Vision on Next Level Quantum Software Tooling

Rob F.M. van den Brink
TNO
ICT Department
The Hague, The Netherlands

Frank Phillipson
TNO
ICT Department
The Hague, The Netherlands
Email: frank.phillipson@tno.nl

Niels M.P. Neumann
TNO
ICT Department
The Hague, The Netherlands

***Abstract*—Software tools for programming gate-based quantum computers are being developed by many parties. These tools should now grow towards a phase where they support quantum devices running realistic algorithms to outperform classical algorithms on digital computers. At this moment, they lack capabilities for generic gates, capabilities for quantum debugging and generic quantum libraries. This paper gives a view on the functionalities needed for such software environments looking at the various layers of the software stack and at the interfaces for quantum cloud computing.**

Keywords–Quantum Software; Quantum Computing; Quantum Compiler.

I. Introduction

Quantum computers are still in an early stage of development, and experimental quantum processors are getting to support up to a few dozen of qubits [1][2]. That number is growing, and support for a large number of qubits is just a matter of time. Where we speak of quantum computers in this paper, we mean gate based quantum computers and not quantum annealers. Software tools for programming gate-based quantum computers are also in an early stage of development, where the basics, however, were already set 20 years ago [3]–[5]. Currently, these software tools are mainly supporting a set of low-level quantum instructions, embedded in a classical (digital) programming language. They are adequate for running rudimentary quantum applications [6] that can already run on only a few qubits. Quantum computing should now grow towards a phase where the development of quantum software must get more emphasis. This is comparable with the sixties of the previous century, when programming tools like Fortran and Algol brought a much higher abstraction to digital computers than just assembly language.

Since it is not obvious what 'higher abstraction' means for quantum computing, this paper discusses what functionality is needed in the next level of quantum computing and how this can be implemented in a structured way, by means of a stack of software layers. Important here is that, in this paper, we assume that (1) quantum processors remain bulky devices for a long time (like digital computers in the beginning), (2) that real quantum applications will always be a mix of digital and quantum computing, where only a part of the problem is solved in a quantum manner, and (3) that we want to make this available for a larger public, that has limited knowledge about the underlying quantum layer. For this, we require (1) a strict interface between remote quantum hardware, (2) local software that runs at the user side that can, (a) independent of the used local programming language, interact with the remote hard- and software, (b) independent of the used local programming language, make use of high level quantum algorithms in libraries, and (c) has profound quantum specific debugging capabilities where (small versions of) the desired algorithms can be analysed, while stepping through the program in debug mode, up to the underlying quantum states using simulations.

TABLE I. Shortlist of languages being used for quantum software tools.

C / C++ based (>30)	Matlab / Octave based (12)
F# based (1)	Maxima based (2)
GUI based (>10)	NET based (4)
Java based (>15)	Online service (10)
JavaScript based (1)	Perl / PHP based (3)
Julia based (1)	Python based (>6)
Maple based (3)	Scheme/Haskell/LISP/ML based (8)
Mathematica based (8)	

The remainder of this paper is organised as follows. In Section II, we give an overview of the current state of the art, by making a functional grouping of existing tools and their capabilities. In Section III, we discuss the functionality we expect from a quantum software environment to meet the above requirements. In Section IV, we sketch our desired software environment, by defining layers and discussing the functionality of each of those layers and the placement of the separation between local and cloud, followed by some examples of (later defined) quantum function libraries in Section V. We conclude in Section VI with a summary and conclusions.

II. State of the Art

Many quantum software tools are freely available via the Internet, and overviews with good summaries of those tools can be found in [7]–[9]. Most of these tools are still in the development phase. The list of available tools is too long to be duplicated here, but a first impression can be gained by organising them according to the used programming language. Table I shows the programming languages that are being used for implementing quantum tools, and the number of each of these quantum tools. What can be observed in this table is the wide range of different programming languages that has been used. There is no clear winner from that, simply because almost any classical programming language is suitable for this job, with each of these languages having its own advantages and disadvantages.

An other way to group the tools is taking a high-level view on their functionality. We recognise the following functional

TABLE II. Examples of quantum compiling tools

Name	Remarks
Scaffold or ScafCC	C-alike, updated version of CLANG. Compiles to LLVM & QASM [10].
Liquid (Microsoft)	F#-alike, updated version of F# (called Q#), with an emulator on board [11].
Quipper	Haskel-alike [12]
QCL	mix of C and Pascal alike, detailed language specification with emulator on board [13].

groups: (a) quantum compilers, (b) quantum function libraries, and (c) back-end quantum simulators. We are aware that some tools may not fully fit into one particular group and these groups may overlap, but it is helpful to identify several high-level functionalities. We will now discuss the characteristics of each of these groups one by one.

A. Quantum compilers

The word *compiler* refers in this context to a tool that translates an entire program from higher level statements into some lower level instructions (e.g., binary or assembly). The execution of that program is started thereafter. The purpose of a compiler is to generate low level quantum instructions (in assembly or as native code), from a mix of low and high level quantum statements, of classical control statements and loops, of organised code in reusable routines, etc.

Quantum tools that present themselves as quantum compilers are mainly modifications from an existing (classical) compiler. The result is a changed syntax to have it extended with quantum specific instructions. The present extensions are mainly to offer a syntax for elementary instructions at the level of quantum assembly code. Table II offers a few examples of those tools.

Their main disadvantage is that creating a new language may break access to existing code libraries with classical algorithms. For instance, the development of quantum algorithms for outperforming classical machine learning algorithms will draw significant benefits if classical code libraries with, for example, neural network algorithms are well-accessible from the same software environment.

Next to this, quantum compilers that introduce new languages, or break compatibility with existing languages lack in most cases powerful development and (quantum) debugging tools.

B. Quantum function libraries

The term *function library* refers in this context to the use of an existing and well-supported (classical) programming environment, where the quantum programmer can call quantum-specific routines from a library. Their purpose is similar to that of compilers, with the difference that the generation of low level quantum code occurs in run time. Examples of such quantum tools are: Qiskit (IBM), Quantum Learning Machine (QLM, Atos), PyQuil (Rigetti), ProjectQ (ETH Zurich), and OpenQL (TU Delft).

The approach of using function libraries brings hardly any limitation, think of generating thousands of dedicated quantum instructions by a single call to a library function. For instance, a single call to a routine for a Quantum Fourier Transform or a matrix expression, that automatically generates hundreds of quantum instructions operating on tens of qubits. However, the present quantum function libraries have a strong focus on low-level quantum computing, and may only be targeted at a specific quantum processor. Quantum programming with the present tools is mainly a matter of calling routines for generating individual low level quantum instructions (like assembler). By calling several of them with different parameters, one can generate a sequence of quantum instructions to build a quantum circuit. The identification and handling of higher level quantum instructions is still a topic of further research.

Some of these tools already offer powerful quantum debugging capabilities, like the generation of a drawing of the generated quantum circuit and access to a build-in quantum simulator. Such simulator can return intermediate results, like, for instance, a quantum state, that can never be obtained using a real quantum processor. Some of these tools also offer a graphical user interface, for positioning a few quantum gates into a quantum circuit to process a few qubits. That approach mainly serves an educational purpose for those who are setting their first steps in quantum computing. However, as soon as your quantum program grows in size, the use of scripting for calling quantum functions may become more convenient.

C. Back-end simulators

The term *back-end simulator* refers in this context to a tool that reads low level quantum assembly and executes them in a manner like a real quantum processor would do. As such, it becomes irrelevant for the quantum programmer in what language such tool is written since that tool is instructed directly via low level assembly instructions. The concepts of high level instructions and function libraries are not applicable here and therefore we consider it as another group.

Back-end simulators contain a very simple translator, to feed low level assembly instructions (usually stored in an ASCII file) to a build-in simulation engine. They are by definition dedicated to low-level functionality only, and can simulate/emulate on a digital computer what a real quantum processor would have returned. These tools aim at simulating a real quantum processor as good as possible on a classical digital computer. They may even try to simulate the physical imperfections of a particular quantum processor as well. For instance, by adding some random mechanism (noise) to mimic the loss of quantum coherence after executing more and more quantum instructions, or by deliberately accounting for topological limitations of a particular quantum processor.

These tools may be offered as a stand-alone tool, or as part of a larger software environment (sometimes referred to as ‘virtual quantum machine’). The translation functionality they possess, may also be used to interface with a real quantum processor, but that is out of scope here. Examples of such quantum tools are (modules inside) QX (QuTech) [14], QLM (Atos) [15], QVM (Rigetti) [16] and Quantum Experience (IBM) [17].

These tools serve purposes other than quantum function libraries and quantum compilers do. They are valuable to study the quantum assembly language, to study how to deal with limitations in the quantum instruction set of the target machine, or to study quantum error correction methods against decoherence. They are also valuable to study the interfacing between a local software environment and a quantum processor somewhere in the cloud.

TABLE III. Example of two different dialects of quantum assembler, both representing the same quantum circuit

Atos-QASM	QX
BEGIN	
qubits 18	qbits 18
cbits 0	.ckt
H q[0]	H q[0]
H q[5]	H q[5]
CTRL(PH[1.570]) q[15],q[16]	CR q[15], q[16], 1.570
CTRL(PH[0.785]) q[14],q[16]	CR q[14], q[16], 0.785
CTRL(PH[0.392]) q[13],q[16]	CR q[13], q[16], 0.392
H q[15]	H q[15]
...	...
H q[16]	H q[16]
END	

III. Desired software functionality

In this section, we describe the functionality we expect in quantum software, such that it supports implementation of more complex, hybrid digital and quantum algorithms and that it will enable a larger public that has limited knowledge about the underlying quantum layer to access quantum computing.

A. Commonly available functionality

In spite of all functional differences between the discussed tools, they all share a common functionality: the concept of quantum circuits for representing a set of quantum instructions via interconnected gates, and the concept of quantum assembler or Quantum Assembly Language (QASM) as a language for describing those circuits as a list of sequential quantum instructions. There are several of these quantum assembly languages, each with their own dialect (syntax), but from a pure conceptual point of view they all are roughly the same. Table III shows an example of two different QASM dialects, both describing the same quantum circuit. The first one shows the syntax used by QLM (from Atos), the second one shows the syntax used by QX (within Quantum Inspire).

A QASM file is typically a sequence of individual QASM instructions, embedded between a header (with declarations) and a footer. Some QASM tools also allow for grouping those instructions into macros (like functions) to simplify the use of the same code multiple times. Each QASM instruction is build-up from (a) an instruction name, (b) optional parameters, and (c) a list of qubit identifiers on which this instruction should operate. Each instruction can change the contents of a 'quantum register' (via gates or measurements), and many of them in sequence define a quantum circuit. There is an apparent consensus on the naming of several of these gates, for instance, instruction names like H, X, Y, Z, but this consensus does not hold for all gates. This difference can be confusing, but is not a big issue when well defined. When these names are well specified by means of the corresponding matrix representation, the conversion from one QASM dialect into another is pretty straight forward. And when a particular gate is not available in one QASM, it can always be created in another QASM via a combination of a few other gates. Note that some QASM dialects start their counting of qubits from 0, while others start from 1. This is only a matter of convention and preference, mainly driven by the supporting language, and not a big issue either. In conclusion, one may say that it is relatively easy to translate one QASM dialect into another one.

B. Next level functionality

To reach the desired level of quantum programming that supports the implementation of more complex, hybrid quantum algorithms and to enable a larger public that has limited knowledge about the underlying quantum layer to access quantum computing, we need a next level of software functionalities. Examples of these functionalities are:

Desired capabilities for generic gates

- Define circuit libraries with generic gates, with an arbitrary number of qubits, and callable as if it was a single instruction. OpenQASM [18] does supports the concept of macros and can provide this functionality, but that concept is not available in all QASMs. The desired circuit library should generate, for instance, a Quantum Fourier Transform, or a circuit for modular exponentiation by one instruction/call for an arbitrary large number of qubits.
- Define circuit libraries with generic gates in terms of an unitary matrix or a matrix expression, while the tool translates that into circuits with only basic gates. This capability is currently a weak point for almost all tools. It may be available for one or two qubit gates, but the issues get problematic for more qubits.

Desired capabilities for quantum debugging

- The capabilities to let the software draw a circuit from a QASM specification to debug what has been specified. Tools like Liquid, QLM and ProjectQ give support for that, but it should be available on all tools of interest.
- The capability to read out the full vector (or full matrix) with complex numbers representing the present quantum state or circuit (also at intermediate points). This is impossible with a real quantum processor, and only possible with a simulator. However, it is a very powerful and essential debugging facility. Some simulation tools do support this, but there are very primitive solutions among them.
- The capability to analyse generic quantum states and gates (during simulation), while stepping through the program in debug mode, using sophisticated linear algebra tooling.
- The capability to visualise in an abstract manner relevant aspects of the (full) state vector or (full) matrix in case they are too large for a full numerical inspection. For instance, histograms, magnitude plots by means of colours, etc,

Desired quantum libraries

- Libraries that implement many quantum functions/circuits, callable from your quantum program. Many functions are well known from the literature, but inserting, for instance, a Quantum Fourier Transform operating on an arbitrary number of qubits (in arbitrary order) should be as simple as inserting a single qubit X gate. Section V provides further details.
- Running quantum programs from a programming environment with good access to classic libraries with legacy solutions for the quantum application under study. This may mean (a) a language that is different

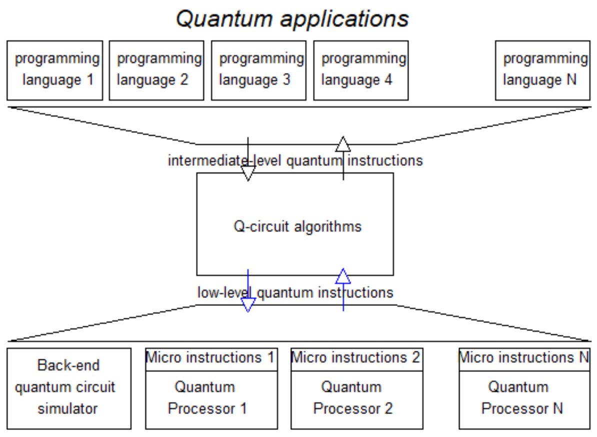

Figure 1. Software stack for quantum tooling with different layers

from the implementation language of the quantum tools, and/or (b) software that prepares the quantum application locally (for debugging reasons) for running it on a remote quantum processor (via a well-supported API, Application Programming Interface), and/or (c) software that uses a different operating system from the one used by the quantum processor. We will also discuss this further in the next chapter.

These desired functionalities are examples only, and the list is not complete. They have currently a strong focus on (quantum) debugging capability, flexibility, and libraries. The present tools may implement fragments of this list, but we have not found a tool that can support them all in a convenient manner.

IV. Desired software environment

If we know what functionality we need in the software to enter the next level of quantum computing, we should define where and how this functionality should be implemented and where the separation between local and cloud can be placed.

A. Layered software stack

To define where and how the functionality should be implemented, we use a full software stack with different layers, as shown in Figure 1, covering functionality from quantum applications down to quantum processors. The concept in Figure 1 is very similar to the picture shown in [19][20]. We, however, put more emphasis on the desired functionality within the intermediate levels.

1) Functionalities of bottom Layers: The very bottom of the low-level layers are reserved for the quantum processors, each equipped with dedicated hardware for controlling it via (binary) micro instructions. For instance, to change the quantum state of a quantum processor via dedicated pulses. A software layer directly above this hardware allows for translating a sequence of low-level quantum instructions (e.g., assembly or binary code) into dedicated pulses for the hardware. This software layer should hide the hardware difference among different quantum processors as much as possible, in order to program them with uniform instructions that are more or less hardware-independent. In practice, very different quantum processors will be implemented, based on different physical principles, and each with their own micro instruction sets. The quantum programmer should not be bothered by that.

These instructions are still very low-level, and the use of some QASM dialect is the most obvious choice here. These instructions allow for defining quantum circuits with basic gates, where the word basic refers to a set of predefined gates operating on 1 or 2 qubits only. These instructions may be restricted to the (hardware) instruction range of the target quantum processor, and may also account for topological (hardware) limitations.

Since quantum processors are still under development, the use of a back-end circuit simulator also has its place in the lower layers. Their aim is to simulate a real quantum processor as fast as possible, with as many qubits as the used digital computer platform can handle, and to simulate/emulate all its imperfections and, if applicable, all topology limitations as good as possible.

2) Functionalities of intermediate Layers: A more intermediate level (the Q-circuit algorithms box in Figure 1) gives the quantum programmer access to all kinds of quantum algorithms, in a uniform manner, ideally independent from the used quantum hardware and programming languages. It is, for instance, equipped with all kinds of algorithm libraries (examples can be found in the next section) and quantum debugging capabilities to design the quantum-specific parts of applications.

These instructions are at least capable of defining arbitrary circuits with generic gates. This means within this context that they can operate on an arbitrary number of qubits, far beyond the instruction set of the quantum processor, and they may even be specified via (unitary) matrices or high level expressions. The translation from quantum circuit with a few generic gates into circuits with many basic gates is the proper place to perform gate and qubit optimisation. The best results can be achieved when this translation is partly guided by control parameters representing some of the hardware limitations of the target quantum processor (hardware aware). These control parameters are set only once for a particular target quantum processor, and preferably invisible in the instructions with generic gates (hardware agnostic). As such, this translation is an important intermediate step in quantum programming, applicable to both quantum compilers and tools based on function libraries.

Figure 2 provides an example of a quantum circuit with generic gates. At a first glance, generic may look the same as basic gates, but in this case the gates can also be black boxes operating on many qubits simultaneously specified as matrices, expressions, or standard functionalities.

One may still consider the supported instructions as rather low-level, but the distinction between low and intermediate level brings a significant advantage. If quantum computing is offered via the cloud, for running quantum applications from all over the world, then the interface between the lower and intermediate layer is a natural interface for the cloud based quantum computer. And the use of one or more QASM dialects is a natural component in the interfacing between these layers. It means that software in the intermediate layers may fully run on a local computer, while software in the lower layer should typically run on a remote quantum host.

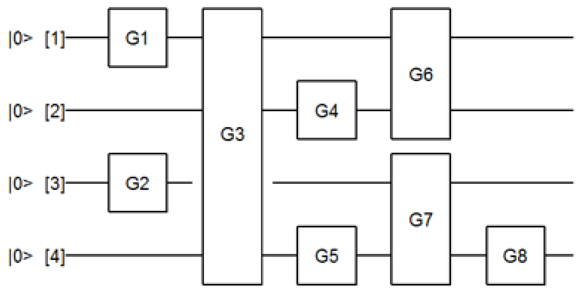

Figure 2. Example of a quantum circuit with generic gates.

3) Functionalities of higher layers: The higher layers (above the Q-circuit algorithms box in Figure 1) are typically reserved for classical programming environments that are extended with quantum capabilities for solving dedicated subproblems. Here the new group of users, with limited knowledge of underlying quantum techniques should be able to play around. It would be a waist of effort if each programming environment has to develop its own collection of libraries with quantum-specific algorithms. Therefore it is far more efficient to equip them only with language-specific interfaces, wrapped around common quantum libraries and debugging capabilities from the intermediate layer.

Quantum applications will most likely be a hybrid mix of classical programming concepts and quantum-specific algorithms. This means that the classical programming languages call a quantum algorithm only when needed for solving particular subproblems. These classical programming languages should offer the following capabilities:

- Good access to classical software libraries, with dedicated algorithms for the problem area. Think of libraries for artificial intelligence applications. But think also of access to quantum circuit libraries with dedicated quantum-specific algorithms like quantum solutions for dealing with decoherence errors or for decomposing a large number into its prime factors.
- Good access to quantum debugging capabilities, also for the quantum specific aspects. Think of inspecting quantum states and matrices of quantum circuits via a build-in simulator and drawing quantum circuits from instructions. These debugging capabilities should easily interact with the higher layers, all in a very interactive and flexible manner.

As such, the universal programming language for everybody does not exist, and therefore the best solution for that is that the intermediate layers offer access to quantum-specific libraries for any programming language of interest. It supports many quantum circuit algorithms as well as quantum-specific debugging capabilities.

The use of languages with build-in support for linear algebra expressions, that are also available as interpreter, give the user powerful extra capabilities for quantum debugging. These tools allow for inspecting and manipulating intermediate results of circuit matrices and state vectors in a very interactive manner during simulation. Linear algebra languages like Matlab/Octave, and to some extend Python with Numpy, are examples of languages offering the desired linear algebra capabilities in an interactive environment and offer access to a broad spectrum of classical code libraries.

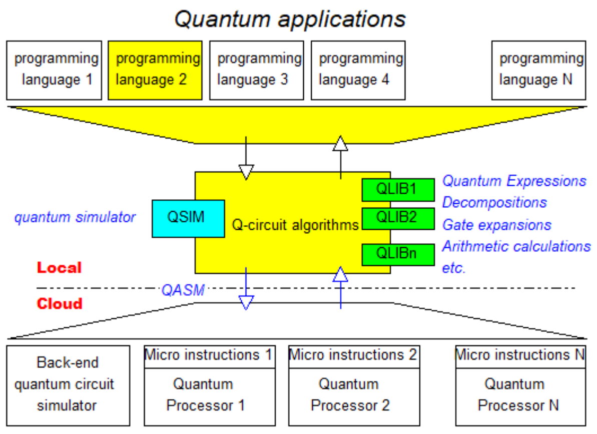

Figure 3. Quantum software stack overview.

B. Separation between Local and Cloud

It is assumed that quantum computers remain big installations with bulky refrigeration equipment for a long time. Commercial deployment of quantum computing will then mean a quantum computer hosted in a remote building, offering access via the cloud to many users all over the world. Today, experimental quantum computers already give access via the cloud, but mainly in a restricted manner; end users have to setup a remote terminal session with the hosting computer and they should develop and run everything on that host. This is quite inconvenient as it limits rapid interaction with local software, like exchanging intermediate results with local debugging software. Moreover, commercial users may not be willing to share their source code with the hosting organisation. Exchange of low-level code (binary or assembly) gives then a similar protection as is common today for distributing programs as a binary executable. In that case, end users develop, test and debug on a small scale (locally) at an intermediate level, and then send low level quantum instructions to a remote host in order to run at full quantum speed and size.

The most convenient way of implementing that is therefore not by opening remote terminal sessions, but by accessing the remote quantum computer via an API. An API allows the user to program his (quantum) application in a language that differs from the programming language being used by the remote host. The intermediate layers will then send QASM-alike quantum instructions to a remote host, while the end-user experiences it as if it runs locally. Figure 3 illustrates this interaction model, where the interface between local and cloud is situated between the intermediate and lower layers. The intermediate layers are equipped with all kinds of libraries for quantum specific calculations, as well as debugging capabilities via a local quantum simulator. These libraries generate the required low level QASM instructions and can forward them through a language-independent interface to the lower layers running in the cloud.

V. Example libraries

To get an idea of the type of algorithms that can be implemented via libraries, we will discuss a few examples: libraries that convert a mathematical expression into a quantum circuit, libraries that decompose a generic gate or generic matrix description of such gates into circuits with basic gates and libraries that convert arithmetic calculations into a quantum circuit.

A. Quantum expression libraries

The first example is dedicated to generic gates described by mathematical functions for generating special unitary matrices. Think, for instance, of the following matrix expressions:

$$G_1(a) = e^{j*a*Z}$$
$$G_2(a) = e^{j*a*ZZ}$$
$$G_3(a) = e^{j*a*ZZZZ}$$
$$G_4(a, b, c, d) = e^{j*(a*XX+b*YY+c*ZZ+d*II)}$$

Where

- a, b, c, d are parameters with arbitrary real values;
- X, Y, Z are the Pauli matrices;
- I is the unity matrix;
- XX, YY, ZZ, II are the Kronecker products of X, Y, Z and I with themselves;
- $ZZZZ$ is the Kronecker product of ZZ and ZZ.
- j indicating the imaginary number $\sqrt{-1}$.

The solution for the first two examples is quite easy, and can be found in almost any basic text book on quantum computing. But finding a good solution for the last one is less obvious. Fortunately, it can be represented by a relative simple circuit [21]. One can easily imagine that the list of such expressions is virtually unlimited, which illustrates the value of bringing them all together into a well-organised quantum expression library.

B. Quantum decomposition libraries

Another example occurs when mathematical functions are not available in the expression libraries, as discussed in the previous section. In those cases a solution may be to use a numerical evaluation of that function into a unitary matrix with complex numbers. For instance, to produce an 8×8 matrix representing a 3 qubit gate. It is not that difficult to decompose an arbitrary matrix into the product of much simpler matrices, but a simpler matrix does not automatically mean a simpler quantum circuit. Examples of useful matrix decompositions are singular value decompositions, sine-cosine decompositions [5] or QR-decompositions with Givens rotations [22]. However, these decompositions can easily result in large quantum circuits with an exponential number of basic gates, and can also produce quite inefficient solutions. Decomposition is still an important topic for further research, because we still need algorithms that convert arbitrary matrices into quantum circuits, such that it is fully automatic and produces an efficient circuit as well.

When the matrix is not fully arbitrary, dedicated solutions may yield far more efficient solutions then the generic approach. Examples are

TABLE IV. Explanation of used gates in Figures 4c and 4b.

Peres approach	QFT approach
$G1 = c([1+q4, 1-q4; 1-q4, 1+q4]/2)$	$G1 = cR(\pi/2)$
$G2 = c([1+q2, 1-q2; 1-q2, 1+q2]/2)$	$G2 = cR(\pi/4)$
$G3 = G2'$ (conjugated transpose of G2)	$G3 = cR(\pi/8)$
$G4 = G1'$ (conjugated transpose of G1)	$G4 = Rs(\pi/8)$
	$G5 = Rs(\pi/4)$
where	$G6 = Rs(\pi/2)$
$q2 = (-j)$	$G7 = Rs(\pi)$
$q4 = \sqrt{(-j)}$	$G8 = cR(-\pi/2)$
$c([g_{11}, g_{12}; g_{21}, g_{22}]) =$	$G9 = cR(-\pi/4)$
$[1, 0, 0, 0;$	$G10 = cR(-\pi/8)$
$0, 1, 0, 0;$	
$0, 0, g_{11}, g_{12};$	where
$0, 0, g_{21}, g_{22}]$	$Rs(\phi) = [1, 0; 0, e^{(j*\phi)}]$
	$cR(\phi) = c(Rs(\phi))$

- U is a 1-qubit gate specified by a 2×2 unitary matrix with arbitrary complex numbers;
- cU is a controlled version of U, representing a 4×4 matrix;
- ccU is a double controlled version of U, representing a 8×8 matrix.

Such generic gates can be decomposed into multiple basic gates and the simplest one can be found in almost any basic text book on quantum computing. But finding solutions for multiple controlled gates is not obvious, and should be generated automatically by a single function call, for an arbitrary matrix U and with an arbitrary number of control inputs. One can easily imagine that the list of different decompositions is virtually unlimited, which illustrates the value of bringing them all together into a well-organised 'quantum decomposition library'.

C. Quantum arithmetic libraries

Several quantum applications make use of algorithms where discrete numbers are represented by distinct quantum states. For instance, algorithms that make use of modular additions, modular multiplications or modular exponentiations. In those cases it is not obvious what the most efficient way is to implement these on many qubits. We will show three example circuits of how to calculate something 'simple' like the modular increment of a discrete number encoded in 4 qubits. The first one (Figure 4a) can be found in any textbook, looks quite simple, however, requires gates with many control inputs. The second one (Figure 4b) with Peres gates is also known [23], looks more complicated, but requires only single qubits gates and single controlled qubit gates. The used gates are explained in Table IV.

The third example (Figure 4c) using quantum Fourier transforms [24], however, appeared to be the most simple one of these three, in the sense that only single qubit gates and controlled phases are used. This may illustrate that generating an algorithm with the most efficient circuit is not obvious even for a very simple problem.

There are many more of these modular arithmetic calculations, each of them with multiple implementations. Their details are out of scope here, but it may again illustrate the value of bringing all these implementations together into a well-organised 'quantum arithmetic library'.

```
1 :--cccX-----------
      |
2 :---o-ccX---------
      |  |
3 :---o--o--cX------
      |  |  |
4 :---o--o--o--X----
```

(a) Basic circuit of evaluating the modular increment of a number, encoded in 4 qubits.

```
1 :---G1-G1-G2-------------G4-G3------
      |  |  |              |  |
2 :-- |  | -o--G2-G2----G3 | -o-------
      |  |     |  |     |  |
3 :-- | -o---- | -o--cX-o--o----------
      |        |     |
4 :---o--------o-----o-----------X----
```

(b) Peres approach of modular increment: looks more complicated than the basic circuit, however, all gates are operating on only one or two qubits.

```
1 :---H--G1----G2-------G3----------G4---------------------------G10-G9-G8-H--
         |     |        |                                         |  |  |
2 :------o--H- | -G1--- | -G2----------G5----------------G9-G8-H- |  | -o-----
               |  |     |  |                             |  |     |  |
3 :------------o--o--H- |  | -G1----------G6-------G8-H- | -o---- | -o--------
                        |  |  |                    |     |        |
4 :---------------------o--o--o--H-----------G7-H--o-----o--------o-----------
```

(c) QFT approach of modular increment: offers the simplest implementation, although looks the most complicated.

Figure 4. Three examples of evaluating the modular increment of a number, encoded in 4 qubits.

D. *Other quantum libraries*

The list of useful libraries is unlimited. Think of circuits to generate a Quantum Fourier Transform, or to step through a quantum random walk. The same applies for the best way to deal with error corrections, to deal with the topology limitations of a particular quantum computer, or to build standard circuits by using only the native gates of a particular quantum computer (those that can be made with a single pulse). Note that implementations of basic gates like X, Y, Z, may require more than one pulse (this depends on the physical implementation being used). Existing tools have some of those algorithms implemented. However, a common library that can be used by different quantum tools is currently only in an early phase of development.

VI. Summary and conclusions

This paper identified what functionality is needed in software that enables the next level of quantum computing and proposes a way to implement this throughout the software stack. This next level quantum computing makes it possible to run more complicated algorithms on quantum computers in the cloud by a larger public.

The needed functionalities were categorised in functionalities for generic gates, for quantum debugging capabilities and quantum libraries.

A layered quantum software stack was discussed with extra attention to the functionality of the intermediate layers. Important is a clear separation between the software that runs locally and software that runs on a remote host computer that controls a quantum processor.

Looking at the layered stack, the lower layers contain one or more quantum processors and/or back-end simulators and are typically at the remote host. The intermediate and higher layers are typically running locally. This enables also a clear separation between the (classical) programming languages being used for the quantum application, and software that implement quantum-specific algorithms and quantum-specific debugging capabilities. The intermediate layers contain all kinds of libraries, and a local simulator to offer this to the higher layers.

The thoughts discussed in this paper are to provide input to a bigger research agenda on software development for quantum computing. A first step to make the desired functionality happen is increasing the effort on software development for the intermediate layers as well. Activities that deserve more attention are: (a) Interfacing between a local computer and quantum processor at a remote host. This should not only be defined in a language-independent manner, but also be defined for different quantum processors (at different remote hosts) in a common manner. (b) Collecting a wide variety of quantum circuit algorithms into libraries, in a uniform manner that can be used on any quantum processor. This may require an automated translation from an abstract QASM syntax tree (generated by the libraries) into the various QASM dialects of different processors.

This paper has shown a few examples of those libraries. All kinds of good algorithms are scattered around in literature, and paying more attention on bringing them all together into common libraries is a good start.

References

[1] G. W. Dueck, A. Pathak, M. M. Rahman, A. Shukla, and A. Banerjee, “Optimization of circuits for ibm’s five-qubit quantum computers,” in 2018 21st Euromicro Conference on Digital System Design (DSD). IEEE, 2018, pp. 680–684.

[2] X. Fu et al., “A microarchitecture for a superconducting quantum processor,” IEEE Micro, vol. 38, no. 3, 2018, pp. 40–47.

[3] S. Bettelli, T. Calarco, and L. Serafini, “Toward an architecture for quantum programming,” The European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics, vol. 25, no. 2, 2003, pp. 181–200.

[4] P. Selinger, “A brief survey of quantum programming languages,” in International Symposium on Functional and Logic Programming. Springer, 2004, pp. 1–6.

[5] R. R. Tucci, “A rudimentary quantum compiler (2cnd ed.),” arXiv preprint quant-ph/9902062, 1999.

[6] P. J. Coles et al., “Quantum algorithm implementations for beginners,” arXiv preprint arXiv:1804.03719, 2018.

[7] R. LaRose, “Overview and comparison of gate level quantum software platforms,” arXiv preprint arXiv:1807.02500, 2018.

[8] “Quantum Computing Report,” [Online] URL: https://quantumcomputingreport.com/resources/tools/ [accessed: 2019-03-04].

[9] “Quantiki,” [Online] URL: https://www.quantiki.org/wiki/list-qc-simulators [accessed: 2019-03-04].

[10] “Scaffold,” [Online] URL: https://github.com/epiqc/ScaffCC [accessed: 2019-03-04].

[11] “Liquid,” [Online] URL: https://www.microsoft.com/en-us/quantum/development-kit [accessed: 2019-03-04].

[12] “Quipper,” [Online] URL: https://www.mathstat.dal.ca/~selinger/quipper/ [accessed: 2019-03-04].

[13] “QCL,” [Online] URL: http://tph.tuwien.ac.at/~oemer/qcl.html [accessed: 2019-03-04].

[14] “Quantum Inspire,” [Online] URL: https://www.quantum-inspire.com [accessed: 2019-03-04].

[15] “Atos,” [Online] URL: https://atos.net/en/insights-and-innovation/quantum-computing/atos-quantum [accessed: 2019-03-04].

[16] “Rigetti,” [Online] URL: https://www.rigetti.com/products [accessed: 2019-03-04].

[17] “IBM,” [Online] URL: https://quantumexperience.ng.bluemix.net/qx/experience [accessed: 2019-03-04].

[18] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, “Open quantum assembly language,” arXiv preprint arXiv:1707.03429, 2017.

[19] Koen Bertels. From qubits to a quantum computer architecture. Intel on the First International Workshop on Quantum Computer Architecture. [Online]. Available: http://www.ce.ewi.tudelft.nl/fileadmin/ce/files/quantum-computer-architecture/QCA_2017_-_Koen_Bertels.pdf

[20] C. G. Almudever et al., “Towards a scalable quantum computer,” in 2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS). IEEE, 2018, pp. 1–1.

[21] P. B. Sousa and R. V. Ramos, “Universal quantum circuit for n-qubit quantum gate: A programmable quantum gate,” arXiv preprint quant-ph/0602174, 2006.

[22] J. J. Vartiainen, M. Möttönen, and M. M. Salomaa, “Efficient decomposition of quantum gates,” Physical review letters, vol. 92, no. 17, 2004, p. 177902.

[23] M. Szyprowski and P. Kerntopf, “Low quantum cost realization of generalized peres and toffoli gates with multiple-control signals,” in Nanotechnology (IEEE-NANO), 2013 13th IEEE Conference on. IEEE, 2013, pp. 802–807.

[24] L. Ruiz-Perez and J. C. Garcia-Escartin, “Quantum arithmetic with the quantum fourier transform,” Quantum Information Processing, vol. 16, no. 6, 2017, p. 152.