C Code In, D-Wave QMI Out

D-Wave Qubits North America Users Conference 2019

Mohamed Hassan,
Scott Pakin, and
Wu-chun Feng

25 September 2019

Los Alamos NATIONAL LABORATORY EST. 1943

NNSA National Nuclear Security Administration

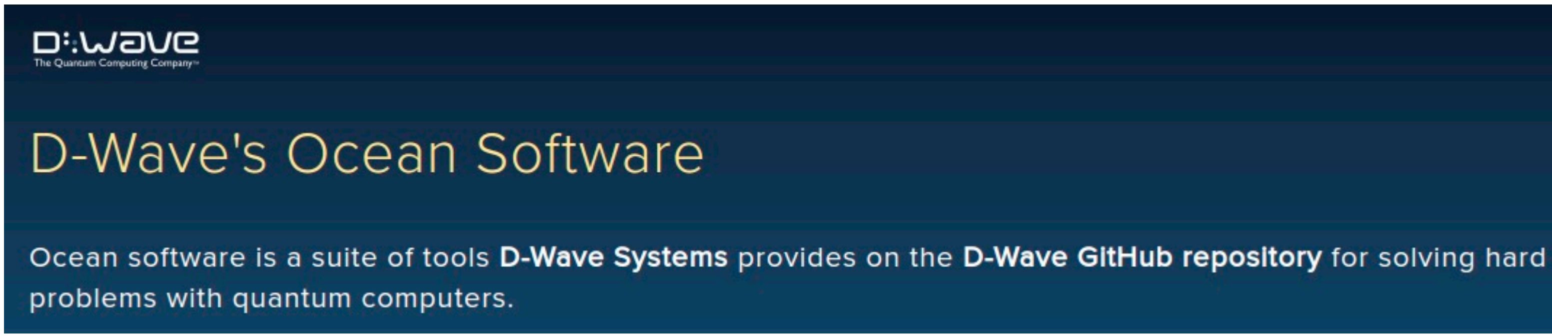
LA-UR-19-29334

Outline

- Goal and motivation
- Approach
- Example and results
- Conclusions

Programming a D-Wave System

- Vendor-supported SDK: Ocean



- Provides a variety of Python-based APIs for constructing BQMs and submitting these to a D-Wave quantum annealer for solution
  - BQM = binary quadratic model (a QUBO or Ising-model Hamiltonian)
- Let's write an Ocean program that adds two small integers and returns their sum…

Teaching a D-Wave to Add Two Numbers

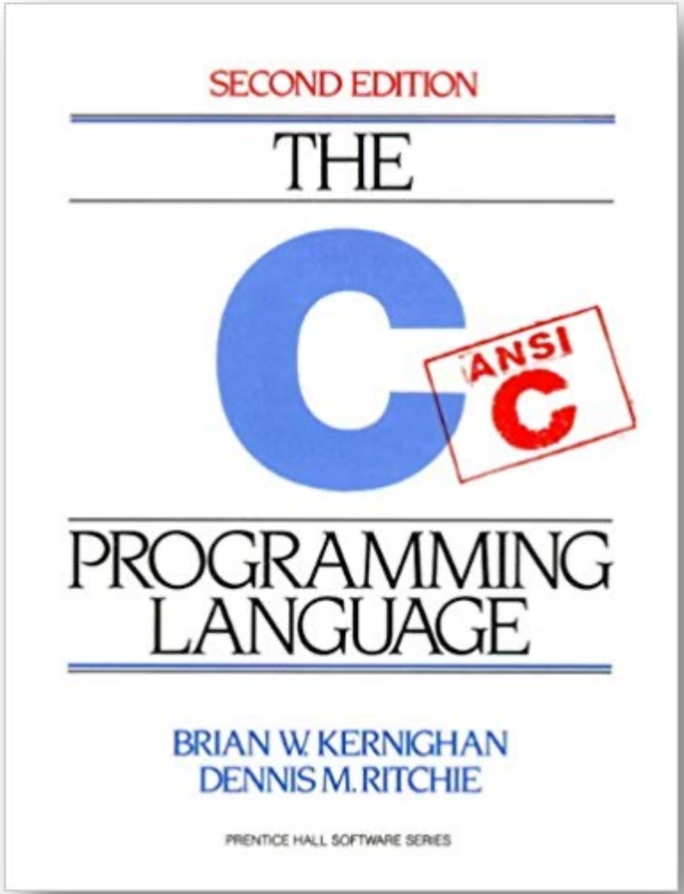
```
#! /usr/bin/env python

from dwave.system import DWaveSampler,
EmbeddingComposite
from dwave.cloud import Client

client = Client.from_config()
sampler =
EmbeddingComposite(DWaveSampler(solver=clie
nt.default_solver))

Q = {("in1[0]",     "in1[0]"):     -3.0000,
     ("in1[0]",     "result[0]"):  -2.0000,
     ("in1[0]",     "temp23"):      4.0000,
     ("in1[0]",     "temp4"):       4.0000,
     ("in1[1]",     "in1[1]"):     -5.0000,
     ("in1[1]",     "in2[1]"):      2.0000,
     ("in1[1]",     "temp14"):      4.0000,
     ("in1[1]",     "temp20"):     -2.0000,
     ("in1[1]",     "temp24"):      4.0000,
     ("in1[1]",     "temp25"):      4.0000,
     ("in1[2]",     "in1[2]"):      1.0000,
     ("in1[2]",     "in2[2]"):      2.0000,
     ("in1[2]",     "temp33"):      4.0000,
     ("in1[2]",     "temp39"):      2.0000,
     ("in1[2]",     "temp43"):     -4.0000,
     ("in1[2]",     "temp44"):     -4.0000,
     ("in1[3]",     "in1[3]"):     -5.0000,
     ("in1[3]",     "temp52"):      4.0000,
     ("in1[3]",     "temp58"):     -2.0000,
     ("in1[3]",     "temp62"):      4.0000,
     ("in1[4]",     "in1[4]"):     -1.0000,
     ("in1[4]",     "in2[4]"):     -2.0000,
     ("in1[4]",     "temp65"):      4.0000,
     ("in1[4]",     "temp71"):      2.0000,
```

… 100 lines deleted …

```
     ("temp71",     "temp65"):      4.0000,
     ("temp71",     "temp69"):     -4.0000,
     ("temp71",     "temp71"):      2.0000}

result = sampler.sample_qubo(Q,
num_reads=1000)
print(result)
```

Raising the Level of Abstraction

- This is not a natural way to express $x + y$
- *Goal*: Use a conventional, classical programming language to express BQMs
- In this work, we consider using C as our source programming language



Clarification: What the Goal is *Not*

- The goal is not to express the BQM's linear and quadratic coefficients as a C data structure instead of as a Python data structure:

```
typedef struct {
  char *q1;
  char *q2;
  double val;
} qubo_t

qubo_t Q[] =
  {{"in1[0]",     "in1[0]",      -3.0000},
   {"in1[0]",     "result[0]",   -2.0000},
   {"in1[0]",     "temp23",       4.0000},
   {"in1[0]",     "temp4",        4.0000},
   {"in1[1]",     "in1[1]",      -5.0000},
   {"in1[1]",     "in2[1]",       2.0000},
   {"in1[1]",     "temp14",       4.0000},
   {"in1[1]",     "temp20",      -2.0000},
   {"in1[1]",     "temp24",       4.0000},
   {"in1[1]",     "temp25",       4.0000},
   {"in1[2]",     "in1[2]",       1.0000},
   {"in1[2]",     "in2[2]",       2.0000},
   {"in1[2]",     "temp33",       4.0000},
   {"in1[2]",     "temp39",       2.0000},
   {"in1[2]",     "temp43",      -4.0000},
   {"in1[2]",     "temp44",      -4.0000},
   {"in1[3]",     "in1[3]",      -5.0000},
   {"in1[3]",     "temp52",       4.0000},
   {"in1[3]",     "temp58",      -2.0000},
   {"in1[3]",     "temp62",       4.0000},
   {"in1[4]",     "in1[4]",      -1.0000},
   {"in1[4]",     "in2[4]",      -2.0000},
   {"in1[4]",     "temp65",       4.0000},
   {"in1[4]",     "temp71",       2.0000},
   {"in2[0]",     "in2[0]",      -3.0000},
   {"in2[0]",     "result[0]",    2.0000},
   {"in2[0]",     "temp23",       4.0000},
   {"in2[0]",     "temp4",       -4.0000},
   {"in2[1]",     "in2[1]",      -5.0000},
                  … etc. …
```

What the Goal *Is*

- We want to be able to
  - Write a C function such as that shown to the right
  - Compile it to a quantum machine instruction (QMI)
  - Run the QMI on a D-Wave system
  - Report the results in terms of source-program variables and data types

```c
int adder(int in1, int in2)
{
  return in1 + in2;
}
```

Outline

- Goal and motivation
- Approach
- Example and results
- Conclusions

Reining in Expectations

- The work presented here is functional but very much a proof of concept
- Please don't expect to be able to recompile your million-line C program with **`-march=dwave`** and have it run on a quantum annealer
- Many (most) C features are not supported

Supported	Not supported
Basic **for**, **while**, and **if** constructs	Loops with variable-length trip counts
Small integers and Booleans	Characters, strings, or floating point
Fixed-length arrays (including multi-D)	**struct**s, variable-length arrays, pointers
Variable assignments, most operators	Recursion

➔In short, only the very simplest of C programs can be expressed

Challenges

- A quantum annealer has no mutable state
  - Can’t assign a value to variable *x* and later assign a different value to *x*
- A quantum annealer has no clock
  - Can’t perform one operation at time *t* then another operation at time *t* + 1
- A quantum annealer has no explicit inputs
  - All inputs must be encoded as problem coefficients

➔Must bridge a huge semantic gap to convert C code to a QMI

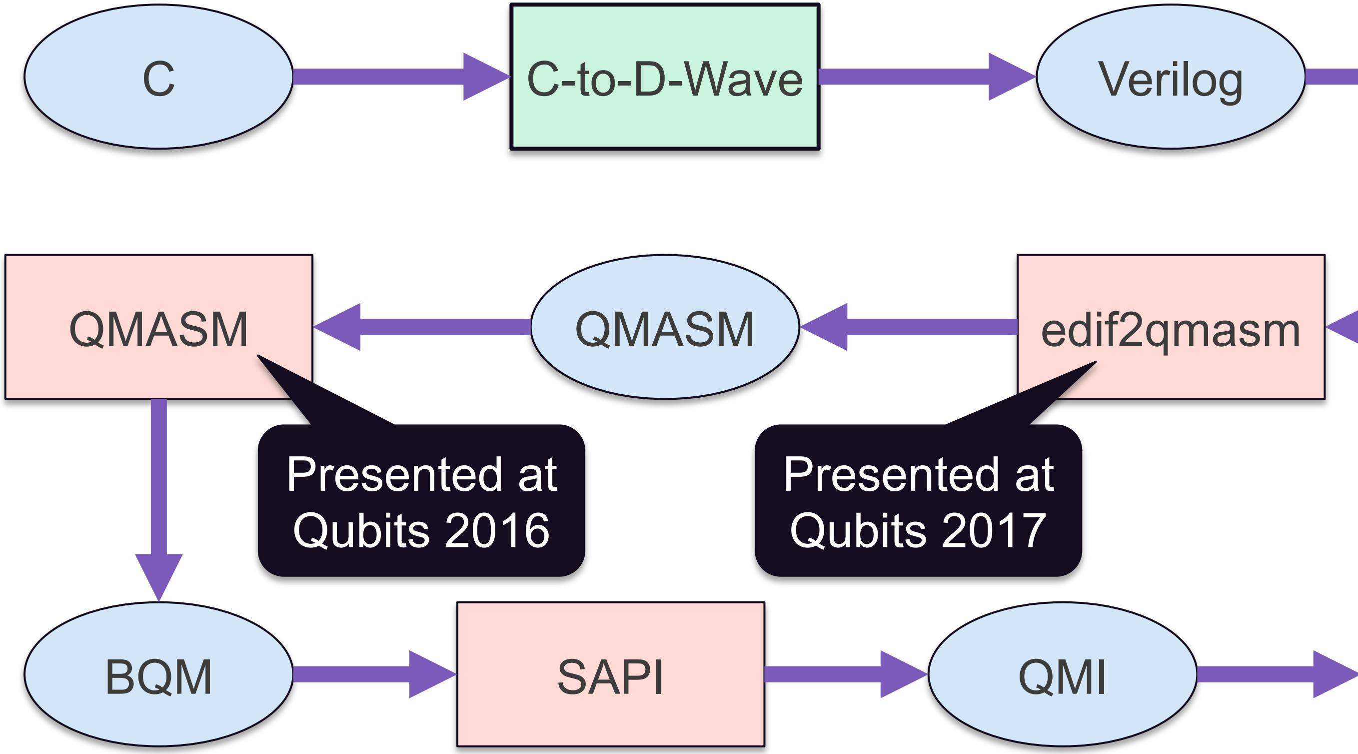
C

QMI

```
int adder(int in1, int in2)
{
  return in1 + in2;
}
```

$$\arg\min_{\sigma}\left(\sum_i h_i\sigma_i + \sum_i\sum_j J_{i,j}\sigma_i\sigma_j\right)$$

The C-to-D-Wave Software Stack



The C-to-D-Wave Approach

- Source-to-source translator (C → Verilog)
- Based on the Clang/LLVM compiler framework
- Walks the C abstract syntax tree (AST), converting each node in turn to Verilog
- Why Verilog?
  - Supports some high-level constructs (multi-bit values, conditionals, arithmetic/relational operators)
  - Compiles to a small set of simple primitives (AND, OR, NOT, etc.), suitable for mapping to BQMs

Preparing C Code for C-to-D-Wave

- C-to-D-Wave expects C code to be written in a slightly stylized form
- Function parameters are considered program inputs
  - That is, there is no `main()` function with `argc`/`argv` arguments
- The `return` statement defines the output
- `int` variables and constants are 5 bits wide
  - Attempts to strike a balance between usefulness and qubit consumption
  - Arbitrary; can be changed
- `bool` variables and constants are 1 bit wide
  - Reduces wasted qubits
- The `register` keyword indicates the need for a Verilog register
  - Loop induction variables
  - Variable reassignments (e.g., `temp` in “`temp = temp + val`”)

Is It Worth It?

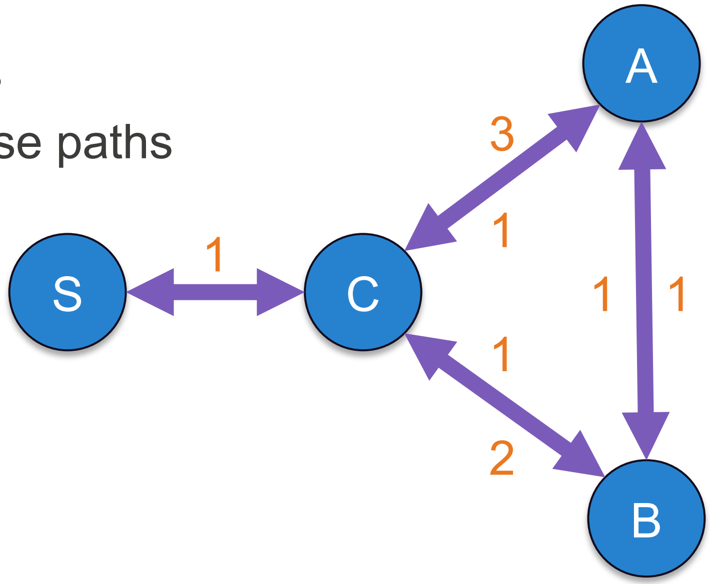
- For conventional code execution, no
  - A modern CPU can perform a *lot* of work in the time it takes to send a QMI to a D-Wave system and get back the results
- However,
  - The code generated by C-to-D-Wave is a *relation* of inputs and outputs, not a *function* from inputs to outputs
  - This means that we can not only supply inputs and receive outputs, but we can also supply outputs and receive the corresponding inputs
- This property simplifies the expression of challenging computational problems
  - Declarative approach: Describe *what* the solution looks like rather than *how* to produce the solution

Outline

- Goal and motivation
- Approach
- Example and results
- Conclusions

A Traveling-Salesman Problem

- Decision-problem variant
  - Given a weighted graph *G* and an integer *t*, is there a Hamiltonian path in *G* that costs at most *t*?
  - In answering the question, return the Hamiltonian path
- Example graph with weights
  - Inner weights are for clockwise paths
  - Outer weights are for counterclockwise paths



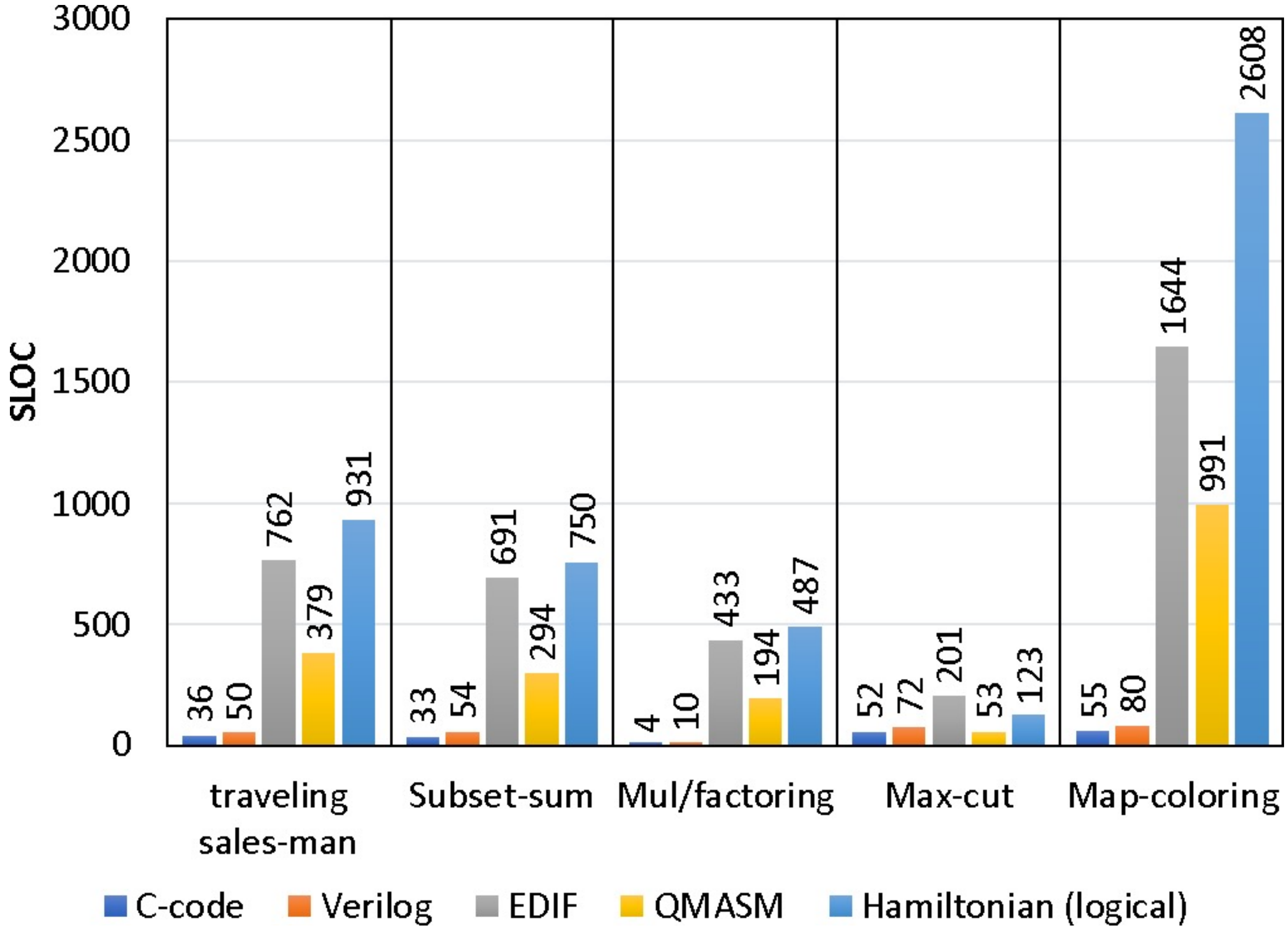
TSP Written in C

```c
bool TSP(int a, int b, int c, int tspdist) {
  bool valid;
  int arr_s[4];
  int arr_a[4];
  int arr_b[4];
  int arr_c[4];

  // starting city S costs
  arr_s[0] = 31;
  arr_s[1] = 31;
  arr_s[2] = 31;
  arr_s[3] = 1;

  // City A costs
  arr_a[0] = 31;
  arr_a[1] = 31;
  arr_a[2] = 1;
  arr_a[3] = 1;

  // City B costs
  arr_b[0] = 31;
  arr_b[1] = 1;
  arr_b[2] = 31;
  arr_b[3] = 2;

  // City C costs
  arr_c[0] = 1;
  arr_c[1] = 3;
  arr_c[2] = 1;
  arr_c[3] = 31;

  int totcost;
  totcost = arr_s[3] + arr_a[a] + arr_b[b] +
arr_c[c];

  if (totcost < tspdist && a > 0 && b > 0 &&
c > 0 && a < 4 && b < 4 && c < 4 &&
      a != b && a != c && c != b)
    valid = 1;
  else
    valid = 0;
  return valid;
}
```

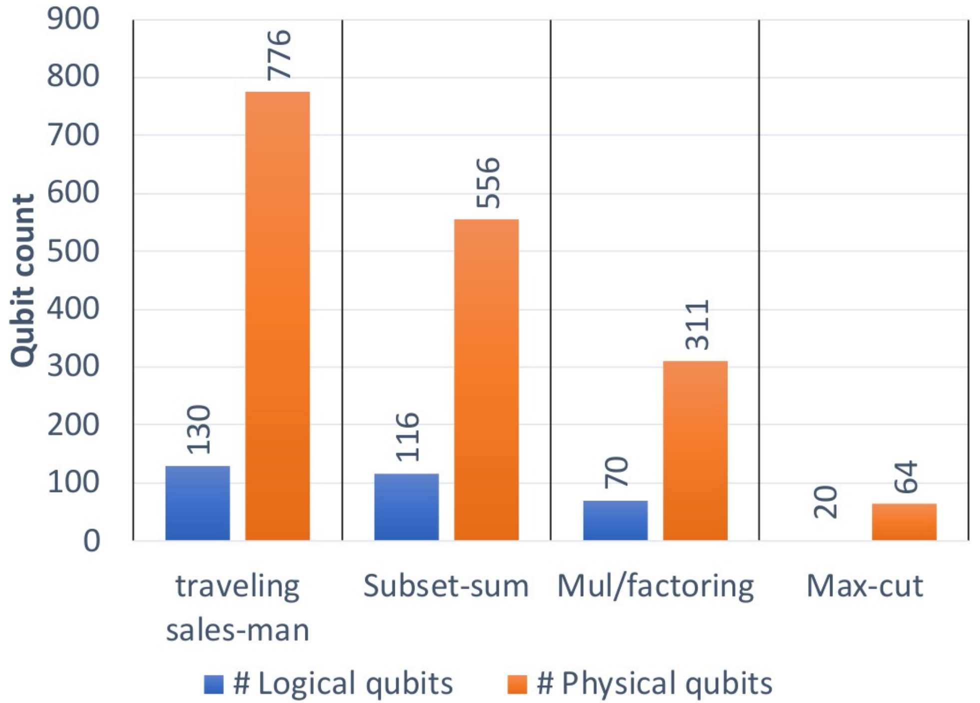
One Productivity Metric: Source Lines of Code (SLOC)

C (31) → C-to-D-Wave → Verilog (45) → Yosys → EDIF (684) → edif2qmasm → QMASM (328) → QMASM → BQM (449) → SAPI → QMI (1459) →

SLOC Counts for Other Test Cases



Another Metric: Qubit Count



- Even small bits of code consume a large fraction of a Chimera graph
- Looking forward to testing this against Pegasus to see how much larger these problems can scale

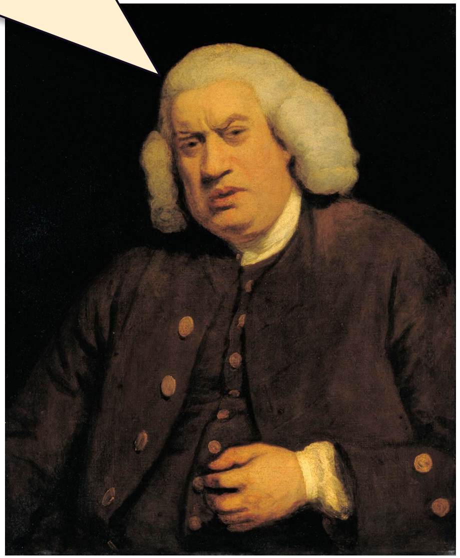
Outline

- Goal and motivation
- Approach
- Example and results
- Conclusions

Conclusions

Sir, [compiling C to a QMI] is like a dog's walking on his hind legs. It is not done well; but you are surprised to find it done at all.

- It is indeed possible to compile C code to a D-Wave QMI
  - Many limitations imposed due to the need to work around the large semantic gap
  - Technically, these could be bridged given a sufficient (→ extremely large) number of qubits
- Benefits of programming a D-Wave in C
  - Programmer-productivity gain versus manual construction of a QMI
  - Enables declarative solution to complex problems



Samuel Johnson, 1709–1784

For More Information…

- Mohamed W. Hassan, Scott Pakin, and Wu-chun Feng. “C to D-Wave: A High-level C Compilation Framework for Quantum Annealers”. In *Proceedings of the 23rd IEEE High Performance Extreme Computing Conference* (HPEC 2019). 24–26 September 2019, Waltham, Massachusetts, USA.
- https://github.com/lanl/c2dwave
  - BSD-3 Clear open-source license
  - Tested against Clang/LLVM 7.0
  - *Caveat*: Code is at best alpha quality and unlikely ever to be actively maintained