ScaffCC: Scaffold Compiler Collection

User Manual

Ali JavadiAbhari, Adam Holmes, Shruti Patil, Jeff Heckey, Daniel Kudrow,
Pranav Gokhale, David Noursi, Lee Ehudin

June 2016

Abstract

ScaffCC [2] is a compiler and scheduler for the Scaffold programing language [**?**]. It is written using the LLVM [**?**] open-source infrastructure. It is for the purpose of writing and analyzing code for quantum computing applications.

ScaffCC enables researchers to compile quantum applications written in Scaffold to a low-level quantum assembly format (QASM), apply error correction, and generate time and area metrics. It is written to be scalable up to problem sizes in which quantum algorithms outperform classical ones, and as such provide valuable insight into the overheads involved and possible optimizations for a realistic implementation on a future device technology.

If you use ScaffCC in your publications, please cite this work as follows:

Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic Chong and Margaret Martonosi, *"ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs,"* ACM International Conference on Computing Frontiers (CF 2014), Cagliari, Italy, May 2014

Contents

1 Installation **3**
- 1.1 Getting ScaffCC . . . 3
- 1.2 Building ScaffCC . . . 3

2 Using ScaffCC **5**
- 2.1 Running the Compiler . . . 5
  - 2.1.1 Basic Example: . . . 5
- 2.2 Compiler Options . . . 5
- 2.3 Sample Scripts . . . 6
  - 2.3.1 Generating LLVM Intermediate Format: ./gen-ll.sh . . . 6
  - 2.3.2 Critical Path Estimation: ./gen-cp.sh . . . 6
  - 2.3.3 Module Call Frequency Estimation: ./gen-freq-estimate.sh . . . 6
  - 2.3.4 Generate Longest-Path-First-Schedule (LPFS): ./gen-lpfs.sh . . . 6

3 Built-in Quantum Applications **8**

4 RKQC:RevKit For Quantum Computation **9**
- 4.1 Installation . . . 9
  - 4.1.1 Obtain . . . 9
  - 4.1.2 Build and Compile . . . 9
- 4.2 First Example . . . 10
- 4.3 Declaration of Circuits . . . 10
- 4.4 Signal Types . . . 11
- 4.5 Creating and Using Modules . . . 11
- 4.6 Arrays . . . 12
- 4.7 Signals and Gates . . . 12
- 4.8 Assign Value . . . 13
- 4.9 Integer Arithmetic Modules . . . 14

5 Expanding ScaffCC **15**

Chapter 1

Installation

1.1 Getting ScaffCC

1. Go to https://github.com/epiqc/ScaffCC
2. Download the repository:

```
git clone https://github.com/epiqc/ScaffCC.git [dir]
```

1.2 Building ScaffCC

First you need to install the following dependencies. For each one, you can either install by building from source, or use the package manager of your system ("yum" on Red Hat or "apt-get" on Ubuntu).

1. Static libraries for libstdc++ and glibc
   - "Ubuntu"
     Install GNU gold linker
     You can check if you have this now by doing 'ld -v' and if it says 'GNU gold' you have it

     ```
     sudo apt-get install binutils-gold
     ```

   - "Red Hat"

     ```
     sudo yum install libstdc++-static
     sudo yum install glibc-static
     ```

2. GCC 4.5 or higher NOTE: if you need to preserve an older build, consider using 'update-alternatives' as system-wide method for preserving and maintaining these.
3. Boost 1.48
   - "Source Build"
     Boost installation instructions are here:
     `http://www.boost.org/doc/libs/1_48_0/doc/html/bbv2/installation.html`

```
wget
   http://sourceforge.net/projects/boost/files/boost/1.48.0/boost_1_48_0.tar.gz
tar zxf boost_1_48_0.tar.gz && cd boost_1_48_0/
sudo ./bootstrap.sh
sudo ./b2 install --prefix=/usr/local
```

4. The GNU Multiple Precision Arithmetic Library (gmp and gmpxx)

   - "Ubuntu" Use tab-completion to verify the correct packages (e.g. libgmp¡tab¿¡tab¿)

     ```
     sudo apt-get install libgmp-dev libgmpxx4ldbl
     ```

   - "Source Build"

     ```
     wget https://ftp.gnu.org/gnu/gmp/gmp-6.0.0a.tar.bz2
     sudo ./configure --disable-shared --enable-static --enable-cxx
     sudo make && sudo make check && sudo make install
     ```

5. The GNU MPFR Library (mpfr)

   - "Ubuntu"

     ```
     sudo apt-get install libmpfr-dev
     ```

   - "Source Build"

     ```
     wget http://www.mpfr.org/mpfr-current/mpfr-3.1.4.tar.bz2
     sudo ./configure --disable-shared --enable-static
     sudo make && sudo make check && sudo make install
     ```

6. Python 2.7 (or later)

Once you have all of the required libraries, simply run

```
make
```

or

```
make USE_GCC=1
```

at the root of the repository. The *USE_GCC* flag will force the Makefile to use GCC to compile instead, and this has been seen to be faster on some systems.

Chapter 2

Using ScaffCC

2.1 Running the Compiler

To run the compiler, simply use the 'scaffold.sh' script in the main directory, with the name of the program and optional compiler flags.

2.1.1 Basic Example:

The command below runs the compiler with default options on the Binary Welded Tree algorithm, with n=100 and s=100 as problem sizes. The default compiler option is to generate resource estimations (number of qubits and gates).

```
./scaffold.sh Algorithms/Binary Welded Tree/Binary_Welded_Tree_n100s100.scaffold
```

2.2 Compiler Options

To see a list of compiler options which can be passed as flags, run:

```
./scaffold.sh -h
```

```
Usage: ./scaffold.sh [-h] [-rqfRFcpd] [-L #] <filename>.scaffold
    -r   Generate resource estimate (default)
    -q   Generate QASM
    -f   Generate flattened QASM
    -R   Disable rotation decomposition
    -T   Disable Toffoli decomposition
    -l   Levels of recursion to run (default=1)
    -F   Force running all steps
    -c   Clean all files (no other actions)
    -p   Purge all intermediate files (preserves specified output,
         but requires recompilation for any new output)
    -d   Dry-run; show all commands to be run, but do not execute
```

2.3 Sample Scripts

This section describes some of the example scripts contained in the 'scripts/' directory. They are written to test the various functionalities of ScaffCC, as detailed below.

Each of them automates the process of running multiple compiler passes on an input file to perform the required analysis or optimization.

2.3.1 Generating LLVM Intermediate Format: ./gen-ll.sh

Lowers .scaffold source file to .ll file (intermediary LLVM format). Creates ¡algorithm¿.ll The .ll file is the main file in LLVM on which all transformations, optimizations and analysis are performed.

2.3.2 Critical Path Estimation: ./gen-cp.sh

Finds critical path information for several different flattening thresholds by doing the following:

1. Finding module sizes using the ResourceCount2 pass.
2. Flattening modules based on the found module sizes and the requested threshold.
3. Finds length of critical path, in terms of number of operations on it. Look for the number in front of "main" in the output.

flattening_thresh.py

Divides modules into different buckets based on their size, to be used for flattening decision purposes.

2.3.3 Module Call Frequency Estimation: ./gen-freq-estimate.sh

Generates an estimate of how many times each module is called, which can guide flattening decisions.

2.3.4 Generate Longest-Path-First-Schedule (LPFS): ./gen-lpfs.sh

Generates LPFS schedules with different options as specified below.

Options in the script: K=number of SIMD regions / D=capacity of each region / th=flattening thresholds

Calls the following scripts:

./regress.sh

Runs the 3 different communication-aware schedulers, LPFS, RCP, SS, with different scheduler configurations. Look in ./sched.pl for configuration options. For example using -m gives metrics only, while -s outputs entire schedule.

./sched.pl

The main scheduler code for LPFS and RCP.

./comm_aware.pl

Applies the communication penalty to timesteps.

All output files are placed in a new directory to avoid cluttering.

Chapter 3

Built-in Quantum Applications

This section describes the apps provided with this software, in the 'Algorithms/' directory.

1. Cat-State Preparation: Prepares an n-bit quantum register in the maximally entangled Cat-State. The app is parameterized by n.

2. Quantum Fourier Transform (QFT): Performs quantum Fourier transform on an n-bit number. The app is parameterized by n.

3. Square Root: Uses a quantum concept called *amplitude amplification* to find the square root of an n-bit number with the Grover's search technique[**?**]. The app is parameterized by n.

4. Binary Welded Tree: Uses quantum random walk algorithm to find a path between an entry and exit node of a binary welded tree [**?**]. The app is parameterized by the height of the tree (n) and a time parameter (s) within which to find the solution.

5. Ground State Estimation: Uses quantum phase estimation algorithm to estimate the ground state energy of a molecule [**?**]. The app is parameterized by the size of the molecule in terms of its molecular weight (M).

6. Triangle Finding Problem: Finds a triangle within a dense, undirected graph [**?**]. The app is parameterized by the number of nodes n in the graph.

7. Boolean Formula: Uses the quantum algorithm described in [**?**], to compute a winning strategy for the game of Hex. The app is parameterized by size of the Hex board (x, y).

8. Class Number: A problem from computational algebraic number theory, to compute the class group of a real quadratic number field [**?**]. The app is parameterized by p, the number of digits after the radix point for floating point numbers used in computation.

9. Secure Hash Algorithm 1: An implementation of the reverse cryptographic hash function [**?**]. The message is decrypted by using the SHA-1 function as the oracle in a Grovers search algorithm. The app is parameterized by the size of the message in bits (n).

10. Shor's Factoring Algorithm: Performs factorization using the Quantum Fourier Transform [**?**]. The app is parameterized by n, the size in bits of the number to factor.

Chapter 4

RKQC:RevKit For Quantum Computation

RKQC is a compiler for reversible logic circuitry. The framework has been developed to compile high level circuit descriptions down to assembly language instructions, primarily for quantum computing machines. Specifically, input files to the RKQC compiler contain descriptions of reversible circuits, and the output files are the assembly instructions for the circuit, in the ".qasm" format.

In many important quantum computing algorithms, a large portion of the modules use only classical reversible logic operations that can be decomposed into the universal set of NOT, CNOT, and Toffoli gates. Often these are referred to as "classical oracles." These oracles can also be simulated on a conventional computer.

RKQC is used by the Scaffold quantum circuits compiler as a subroutine for the compilation of purely classical reversible logic modules, or oracles. It has also been designed to operate as a stand alone tool, and can be used in this fashion. It was also developed as a full conversion of the RevKit platform [3].

4.1 Installation

4.1.1 Obtain

The first step in installation is to obtain RKQC from the public repository. It can be cloned through the Git tool by:

```
mkdir rkqc
git clone https://github.com/ah744/RKQC.git rkqc/
```

4.1.2 Build and Compile

There are two steps to building and compiling RKQC. The first is to build and obtain the dependencies, and the second is to compile the system. These are both combined in the script "build.sh", and invoking this script from the main directory will build and compile the system.

```
./build.sh
```

RKQC is now built and compiled on your system.

4.2 First Example

The example circuits are located in:

```
src/examples
```

In order to compile and run any of these examples, the "revkit" script may be invoked, located in the main directory of RKQC.

The example circuit "e001_hello_world.cpp":

```
using namespace revkit;
int main( int argc, char ** argv )
{
   qbit a;
   qbit b;
   qbit c;

   NOT(a);
   cnot(b, c);
   toffoli(a, b, c);

   return 0;
}
```

can be compiled with RKQC by invoking the revkit script and the example circuit name, without the file extension, from the main directory of RKQC. This circuit has three input and output signals, and three reversible logic gates.

```
./revkit e001\_hello\_world
```

The compiler then creates the output ".qasm" file, containing the quantum assembly language instructions generated for this circuit. For "e001", the file "e001_hello_world.qasm" contains:

```
qubit w0
qubit w1
qubit w2
X w0
cnot w1, w2
toffoli w0,w1,w2
```

4.3 Declaration of Circuits

The construction of RKQC is designed to be as similar as possible to C++ programs which contain a number of functions. To this end, circuits are specifiable through a main function, as well as through various submodules that are declared exactly as common C++ functions. Exactly one module must be named "int main", however this module, as well as any module, can call any other module at any time.

4.4 Signal Types

There is one major type of signal in RKQC, the **qint**. This type forms the basis for the five other types of signals that are derived from this main class:

- qbit
- zero_to_garbage ancilla
- zero_to_zero ancilla
- one_to_garbage ancilla
- one_to_one ancilla

All of the above signals can be declared and used in the specification of gates and circuit descriptions. The primary differences are that the "_to_garbage" ancilla are ancilla signals that are not guaranteed to maintain their original state during computation, while the two other ancilla types do make this guarantee.

As will be explained later on, all of these signals are of a base type **qint**, which allows for circuit description flexibility when creating and passing parameters to submodules.

4.5 Creating and Using Modules

At the circuit level, both user defined modules as well as built in modules can be instantiated in the same fashion.

In the example circuit "e008_swap.cpp", the built in function "cnot" is called, creating CNOT gates on the qubits passed in as parameters:

```
int main( int argc, char ** argv )
{
    qbit a;
    qbit b;

    cnot(a, b);
    cnot(b, a);
    cnot(a, b);

    return 0;
}
```

In exactly the same fashion, a circuit designer can create a submodule, and instantiate it within the main function, as in file "e009_swap_triples.cpp":

```
void swap_bits( qint x, qint y){
    cnot(x, y);
    cnot(x, y);
    cnot(x, y);
}

int main( int argc, char ** argv ){
```

```
    qbit a0;
    qbit a1;
    qbit a2;

    qbit b0;
    qbit b1;
    qbit b2;

    swap_bits(a0, b0);
    swap_bits(a1, b1);
    swap_bits(a2, b2);

    return 0;
}
```

4.6 Arrays

RKQC also allows for multi-dimensional arrays of signals to be declared and used in circuits. To demonstrate this, file "e010_swap_triples_2.cpp" replicates the functionality of "e009_swap_triples.cpp" with arrays:

```
void swap_bits( qint x, qint y){
    cnot(x, y);
    cnot(x, y);
    cnot(x, y);
}

int main( int argc, char ** argv ){
    qbit a(3);
    qbit b(3);

    swap_bits( a[0], b[0] );
    swap_bits( a[1], b[1] );
    swap_bits( a[2], b[2] );

    return 0;
}
```

4.7 Signals and Gates

Gates are built in modules that can be instantiated in a variety of methods, with a variety of signal types. Specifically,

- single qint signals,
- qint arrays,
- single qbit signals,

- qbit arrays,
- single ancilla signals, and
- ancilla arrays

all are capable of being passed through as parameters to built in gates. For an example of this, file "e002_signals_and_gates.cpp" calls all of the intrinsic built in gates with all of the different signal types:

```
int main( int argc, char ** argv )
{

    qbit a;
    qbit b;
    qbit c;
    qbit d(10);
  qbit e(10);
  qbit f(10);
/*------------ Gates Called By Qubit --------------*/
    NOT(a);
    cnot(a,b);
    toffoli(a, b, c);
/*------------ Gates Called By Register Index -----*/
    NOT(d[0]);
    cnot(d[1], d[9]);
    toffoli(d[0], e[0], c[0]);
/*------------ Gates Called By Mixed Register Index & Qbits -----*/
    cnot(a, d[0]);
    cnot(e[0], b);
  toffoli(d[0], a, b);
  toffoli(a, d[0], b);
  toffoli(a, b, d[0]);
  toffoli(d[0], e[0], a);
  toffoli(d[0], a, e[0]);
  toffoli(a, d[0], e[0]);
/*------------ Gates Called By Full Qbit Reg ------*/
  NOT(d);
  cnot(d, e);
  toffoli(d, e, f);

    return 0;
}
```

4.8 Assign Value

A commonly used function of assigning value of one signal to another is built into RKQC in the form of a function:

```
assign_value_of_b_to_a();
```

This function can be instantiated a number of different ways, with integer constants or with two **qint** signals. These instantiation methods are detailed in file "e004_assign_value.cpp":

```
int main( int argc, char ** argv )
{
    qbit a;
    assign_value_of_b_to_a(a, "0", 1);

    qbit b(8);
    assign_value_of_b_to_a(b, "1", 8);

    qbit c(8);
    assign_value_of_b_to_a(c, b, 8);

    return 0;
}
```

4.9 Integer Arithmetic Modules

RKQC comes with implementations of quantum adders and multipliers developed in recent theoretical work. For instance, the quantum ripple-carry adder implemented in the function:

```
a_eq_a_plus_b();
```

is derived from work by Wang et. al [4], and an example of usage is contained in file "e005_adder.cpp":

```
int main( int argc, char ** argv )
{
    qbit a(32);
    qbit b(32);

    a_eq_a_plus_b(a,b,32);

    return 0;
}
```

Alternatively, an implementation of an ancilla-free addition circuit is also contained within RKQC, derived from earlier work by Cuccaro [1].

Chapter 5

Expanding ScaffCC

ScaffCC is completely open-source and may be expanded to accomodate future needs of quantum circuit analysis and optimization.

At the core of ScaffCC are the LLVM compiler passes, which operate on LLVM IR (.ll) code. All LLVM passes are stored in:

```
llvm/lib/Transforms/
```

Passes specific to ScaffCC are stored in:

```
llvm/lib/Transforms/Scaffold
```

In general, to run a pass in LLVM, you invoke the opt program as follows:

```
build/Release+Asserts/bin/opt -S -load build/Release+Asserts/lib/Scaffold.so
   {pass_name} {input_ll_file} > {output_ll_file} 2> {log_file}
```

Note that *pass_name* referes to the unique name of the pass by which it is registered in the LLVM system, by invoking the following in the implementation of the pass:

```
static RegisterPass<{pass_name}> X({pass_name}, {description_of_functionality});
```

To write a new pass, start by looking at the previously implemented examples in this directory, and consulting the LLVM Documentation: `http://llvm.org/docs/WritingAnLLVMPass.html`

Bibliography

[1] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, "A new quantum ripple-carry addition circuit," Sep 2004.

[2] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "ScaffCC: A framework for compilation and analysis of quantum computing programs," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/2597917.2597939

[3] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, "RevKit: An open source toolkit for the design of reversible circuits," in *Reversible Computation 2011*, ser. Lecture Notes in Computer Science, vol. 7165, 2012, pp. 64–76, RevKit is available at www.revkit.org.

[4] F. Wang, M. Luo, H. Li, Z. Qu, and X. Wang, "Improved quantum ripple-carry addition circuit," *Science China Information Sciences*, pp. 1–8, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11432-015-5411-x