A Quantum Macro Assembler

Scott Pakin

28 September 2016
Outline

• Background and motivation
• Design and implementation
• Results and analysis
• Conclusions and future work
Motivating Example

- Circuit-satisfiability problem
- For what inputs (if any) is the output of a given circuit \textit{true}?
- Classic NP-complete problem—can’t beat exhaustive search in the general case (although usable heuristics do exist)
Refresher: Low-Level Programming Model

- Ising-model Hamiltonian with at most two-spin interactions:

$$\arg\min_{\sigma} E(\sigma) = \arg\min_{\sigma} \left( \sum_{i=1}^{N} h_i \sigma_i + \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} J_{i,j} \sigma_i \sigma_j \right)$$

for given $h \in \mathbb{R}^{N}$, $J \in \mathbb{R}^{N \times N}$ and solving for $\sigma \in \{-1, +1\}^{N}$
The Problem

• The preceding expression does not lead to a particularly user-friendly way to write programs:

| 0 0 -0.5 | 10 13 -0.1818 | 98 103 -0.6667 | 114 114 0.16665 | 206 214 -1 |
| 0 7 1 | 10 14 -0.1818 | 98 98 0.3333 | 114 118 0.3333 | 208 215 -1 |
| 0 96 -1 | 11 11 -0.13635 | 100 100 0.3333 | 114 119 -1 | 209 209 0.3636 |
| 2 6 -1 | 11 12 0.04545 | 100 108 -1 | 117 117 -0.25 | 209 212 -0.1818 |
| 2 98 -1 | 11 13 -0.0909 | 102 110 -1 | 118 118 0.16665 | 209 213 -0.1818 |
| 6 14 -1 | 11 14 -1 | 103 103 -0.33335 | 119 119 0.16665 | 209 214 -0.3636 |
| 7 15 -1 | 12 12 -0.13635 | 103 111 -1 | 192 192 0.16665 | 209 215 -0.1818 |
| 7 7 -0.5 | 13 13 0.1818 | 105 108 -1 | 192 196 0.16665 | 210 210 0.1818 |
| 8 12 -1 | 14 14 -0.13635 | 106 106 0.1818 | 192 197 -1 | 210 212 -0.0909 |
| 8 13 -0.0909 | 15 23 -1 | 106 108 -0.3636 | 192 198 -0.6667 | 210 213 -1 |
| 8 14 0.04545 | 16 16 -0.5 | 106 202 -1 | 194 194 0.16665 | 210 214 0.3636 |
| 8 15 -1 | 16 22 1 | 110 110 0.16665 | 194 196 -1 | 210 215 -0.1818 |
| 8 8 -0.13635 | 16 23 -1 | 110 118 -1 | 194 197 0.16665 | 211 211 -0.13635 |
| 9 105 0.3636 | 18 22 -1 | 111 119 -1 | 194 198 -0.6667 | 211 212 -1 |
| 9 12 -0.0909 | 22 22 -0.5 | 112 112 -0.6667 | 196 196 0.16665 | 211 213 -0.0909 |
| 9 13 -1 | 96 102 -1 | 112 118 -0.6667 | 196 204 -1 | 211 215 0.0909 |
| 9 14 -0.0909 | 96 192 -1 | 112 119 -0.6667 | 197 197 0.16665 | 212 212 -0.13635 |
| 9 9 0.1818 | 97 100 -0.6667 | 112 208 -1 | 198 198 -0.6667 | 213 213 0.1818 |
| 10 10 0.1818 | 97 103 -1 | 113 113 -0.5 | 198 206 -1 | 215 215 -0.2727 |
| 10 106 -1 | 97 97 -0.33335 | 113 117 0.25 | 202 204 -1 | |
| 10 12 -0.1818 | 98 100 0.3333 | 113 209 -1 | 204 212 -1 | |
Physical Representations Exposed Throughout

- **Must consider physical connectivity of the on-chip network**
  - Set of available couplers is very sparse: at most $6 J_{ij}$ for a given $i$
  - Any given installation will have a number of missing qubits (nodes) and couplers (edges) on the chip—differs from installation to installation
  - *Limitation*: No 3-cycles (or any odd cycles); must work around by converting to 4-cycles
  - Think manual place-and-route

- **Must consider physical range of coefficients**
  - Varies from installation to installation but typically around $[-2, +2]$ for point weights and $[-1, +1]$ for coupler strengths
  - **Results are given in terms of physical qubit numbers**
Goal

- **Abstract away physical system characteristics**
  - Still have to understand underlying D-Wave architecture and properties
- **Macros to support code reuse**
- **Analogy to classical assembly language**
  - On Intel 64, `imull` maps to `0000 1111 1010 1111` for `reg_1 \times reg_2` but to `0110 1011` for `reg_1 \times imm`
  - More convenient to write (and remember) `imull`, and no expressiveness is lost

➤ Implement a *quantum macro assembler*
  - Use as a building block for (future) higher-level programming languages
A First Look at QASM

• Let’s define a NOT operator in terms of a Hamiltonian:

\[
E(\sigma) = -\frac{1}{2}\sigma_1 + -\frac{1}{2}\sigma_2 + 1\sigma_1\sigma_2
\]

<table>
<thead>
<tr>
<th>(\sigma_1)</th>
<th>(\sigma_2)</th>
<th>(E(\sigma))</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>0</td>
</tr>
</tbody>
</table>

All the effort goes into computing appropriate \(h\) and \(J\) coefficients.

By design, \(\arg\min_\sigma E(\sigma)\) occurs exactly where \(\sigma_2 = \text{NOT} \, \sigma_1\).

• Here’s the corresponding (and complete) QASM program for NOT:

```
s1 -0.5
s2 -0.5
s1 s2 1.0
```

s1 and s2 are arbitrary variable names. (I was going to use “bele” and “lokai” but figured that reference is too obscure.)
Running a QASM Program

Compiles and runs the program
Logical-to-physical mapping

Solutions reported in terms of the symbolic names used in the source code

```
$ qasm --run not.qasm
# s1 --> 774
# s2 --> 782
Solution #1 (energy = -1.00):

<table>
<thead>
<tr>
<th>Name(s)</th>
<th>Spin</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>-1</td>
<td>False</td>
</tr>
<tr>
<td>s2</td>
<td>+1</td>
<td>True</td>
</tr>
</tbody>
</table>

Solution #2 (energy = -1.00):

<table>
<thead>
<tr>
<th>Name(s)</th>
<th>Spin</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>+1</td>
<td>True</td>
</tr>
<tr>
<td>s2</td>
<td>-1</td>
<td>False</td>
</tr>
</tbody>
</table>
```
Taking Advantage of Macros

• Our sample circuit contains two NOT gates
• Let’s wrap up our definition in a macro to facilitate reuse:

```
s1 -0.5
s2 -0.5
s1 s2 1.0
```

# NOT (Y = not A)

```
!begin_macro not1
$A -0.5
$Y -0.5
$A $Y 1.0
!end_macro not1
```

• Any variable name containing “$” is considered “internal” or “uninteresting” and not output by default
  • `--verbose --verbose` will show it, though
• In our example, we don’t care about individual operators, just the overall circuit inputs and outputs
A Circuit Library

- For our sample circuit, we need a 3-input AND and a 2-input OR in addition to the NOT we just defined

```
# 3-input AND (Y = A and B and C)
!begin_macro and3
$A  0.2727
$B  0.0000
$C  -0.2727
$Y  0.3636
$a1 0.3636

$A $B   0.0000
$A $C   0.0909
$A $Y    -0.1818
$A $a1   -0.1818

$B $Y   -0.3636
$B $a1   0.3636

$C $Y   -0.1818
$C $a1   -0.1818

$Y $a1   -0.1818

!end_macro and3
```

```
# 2-input OR (Y = A or B)
!begin_macro or2
$A  0.3333
$B  0.3333
$Y  0.6667

$A $B  0.3333
$A $Y  0.6667
$B $Y  0.6667

!end_macro or2
```
# Solve a circuit-satisfiability problem

```plaintext
!use_macro not1 not_x4
not_x4.$A = x3
not_x4.$Y = $x4

!use_macro or2 or_x5
or_x5.$A = x1
or_x5.$B = x2
or_x5.$Y = $x5

!use_macro and3 and_x7
and_x7.$A = x1
and_x7.$B = x2
and_x7.$C = $x4
and_x7.$Y = $x7

!use_macro or2 or_x8
or_x8.$A = $x5
or_x8.$B = $x6
or_x8.$Y = $x8

!use_macro or2 or_x9
or_x9.$A = $x6
or_x9.$B = $x7
or_x9.$Y = $x9

!use_macro and3 and_x10
and_x10.$A = $x8
and_x10.$B = $x9
and_x10.$C = $x7
and_x10.$Y = x10
```

- We use “=” to chain (equate) two variables
- Implementation: Choose $J_{ij}<0$ to favor $\sigma_i = \sigma_j$
Aside: Why Not Use ToQ?

- Higher-level programming model
  - Constraint satisfaction
- Pros
  - Rich library of built-in functions (including Booleans)
  - User doesn’t need to convert truth tables to Hamiltonians
- Cons
  - Substantial fraction of computation is performed classically in pre-processing step

```python
bool: @x1, @x2, @x3
bool: @x4, @x5, @x6
bool: @x7, @x8, @x9

assert: @x4 == Not(@x3)
assert: @x5 == Or(@x1, @x2)
assert: @x6 == Not(@x4)
assert: @x7 == And(And(@x1, @x2), @x4)
assert: @x8 == Or(@x5, @x6)
assert: @x9 == Or(@x6, @x7)
assert: And(And(@x8, @x9), @x7) == 1
end:
```
Compilation

• QASM uses D-Wave’s heuristic embedder to map program variables to physical qubits
  • Some variables are mapped to multiple qubits
• Output \((x_{10})\) is pinned to \textit{true} on the command line
  • Technique: Use truth table in which both \(T=false \rightarrow x_{10}=true\)
    and \(T=true \rightarrow x_{10}=true\)

<table>
<thead>
<tr>
<th>Logical feature</th>
<th>Tally</th>
<th>Physical feature</th>
<th>Tally</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>32</td>
<td>Qubits</td>
<td>53</td>
</tr>
<tr>
<td>Strengths</td>
<td>31</td>
<td>Couplers</td>
<td>71</td>
</tr>
<tr>
<td>Equivalences</td>
<td>21</td>
<td>Chains</td>
<td>23</td>
</tr>
</tbody>
</table>
Execution

```
$ qasm --pin="x10 := true" -O --run circsat.qasm
# x1 --> 204
# x10 --> 1
# x2 --> 194 197
# x3 --> 113
Solution #1 (energy = -39.00):

<table>
<thead>
<tr>
<th>Name(s)</th>
<th>Spin</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>+1</td>
<td>True</td>
</tr>
<tr>
<td>x10</td>
<td>+1</td>
<td>True</td>
</tr>
<tr>
<td>x2</td>
<td>+1</td>
<td>True</td>
</tr>
<tr>
<td>x3</td>
<td>-1</td>
<td>False</td>
</tr>
</tbody>
</table>
```
Performance

- **Measure both compilation time and execution time**
  - No clean distinction between program and inputs
- **Annealing time (time per solution) is a D-Wave 2X input**
  - More samples per unit time vs. increased likelihood of finding ground state
  - For each of 100 runs, we took 1000 samples @ 20µs/sample

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Base compilation</td>
<td>0.98</td>
</tr>
<tr>
<td>B</td>
<td>Optimization</td>
<td>1.01</td>
</tr>
<tr>
<td>C</td>
<td>Device pre- and post-processing</td>
<td>0.27</td>
</tr>
<tr>
<td>D</td>
<td>Annealing</td>
<td>0.02</td>
</tr>
<tr>
<td>E</td>
<td>Other (local prep. + HTTPS round trip)</td>
<td>1.47</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>3.75</strong></td>
</tr>
</tbody>
</table>

**Compilation**

**Execution**
Solution Quality

- The D-Wave 2X is a stochastic device
  - Can get different answers from one annealing cycle to the next
  - Can even get *wrong* answers
- QASM can filter out obviously incorrect results
  - Broken chains (where $x$ should be equal to $y$ but isn’t)
  - Broken pins (where $x$ should be *true* but is in fact *false* (or vice versa)
  - Non-ground state solutions (not $\arg \min_\sigma E(\sigma)$)
- QASM also ignores differences in “uninteresting” variables

<table>
<thead>
<tr>
<th>Solution type</th>
<th>Tally</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>1000</td>
</tr>
<tr>
<td>Distinct</td>
<td>$169.2 \pm 21.1$</td>
</tr>
<tr>
<td>Valid</td>
<td>$89.6 \pm 8.6$</td>
</tr>
<tr>
<td>Ground state</td>
<td>$2.0 \pm 0.1$</td>
</tr>
<tr>
<td>Interesting</td>
<td>$1.0 \pm 0.0$</td>
</tr>
</tbody>
</table>

(100 runs, 1000 samples/run)
Issues

• **Programmability**
  • Need fast algorithm for representing an arbitrary truth table as the ground state of a Hamiltonian

• **Performance**
  • Ought to cache embeddings locally and cache device state near the device to accelerate running same program with different inputs
  • Need faster embedder (possibly special-purpose); don’t want to have to classically solve an NP-complete embedding problem before executing an NP-complete program on the D-Wave

• **Solution quality**
  • The D-Wave 2X seems sensitive to physical placement of qubits
  • Same basic circuit laid out differently on the Chimera graph can observe a radically different fraction of valid solutions
Conclusions

- **QASM provides an improvement in program expressibility and ease of use for quantum (really, any) annealers**
  - ...in the sense that assembly language is easier to use than machine language
- **Low-level, but reduces the number of mundane details to worry about**
  - Embedding Hamiltonians in the D-Wave 2X’s physical topology
  - Reasoning about the physical location(s) of each variable

- **Pinning Boolean values to variables**
- **Reusing code blocks**
- **Interpreting results**
- **QASM can be a useful building block for higher-level programming models**
  - TBD what these might look like
  - *Current thinking*: sparsely connected islands of pre-computed logic blocks (not necessarily Boolean operators)

https://github.com/losalamos/qasm

(BSD-licensed but depends on D-Wave’s proprietary APIs)