

SBCL for Quantum Computing at Rigetti

Robert Smith

SBCL 20th Anniversary

10 December 2019

Me

- SBCL user since approx 2010, relative newbie
- Made one patch for ISQRT (which had bugs)
- **stylewarning** on IRC
- Sometimes noisy and annoying in #sbc1
 - I try to provide feedback, usage experience, perceived bugs, etc
 - Big supporter of porting SBCL to other platforms, but often my only contribution is HW
- Aspire to one day be in the big leagues of nikodemus, dougk, stassats, pkhuong, Xof, Alastair, etc.
- Manage the group at Rigetti responsible for quantum programming tools

Previous discussions

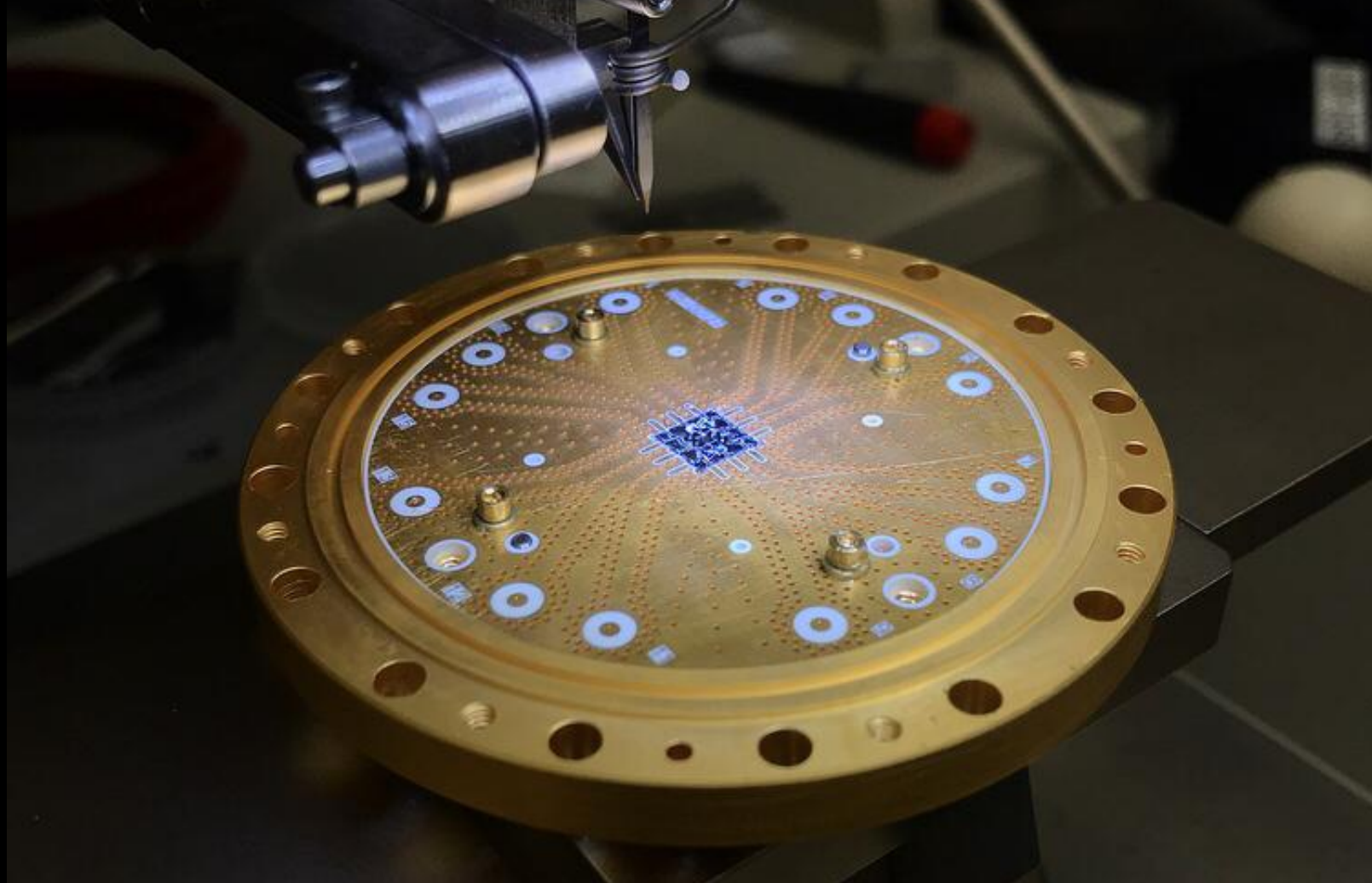
- “Lisp at the Frontier of Computation”, Bay Area Lisp Meetup (on YouTube)
- “Forest: An Open Source Quantum Software Development Kit”, FOSDEM 19 (on Youtube)
- “A Quantum Interpreter in 175 Lines of Lisp”, ELS 2018 in Marbella, Spain
- “An Optimizing Compiler for Near-Term Quantum Computers”, ELS 2020
 - ??? TBD TBD TBD

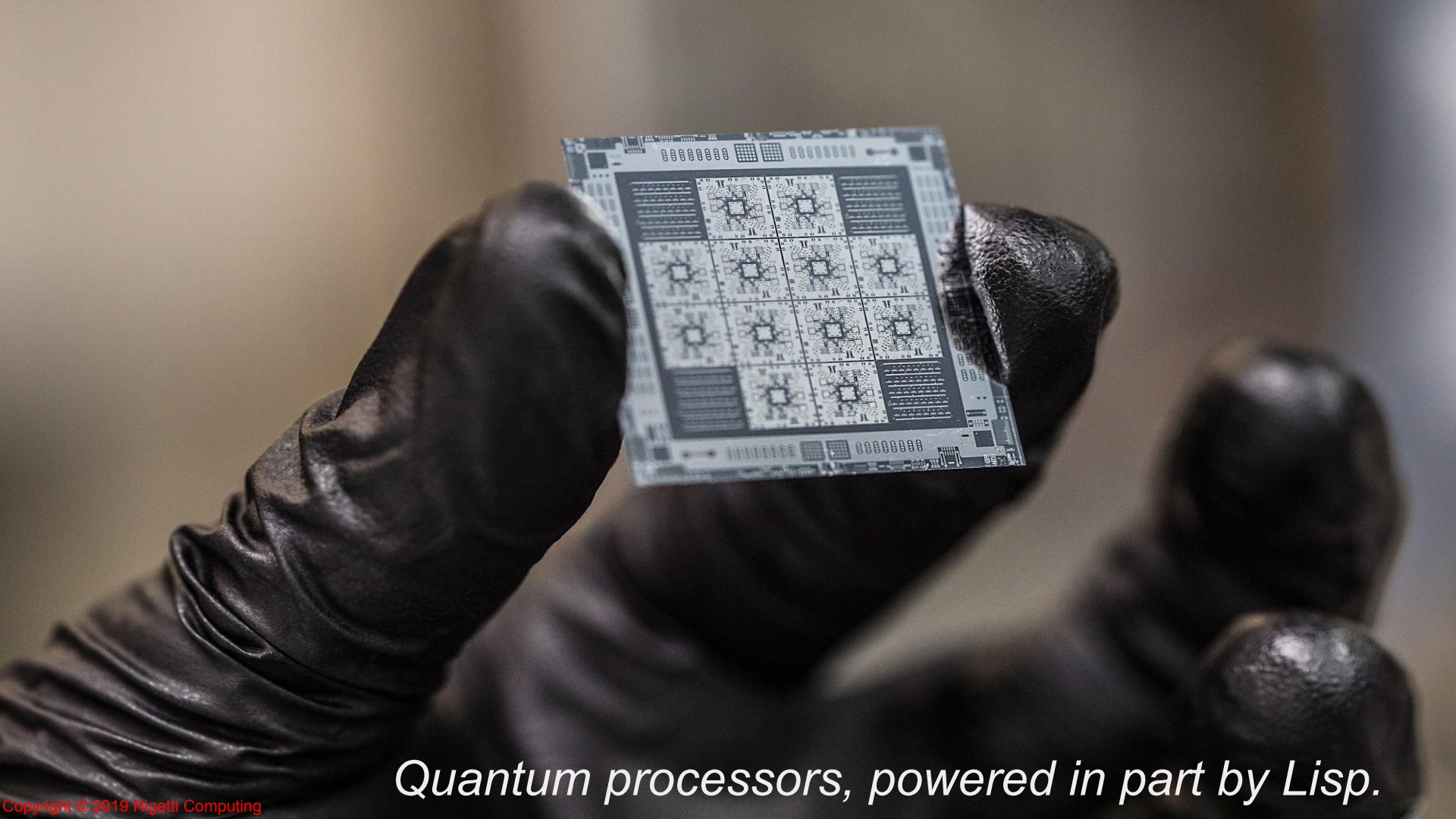
I'll try to talk about stuff I wouldn't ordinarily talk about in those venues.

Rigetti Computing

- >100 employees across the world
- Build and deploy Real™ quantum computers
- HQ & Quantum Data Center is in Berkeley, CA
- Full-service quantum integrated circuit fab in Fremont, CA
- Main product: Quantum Cloud Services (QCS)
 - Highest performing quantum computing service
- Recently announced: Integration into AWS with Amazon Braket







Quantum processors, powered in part by Lisp.

What you need to know about quantum computers

Quantum computers are just computers

- They are real
- They can be programmed
- They have a running “state”
- They run by processing instructions which affect that state

Only twist: What is the state? It's not bits in RAM or processor registers.

It's a quantum state. And unfortunately it's non-trivial to describe.

But it's still a state that is manipulated by instructions.

What is the tensor product?

$$V := \text{span}\{\hat{x}, \hat{y}, \hat{z}\}$$

vectors of the form $\alpha\hat{x} + \beta\hat{y} + \gamma\hat{z}$

$$W := \text{span}\{\hat{p}, \hat{q}\}$$

vectors of the form $\alpha\hat{p} + \beta\hat{q}$

A **tensor product** is one way to combine vector spaces.

What is the tensor product?

$V := \text{span}\{\hat{x}, \hat{y}, \hat{z}\}$ vectors of the form $\alpha\hat{x} + \beta\hat{y} + \gamma\hat{z}$

$W := \text{span}\{\hat{p}, \hat{q}\}$ vectors of the form $\alpha\hat{p} + \beta\hat{q}$

"V tensor W"

$$\begin{array}{c} \downarrow \\ V \otimes W \end{array} := \text{span} \left\{ \begin{array}{ccc} \hat{x} \otimes \hat{p}, & \hat{y} \otimes \hat{p}, & \hat{z} \otimes \hat{p}, \\ \hat{x} \otimes \hat{q}, & \hat{y} \otimes \hat{q}, & \hat{z} \otimes \hat{q} \end{array} \right\}$$

vectors of the form $\alpha(\hat{x} \otimes \hat{p}) + \beta(\hat{y} \otimes \hat{p}) + \cdots + \zeta(\hat{z} \otimes \hat{q})$

Modeling a single qubit

A qubit can be in a zero-state, or one-state, or some kind of **superposition**.

$$\alpha |0\rangle + \beta |1\rangle$$

“The zero-state”

“The one-state”

But the probability of being in either has to be 1.

$$|\alpha|^2 + |\beta|^2 = 1$$

The values α and β are called **probability amplitudes**. They're complex numbers!

Modeling two qubits

A pair of qubits can **interact**.

$$Q_a := \text{span}\{|0\rangle_a, |1\rangle_a\} \quad \text{State-space of qubit } a.$$

$$Q_b := \text{span}\{|0\rangle_b, |1\rangle_b\} \quad \text{State-space of qubit } b.$$

Their combined state-space is a **tensor product**.

$$\begin{aligned} Q_a \otimes Q_b &:= \text{span}\{|0\rangle_a \otimes |0\rangle_b, |0\rangle_a \otimes |1\rangle_b, |1\rangle_a \otimes |0\rangle_b, |1\rangle_a \otimes |1\rangle_b\} \\ &= \text{span}\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\} \end{aligned}$$

The state of **2 qubits** is a vector of **4 complex numbers**.

We simply label each of the possible **basis states** by a **bitstring**.

A general quantum state

We can add more qubits by adjoining more to the tensor product.

The state of **3 qubits** would need **8 components**.

The state of **4 qubits** would need **16 components**, and so on...

$$\psi = \sum_{i=0}^{2^n-1} \psi_i |\text{bitstring } i\rangle$$

The state ψ is sometimes called a **wavefunction**.

Each component indirectly represents a **probability** of observing that state.

$$|\psi_i|^2 = \text{probability of observing bitstring } i$$

Scaling of a simulation...

26 qubits → laptop

36 qubits → expensive server

46 qubits → supercomputer

260 qubits → universe-sized computer

Computing a GHZ state on 4 qubits

```
$ echo 'H 0; CNOT 0 1; CNOT 1 2; CNOT 2 3' | ./qvm
```

```
[... snip ...]
```

Amplitudes:

|0000>: 0.7071067811865475,

|0001>: 0.0,

|0010>: 0.0,

|0011>: 0.0,

|0100>: 0.0,

|0101>: 0.0,

|0110>: 0.0,

|0111>: 0.0,

|1000>: 0.0,

|1001>: 0.0,

|1010>: 0.0,

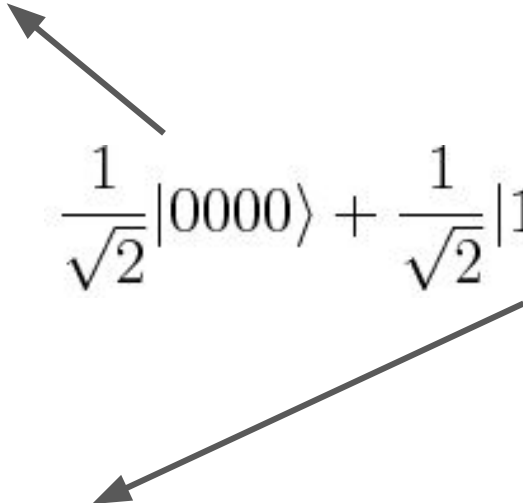
|1011>: 0.0,

|1100>: 0.0,

|1101>: 0.0,

|1110>: 0.0,

|1111>: 0.7071067811865475,


$$\frac{1}{\sqrt{2}}|0000\rangle + \frac{1}{\sqrt{2}}|1111\rangle$$

Program:

H 0

CNOT 0 1

CNOT 1 2

CNOT 2 3

P= 50.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 0.0%

P= 50.0%

Instructions cannot frob amplitudes directly! They can only address qubits!

Back to Lisp

Common Lisp @ Rigetti

github.com/rigetti

- Lisp since 2016
 - Only one Lisp project was rewritten, and it turned out to be too slow after rewriting
- Majority of SW engineers
 - For most of them, Rigetti is their first true exposure to Lisp
- About 50k lines of open source, mission-critical code
 - **qvm**: a quantum computer simulator
 - **quilc**: an optimizing compiler for quantum computers
 - Grab bag of other things: matrix algebra, lexer generators, RPC framework, etc
- Around 10 full-time Lisp programmers
 - Administered lots of project-based contracts; had about 10 Lisp-writing interns over the years
- Pretty exclusively SBCL, but we have dabbled with CCL, ECL, ACL, LW
- Mostly try to write ANSI conforming code

Simulator (qvm)

- One of the fastest simulators in the world
- Certainly the most flexible
 - Supports many kinds of simulation
 - Supports simulating noise
- Has a poor-man's quantum-to-asm JIT
 - Outperforms other simulators by >10%
- Runs on two flavors of computer architecture
 - Single address space, scales to as much RAM as you have
 - Distributed with Marco's CL-MPI, can scale as much as you want

Compiler (quilc)

- World's only quantum compiler that's:
 - Optimizing
 - Automatic
 - Completely open source
- Compiler has parametric backends
 - Crude comparison: Imagine being able to specify at runtime the no. registers, the available arithmetic instructions, memory layout, etc.
- Compiles “high-level” code called **Quil** into native quantum computer instructions

Quil [01]

Lots of different kinds of issues, from...

Implement efficient decomposition of orthogonal 3Q gates #113



ecpeterson opened this issue on Feb 9 · 0 comments



ecpeterson commented on Feb 9

Member



The paper arxiv.org/abs/1203.0722 provides an efficient method for decomposing members of $SU(8) \cap O(8)$. We should look at implementing this and at extending it slightly: can we recognize when a generic member of $SU(8)$ can be easily moved into this subgroup by left-/right-multiplication?

Assignees

No one—assign

Labels

enhancement

math

...to...

BUILD-GATE will accept single-float parameters, but PRINT-INSTRUCTION will raise a cryptic error if parameters are not DOUBLE-FLOAT. #434



notmgsk opened this issue on Sep 23 · 1 comment

Lisp's influence on Quil

Quil was originally an S-expression-based language.

Then the parens were stripped out.

But vestiges of Lispiness still exist:

- DEFGATE: define a new quantum operator
- DEFCIRCUIT: define a shorthand for a sequence of instructions
- Kebab-case variable names
 - Much to the chagrin of people wanting to write $x-y$ for subtraction.

Lisp's influence: DEFINE-COMPILER

What to translate

```
(define-compiler SWAP-to-CNOT ((swap-gate ("SWAP" () q0 q1)))
```

```
  (inst "CNOT" () q0 q1)
```

```
  (inst "CNOT" () q1 q0)
```

```
  (inst "CNOT" () q0 q1))
```

What instructions to expand to.

Lisp's influence: more DEFINE-COMPILER

```
(define-compiler elide-applications-on-eigenvectors
  ((instr :acting-on (psi qubit-indices)
    :where (collinearp psi (nondestructively-apply-instr-to-wf instr psi qubit-indices))))
  nil)
```

Translation to math speak:

If INSTR is an instruction acting on the qubits QUBIT-INDICES of quantum state PSI, and if PSI is an eigenvector of INSTR, then eliminate it.

SBCL @ Rigetti

- Relatively serious users of SBCL
- Heaps sometimes measured in terabytes
- Custom SIMD VOPs
- Lots of performance hacking
- Have deployed Lisp to:
 - Raspberry pi's
 - Regular old laptops
 - Beefy servers (2+ TB RAM)
 - Networked clusters
 - TBD: Supercomputers?
- At a point where we can probably justify hiring a full-time SBCL hacker.

```
;; matmul2-simd-real
;;
;; Multiplies a real-only 2x2 matrix by a 2x1 vector
;;
;; ARGS:
;;   m0r, m1r: Packed real vectors from 2x2matrix-to-simd
;;   a0, a1: Column vector
;;
;; RESULT:
;;   p, q: Components of 2x1 complex double matrix result
;;
(define-vop (qvm-intrinsics::matmul2-simd-real)
  (:translate qvm-intrinsics::%matmul2-simd-real)
  (:policy :fast-safe)
  (:args (m0r :scs (double-avx2-reg))
         (m1r :scs (double-avx2-reg))
         (a0 :scs (complex-double-reg) :target p)
         (a1 :scs (complex-double-reg) :target q))
  (:arg-types simd-pack-256-double
              simd-pack-256-double
              complex-double-float
              complex-double-float)
  (:results (p :scs (complex-double-reg))
            (q :scs (complex-double-reg)))
  (:result-types complex-double-float complex-double-float)
  (:temporary (:sc double-avx2-reg) aa0 aa1)
  (:generator 4
    (let ((acc aa0)) ; Save a register by using the first used temp to aa0
      (qvm-intrinsics::repeat-complex-registers
        (list aa0 a0) ; aa0 = [ a0i a0r a0i a0r ]
        (list aa1 a1)) ; aa1 = [ a1i a1r a1i a1r ]
      (inst vmulpd acc m0r aa0) ; acc = [ (m10r * a1r) (m10r * a1i) (m11r * a1r) (m11r * a1i) ]
      (inst vmadd231pd acc m1r aa1) ; acc += [ (m11r * a1r) (m11r * a1i) (m10r * a1r) (m10r * a1i) ]
      (inst vextractf128 p acc #xFF) ; p = [ acc[3] acc[2] ]
      (inst vextractf128 q acc #x00))) ; q = [ acc[1] acc[0] ]
```

Google recently announced **Quantum Supremacy**

- A quantum computer solved an **artificial, contrived, classically difficult mathematical problem** on the **order of minutes**.
 - The best supercomputer would take centuries
- It still doesn't prove quantum computers will be useful, but it's a good first step.
- In this regime, how do we verify that the quantum computer did something correctly?

How? We use supercomputers.

- First, we choose slightly smaller problems that won't take millenia to solve
- The only supercomputer able to check Google's boldest claim is **Summit**
- Over 9,000 22-core **POWER9** (ppc64el) chips
- Rigetti has been developing supercomputer simulators for almost 2 years
- Originally, we were going to use ECL...
- ...or ask a commercial CL vendor to port^(since it's practically impossible to pay SBCL devs)...
- ...until a bunch of SBCL dev magic happened in the past 6 months
 - (Shoutout to **dougk** & **karlosz** for finishing up the ppc64el job!)
- Still haven't run on Summit, but now it's on the table.

*I also like ppc64el because I have two
POWER8 servers racked up in my spare
bedroom.*

Getting SBCL to play nice(r) with customers

(work by Juan Bello-Rivas, Mark Skilbeck)

Customers want shared libraries

- Funny “servers”, RPC mechanisms, etc. were turn-offs to many of our most prodigious customers and users
- They want to use Python, C++, Julia, etc.
- They also sometimes want extreme customization, shared memory, and other stuff that doesn't fit the RPC model.
- Shared libraries with C headers are one solution
- ECL, ACL, and LW support shared libraries

“So just use ECL, that’s what it’s for!”

- ECL is too slow, hard to optimize, conses a lot
- We found it difficult to integrate with; “documentation” was `ec1.h`
- Seems great if you want to do what it says on the box: “embed a Common Lisp”
 - Seems less great for delivering large applications that mostly look like C
- Biggest hang-up: Doesn’t have SLAD!
 - Loading our simulator takes >2 minutes (!!) on a modern MacBook Pro
- Not “production ready” for our app....
- ... but Daniel K does an excellent job responding to requests, improving, etc.
- **So how do we integrate with C code?**

Hack libsbcl.so.

Interface between C and SBCL

A simple idea:

- Repurpose alien callbacks.
- Put `libsbc1.so` to work.

To simplify further, we take threading out of the picture.

What's needed: don't exit

- An interface to `src/runtime/runtime.c` that returns from Lisp (as opposed to exiting to the OS directly).

This requires modifying `save-lisp-and-die` in Lisp and letting `sbc1_main` in C return to its caller.

What's needed: setup signal handlers

- While switching back and forth between C and Lisp, we save and restore signal handlers.

The SBCL runtime handles signals occurring during Lisp code while the C side handles signals in C code (i.e., at the moment it is not possible to set up a signal handler in Lisp to catch a signal emitted in C).

What's needed: get stack in order

- Additionally, we hijack `funcall_alien_callback` in `src/runtime/x86-64-assem.S` to set up the stack so that C can call into Lisp.

Embedding the core

To provide a more self-contained solution, we embed the SBCL core into:

- Linux shared objects: we create a new ELF section where we store the core.

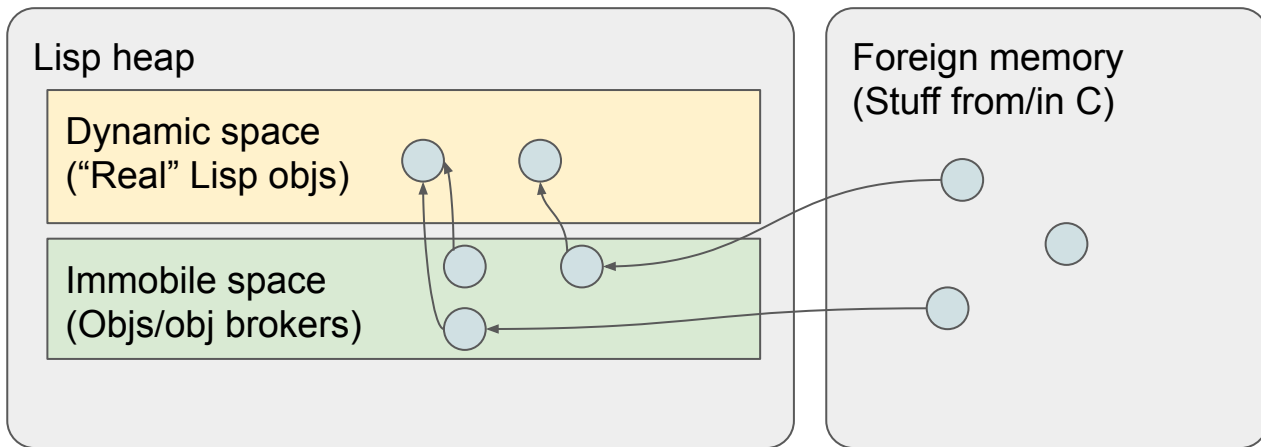
We extract the core into a temporary file right before invoking `sbc1_main` at runtime.

- macOS dynamic libraries: we append the core at the end of the file and use the Mach-O file format to figure out the offset where it is located when needed.
- We could have modified `sbc1_main` to read the core directly from the libraries instead of extracting it but so far we've been conservative in how many changes to SBCL we make.
 - SBCL developers are grumpy about wacky changes :)

Data structure access

We store key objects in immobile space to get persistent memory addresses as well as garbage collection.

This enables the possibility of creating direct C interfaces to Lisp data structures akin to the way the SBCL runtime accesses primitive objects stored in the static memory space.



“CSBCL” is still a work-in-progress

- Single-threaded
- Still lots of boilerplate
- Using `callback-heaven` to generate alien callbacks and manage patching in those function pointers somewhere C accessible
 - <https://github.com/stylewarning/callback-heaven>
- Hoping to get some of the good ideas into SBCL proper!
 - Would love first-class support for shared libraries

Closing comments on the value of Common Lisp for quantum computing.

(From the presentation) Common Lisp, the language, is a major boon to productivity, and we all know it. We rave about the greatness of macros, conditions, CLOS, etc., all of the things that make Common Lisp what it is. Though enormously valuable, it's not the power of the language that makes it especially suited for quantum computing, it's that implementations like SBCL—and certainly Common Lisp itself—provide truly first-class support for rapid development and idea-trying. Rapid development, in the way I'm describing it, is not the same as “rapid prototyping”; Common Lisp and SBCL have proven themselves time after time in building production-grade software. The innovative tools we've built at Rigetti could not have been possible without SBCL, and SBCL's stability has been a foundation for our software. Nobody yet knows how quantum computing will work 10 or 20 or 30 years from now, and innovation will depend on folks—like those at Rigetti—to be able to move fast with their ideas. In my opinion, Common Lisp environments are the only that truly permit that.

thanks; fin