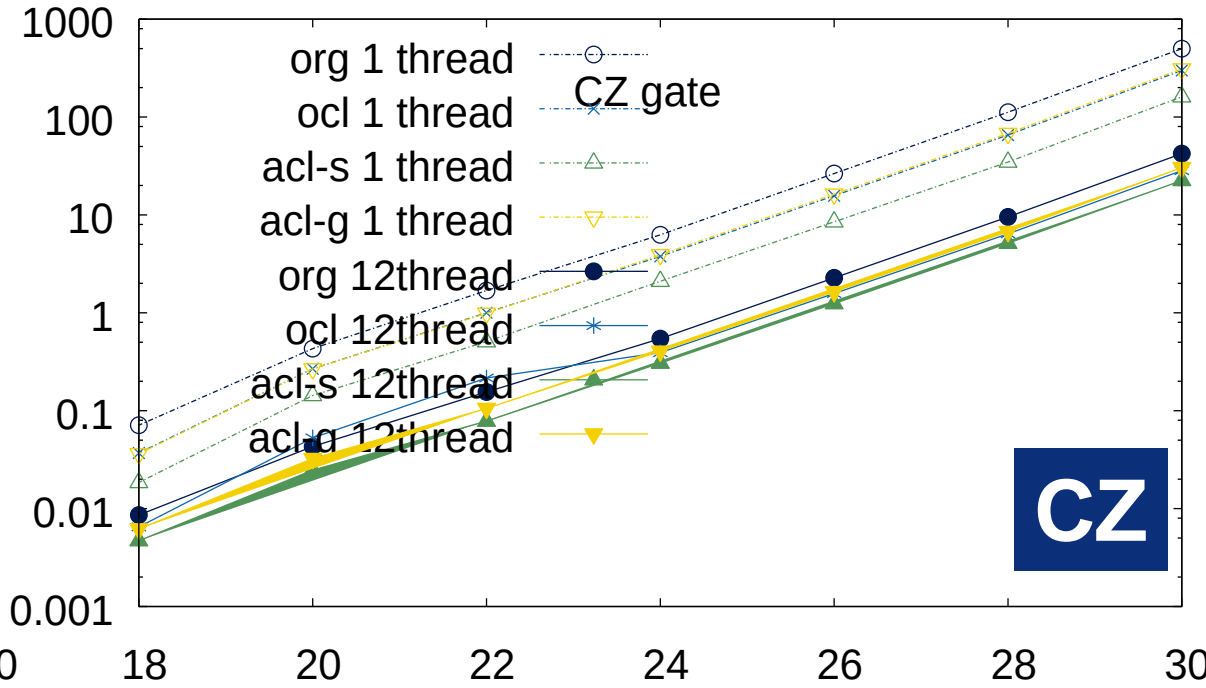
```
// X gate
for (state_index = 0; state_index < loop_dim; state_index += 2) {
      ITYPE basis_index_0 = (state_index&mask_low) + ((state_index&mask_high) << 1);
      ITYPE basis_index_1 = basis_index_0 + mask;
      CTYPE temp0 = state[basis_index_0];
      CTYPE temp1 = state[basis_index_0+1];
      state[basis_index_0] = state[basis_index_1];
      state[basis_index_0+1] = state[basis_index_1+1];
      state[basis_index_1] = temp0;
      state[basis_index_1+1] = temp1;
}
```

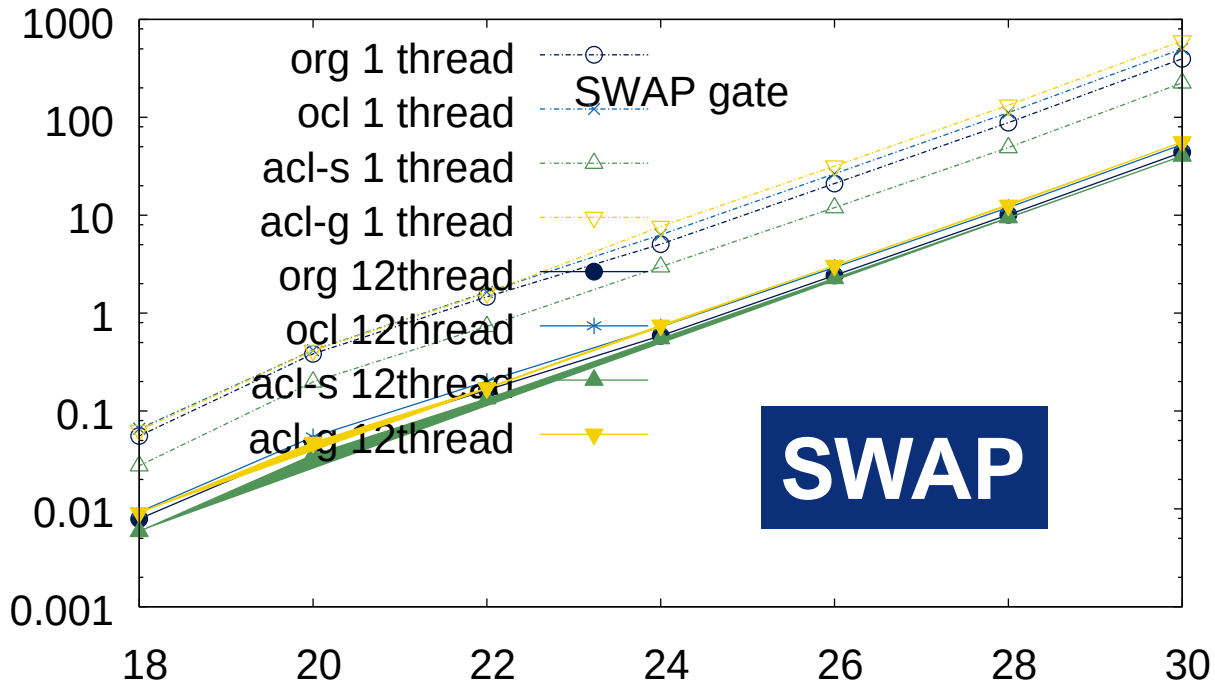
```
// Y gate multiplies 0-i or 0+i
state[basis_index_0] = -imag * state[basis_index_1];
state[basis_index_0+1] = -imag * state[basis_index_1+1];
state[basis_index_1] = imag * temp0;
state[basis_index_1+1] = imag * temp1;
```

Exec. time : two qubit gates

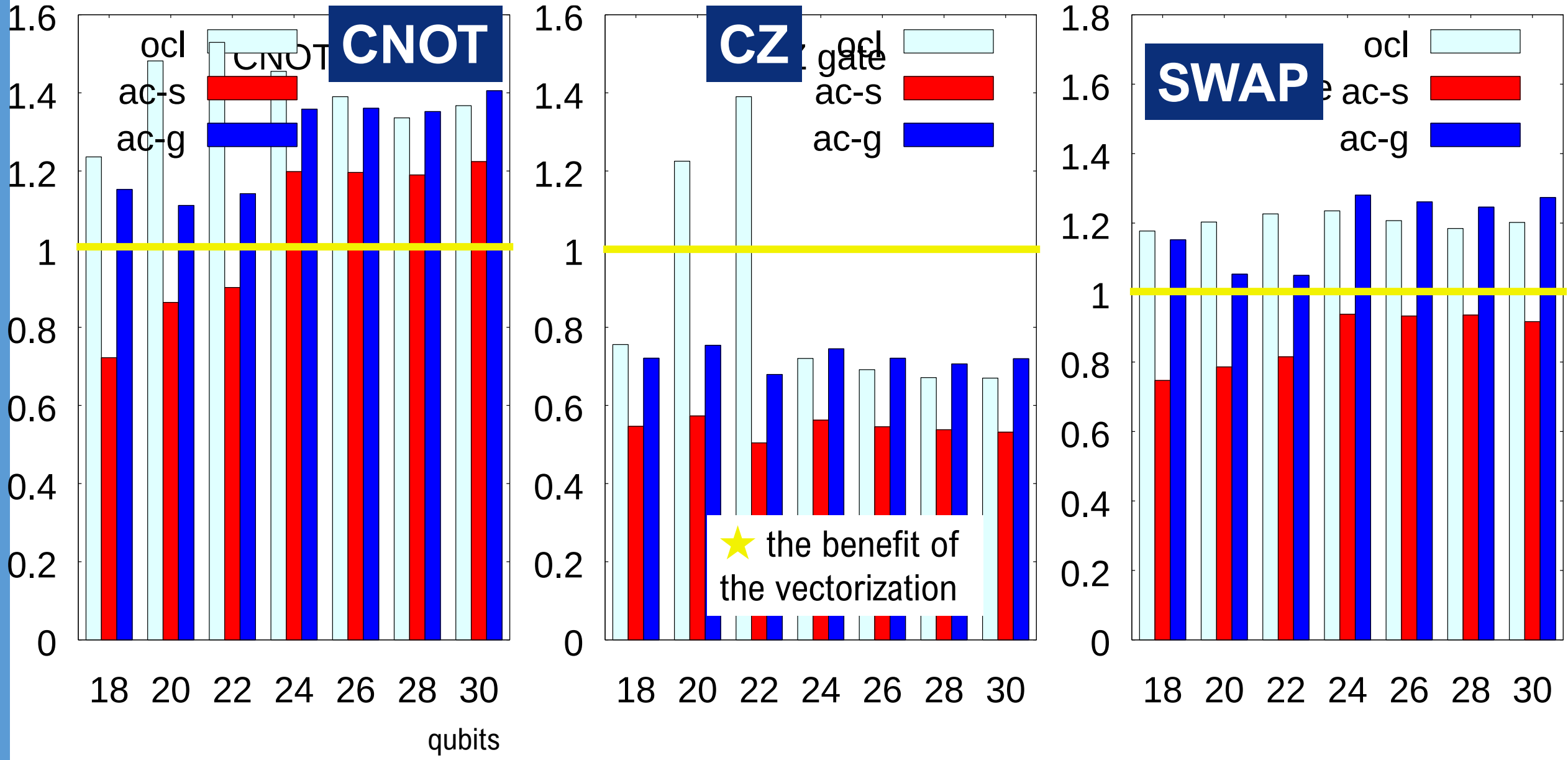
better



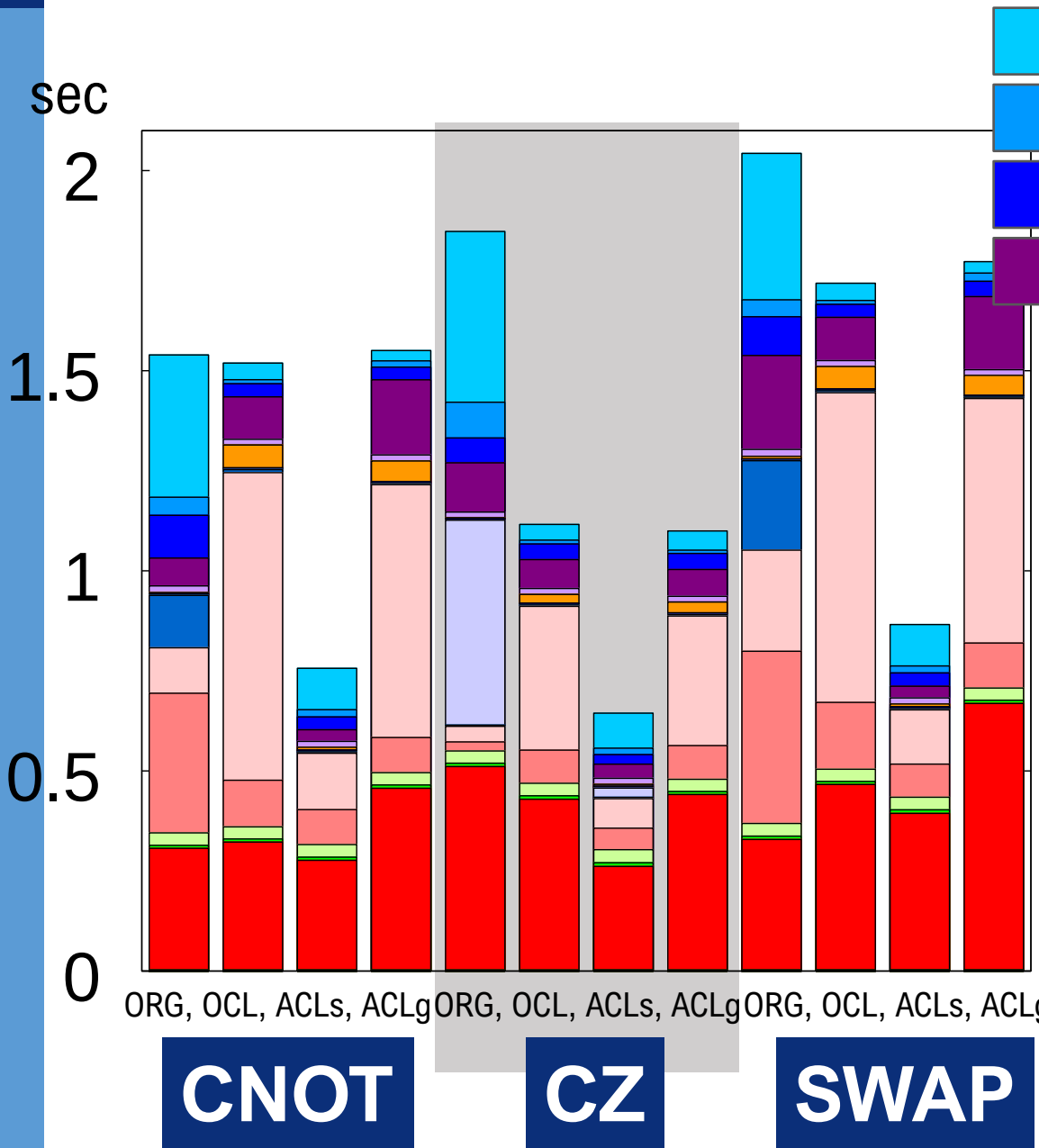


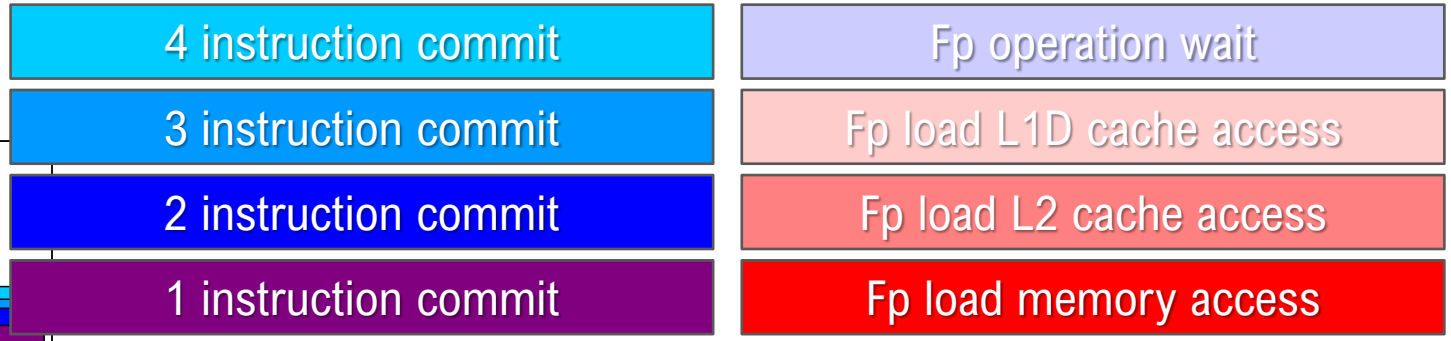


Relative Exec. time : two qubit gates, **12threads** baseline:original code



Cycle accounting exec. time (22qubits, two qubit gates, 1threads)

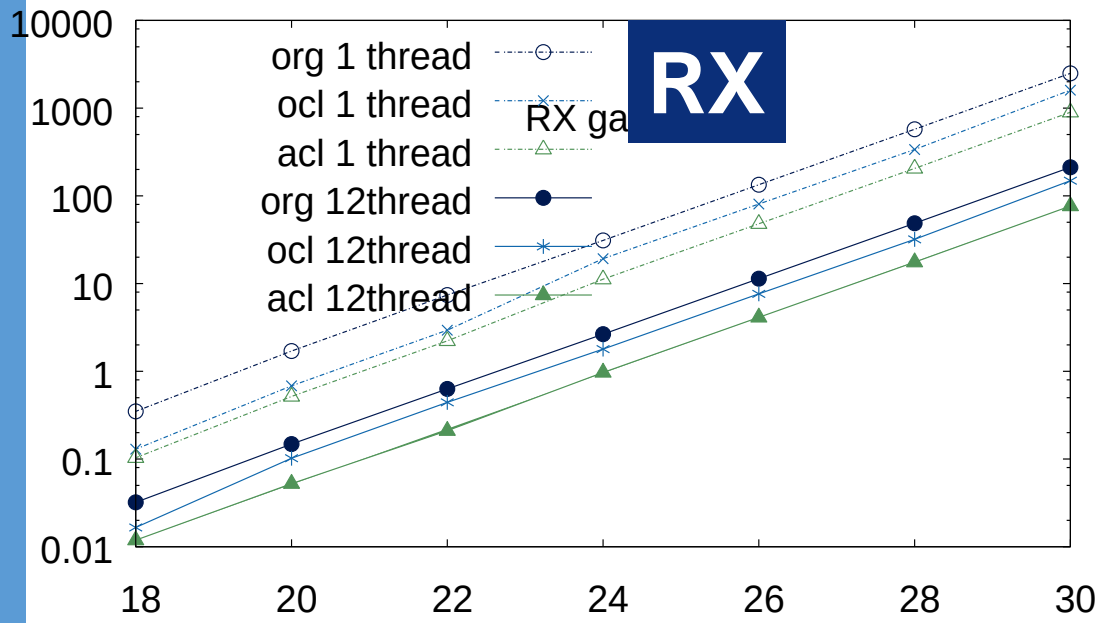


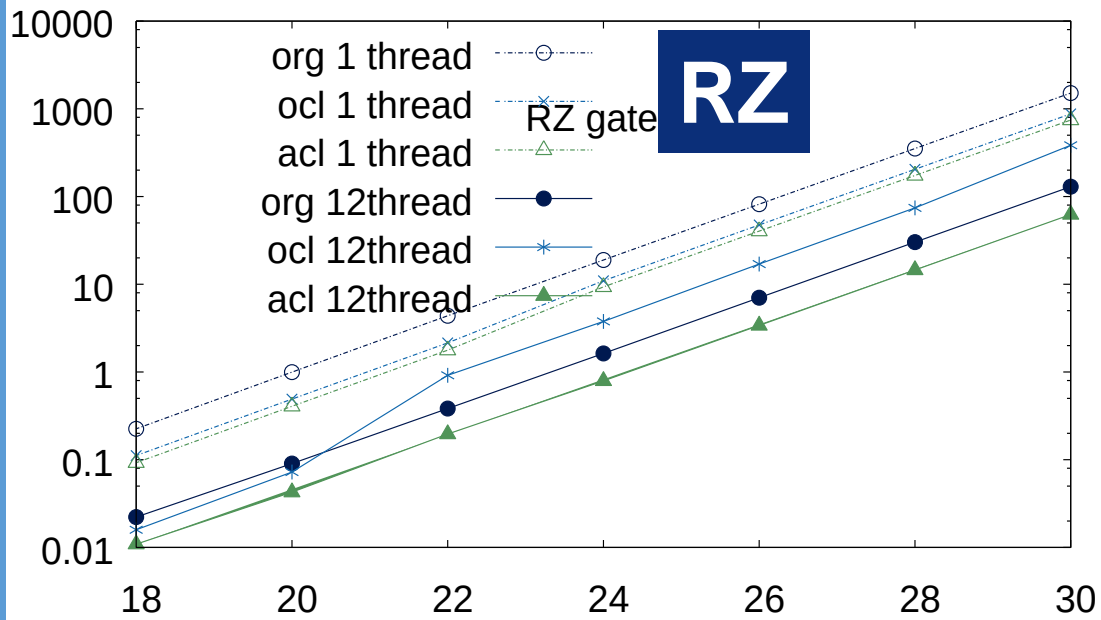


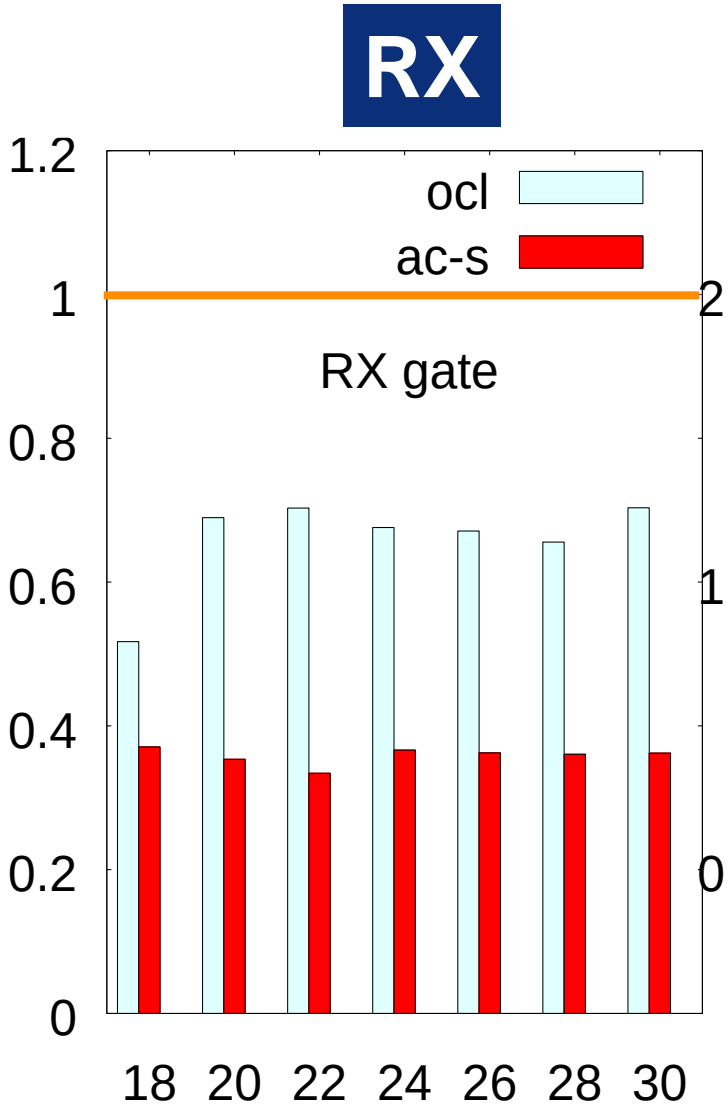
- Tradeoff to reduce the time to commit instructions and the time to wait L1D cache access

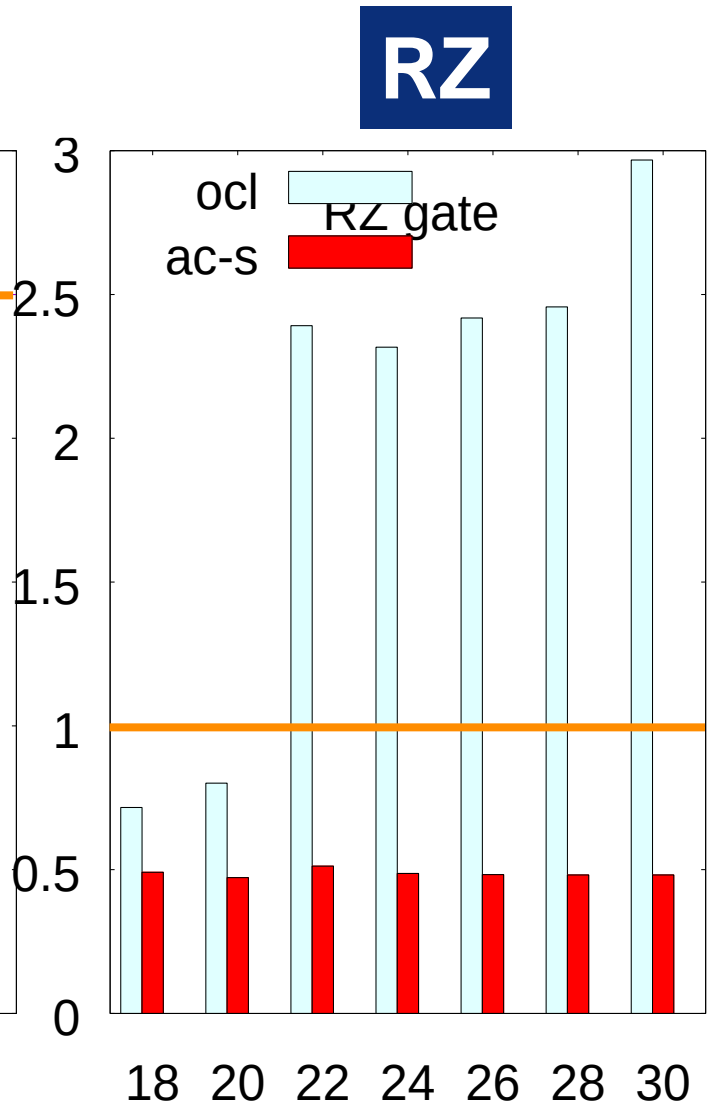
Exec. and relative exec time of single qubit w/ rotation gates

※ ACLE, general has been omitted



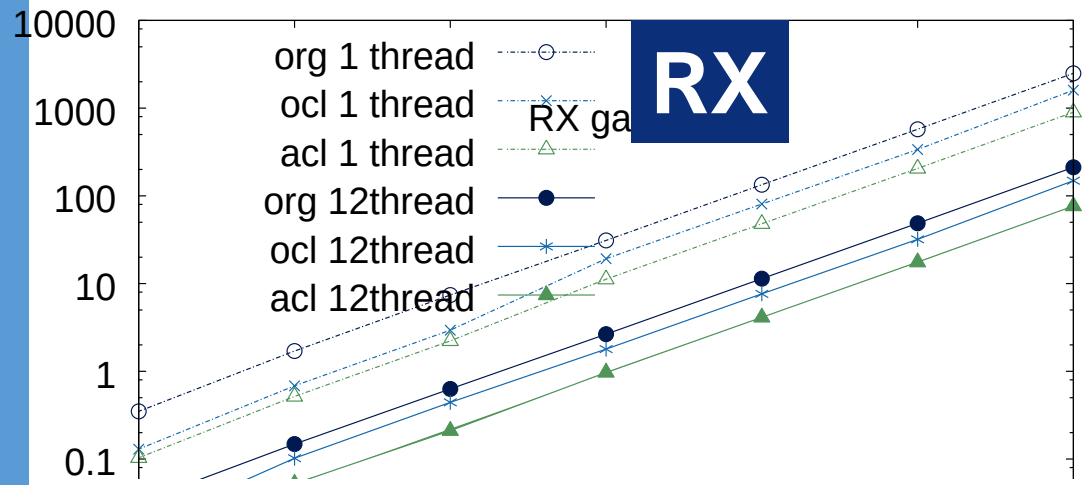




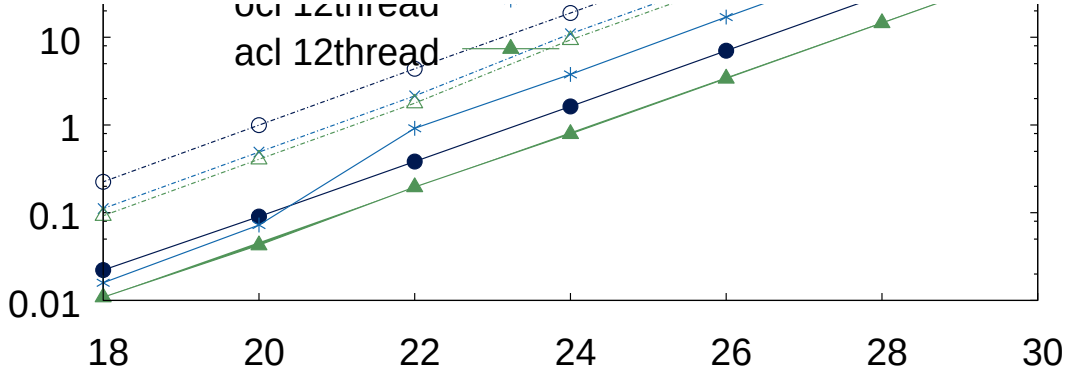


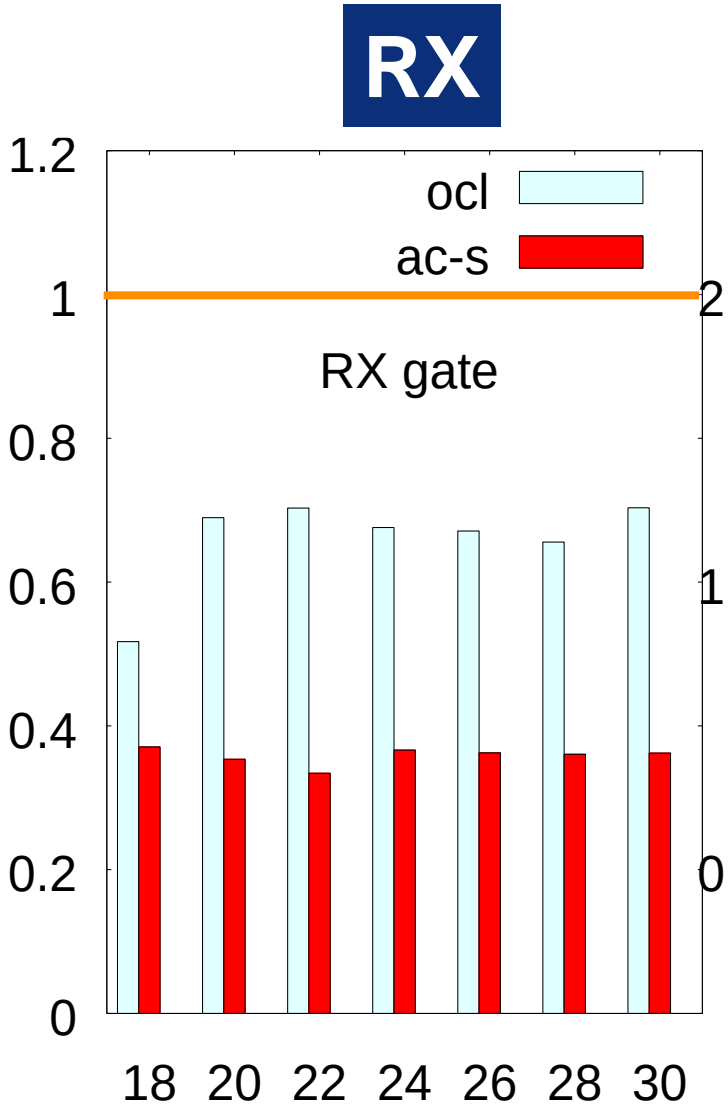
Exec. and relative exec time of single qubit w/ rotation gates

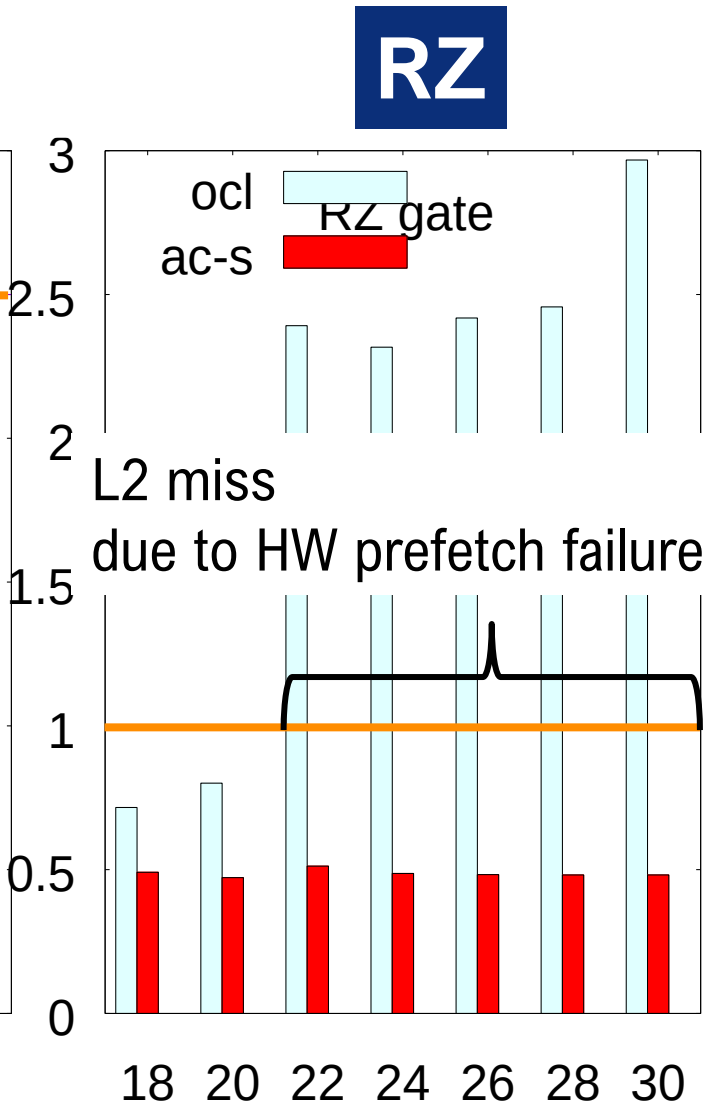
※ ACLE, general has been omitted



- Rotation gates performs floating point operations in the product of complex valued small matrix for the rotation of a specific angle, and the quantum state vector, vectorization can reduce the execution time







L2 miss
due to HW prefetch failure

L1D access wait time

- One of the reasons of a long L1D access wait time is instruction scheduling
  - hide the 9+ cycle latency to move data from L1D to register
- Investigate different compile options
  - striping=2
    - applies loop striping
    - (may) enhance software pipelining

```
for(i=0; i<N; i++){
  a[i] = b[i] + c[i];
}
```

```
for(i=0; i<N; i+=2){
  tmp_b0 = b[i];
  tmp_b1 = b[i+1];
  tmp_c0 = c[i];
  tmp_c1 = c[i+1];
  tmp_a0 = tmp_b0 + tmp_c0;
  tmp_a1 = tmp_b1 + tmp_c1;
  a[i]   = tmp_a0;
  a[i+1] = tmp_a1;
}
```

  - swp_policy=large (default: auto)
    - a software pipelining algorithm for large loops
  - baseline OCL code

-Kocl	-Kocl,striping=2
-Kocl,swp_policy=large	-Kocl,swp_policy=large,strping=2

L1D access wait time and compile options

- relative execution time (baseline: -Kocl)
- 24qubit, 1 thread

- loop striping

improve the performance

for 6 of 9 gates

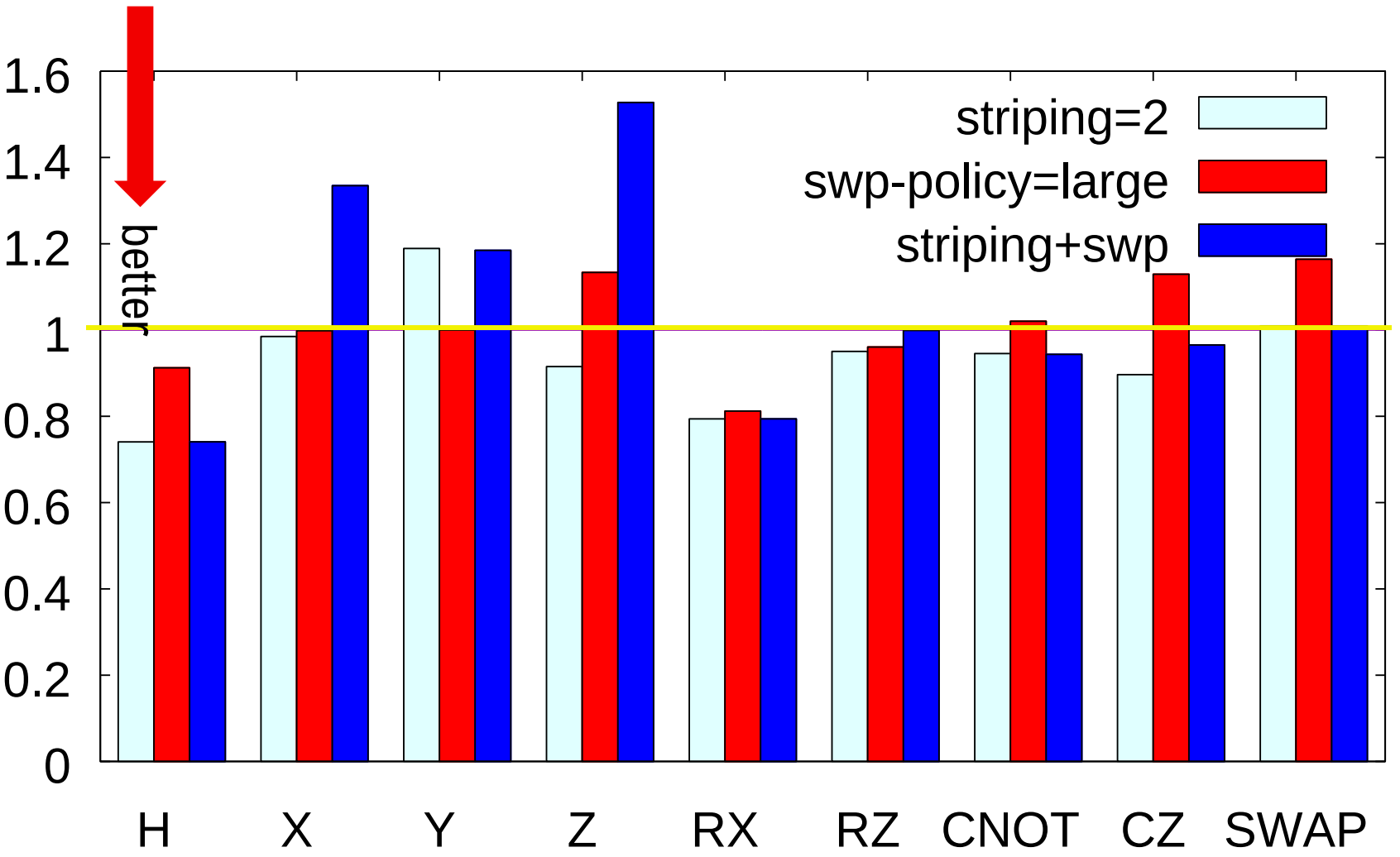
- changing swp_policy

is not effective.

Note that the default policy

(auto) may sometimes choose

“large”



Summary

- Gate operations in QC simulators generally
  - scan a large complex array
  - involve gather loads and scatter stores (indirect accesses)
    - some assumptions avoid the indirect accesses, but it is not always the case….
- Vectorization helps to increase the performance of the gate options which use arithmetic instructions, otherwise, it shall decrease the performance
- For complex numbers, Fujitsu compilers sometimes generate ineffective instructions. Using ACLE helps to reduce the redundant operations
- Successful vectorization induces a long waiting time for L1
- Instruction scheduling should be important to hide the latency of L1
  - loop-striping works well for 6/9 examples
- Future works
  - Investigate the effect of HW/SW prefetches, different software pipelining options