Automatic oracle generation in Microsoft’s Quantum Development Kit using QIR and LLVM passes

Invited paper

Mathias Soeken
Microsoft Quantum
Zürich, Switzerland
mathias.soeken@outlook.com

Mariia Mykhailova
Microsoft Quantum
Redmond, WA, USA
mamykhai@microsoft.com

ABSTRACT
Automatic oracle generation techniques can find optimized quantum circuits for classical components in quantum algorithms. However, most implementations of oracle generation techniques require that the classical component is expressed in terms of a conventional logic representation such as logic networks, truth tables, or decision diagrams. We implemented LLVM passes that can automatically generate QIR functions representing classical Q# functions into QIR code implementing such functions quantumly. We are using state-of-the-art logic optimization and oracle generation techniques based on XOR-AND graphs for this purpose. This enables not only a more natural description of the quantum algorithm on a higher level of abstraction, but also enables technology-dependent or application-specific generation of the oracles.

1 INTRODUCTION
Implementing quantum oracles is difficult. Classical Boolean oracles are treated as black boxes in description of algorithms (see, e.g., amplitude amplification \[8, 10\]). While the quantum parts (reflection operator in Grover, QFT in QPE) of the algorithm are described in detail, the oracle is just a placeholder with no implementation or at best an example for a simple classical function. In this paper, we show how to make use of the LLVM [16] infrastructure to create a QIR-based tool that can automatically generate Q# operations [12, 30] for such classical oracles from Q# functions.

With the help of our approach, a developer can write a quantum program as follows:

```
namespace Operations.Classical {
    internal function Majority3(a: Bool, b: Bool, c: Bool): Bool {
        return (a or b) and (a or c) and (b or c);
    }
}
```

1https://qir-alliance.org

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC ‘22, July 10–14, 2022, San Francisco, CA
© 2022 Association for Computing Machinery.

The program contains an internal classical function Majority3 that takes as input 3 Boolean arguments and returns a single Boolean value that is true if and only if the majority of the input arguments is true. The operation Majority3, with the same name but a different namespace, is empty and will be derived using our approach. It can then be automatically used anywhere in the code, e.g., in the Program operation shown in the sample.

In our approach we investigate the case in which the classical input function is defined over tuples of Boolean input arguments and returns tuples of Boolean output values. Since Q# functions are side-effect free, such functions can be represented by combinational Boolean functions \( f : B^n \rightarrow B^m \). We will first derive such a Boolean function from the LLVM code generated for the Q# function. This function is then mapped to a quantum circuit using state-of-the-art logic synthesis based quantum compilation algorithms (see, e.g., [17–20, 22, 28]). Finally, the quantum circuit is mapped to QIR and combined with the LLVM file generated for the rest of the Q# program before linking. This enables a seamless experience, in which the user is relieved from the burden of implementing the classical function as a quantum operation. In addition, the automatic quantum compilation tools employ logic optimization to reduce the implementation cost for the oracles in terms of operation depth and qubit count.

We implemented the proposed approach in C++ and embedded it into a CMake compilation script. The complete implementation is publicly available as part of the Microsoft Quantum Development Kit samples.1

2 RELATED WORK
In [26], the authors have shown how to integrate phase oracle synthesis and permutation synthesis into ProjectQ [29] using RevKit [25]
and how to integrate automatic permutation synthesis in Q# [30]. In [1], the authors describe a quantum programming flow that allows to augment openQASM programs with Verilog code, which is automatically translated into quantum circuits using tweedle-dum [27]. In [6], the authors present the quantum programming language slilq, which maps common arithmetic operations to quantum operations and supports automatic uncomputation.

3 WORKFLOW

In this section, we illustrate the complete workflow of our proposed approach by walking through the running Majority3 example from the introduction. Fig. 1 shows an overview of the overall flow. The Q# file is translated into QIR using the Q# compiler. This file contains an LLVM function definition for the classical Q# function for which an oracle should be generated and an LLVM function declaration (without code body) for the empty operation. A logic network is then created for F, which is further optimized using logic synthesis techniques. More precisely, we will make use of XOR-AND-inverter graphs (XAGs), which are logic networks consisting of binary XOR and binary AND gates, in which signals may be inverted. From the logic network, LLVM code is generated to be used as the code body for 0. The resulting QIR file is compiled into a quantum executable using additional information from a target runtime. All thick arrows and boxes are contributions of our proposed flow on top of the existing QIR compilation flow. Each subsection in the remainder describes one of the arrows in the figure.

3.1 Translating LLVM into a logic network

The initial LLVM code for the Majority3 function in Q# function looks as follows:

```qsharp
define internal i1 @Classical_Majority3(i1 x, i1 y, i1 z) {
  entry:
  %b = or i1 x, i1 y
  br i1 %b, label %condTrue__1, label %condContinue__1
  %c = or i1 x, i1 z
  br label %condContinue__1
  %1 = or i1 x, i1 y, i1 z
  %a = phi i1 [%1, %condTrue__1], [ %b, %entry ]
  br i1 %a, label %condContinue__1
}
```

```c
condContinue__1:
%4 = phi i1 [ %3, %condTrue__2 ], [ %2, %condContinue__1 ]
ret i1 %4
```

We do not cover LLVM’s syntax in detail, but describe few concepts that are important for the remainder of the paper. Every variable, function, and statement is typed. The type i1 describes a 1-bit integer type, which can encode a Boolean value. Variable names are prefixed by a %. Lines such as entry: and condTrue__1: are labels and mark the beginning of a basic block. Each basic block contains a continuous sequence of statements. Statements may produce a value which is assigned to some variable. The basic block entry: marks the first basic block of the function. Examples for LLVM statements are or which computes the Boolean or of two variables and stores it into a result variable, or the br which jumps to some basic block unconditionally.

The translation makes use of static single assignment (SSA) φ-nodes [16], however, their implicitness make an automatic translation into logic networks difficult. We therefore make use of LLVM’s reg2mem transformation pass which introduces explicit memory instructions to explicitly store intermediate variables. Since our approach only parses the transformed result to create a logic network (and not to execute it as is), these instructions will not cause any overhead in memory access in the target program. The transformed function looks as follows:

```c
define internal i1 @Classical_Majority3(i1 x, i1 y, i1 z) {
  entry:
  %breg2mem = alloca i1, align 1
  %creg2mem = alloca i1, align 1
  %bred2mem = alloca i1, align 1
  %rend2mem = alloca i1, align 1
  %condContinue__1_crit_edge: %breg2mem = bitcast i12 0 to i12
  %a = or i1 x, i1 y
  store i1 %a, i1* %breg2mem, align 1
  %reload8 = load i1, i1* %breg2mem, align 1
  br i1 %reload8, label %condTrue__1, label %entry
  %condContinue__1_crit_edge:
  %reload7 = load i1, i1* %re2mem
  store i1 %reload7, i1* %re2mem11, align 1
  br label %condContinue__1
  %condTrue__1:
  %1 = or i1 x, i1 y, i1 z
  store i1 %1, i1* %reg2mem4, align 1
  %reload5 = load i1, i1* %reg2mem4, align 1
  store i1 %reload5, i1* %reg2mem15, align 1
  br label %condContinue__1
  %condContinue__1:
  %reload2 = load i1, i1* %reg2mem11, align 1
  store i1 %reload2, i1* %reg2mem, align 1
  %reload3 = load i1, i1* %reg2mem, align 1
  br i1 %reload3, label %condContinue__2, label %condContinue__1
  %condContinue__2:
  %2 = or i1 x, i1 y, i1 z
  store i1 %2, i1* %reg2mem4, align 1
  %reload = load i1, i1* %reg2mem, align 1
  store i1 %reload, i1* %reg2mem9, align 1
}
```

We do not cover LLVM’s syntax in detail, but describe few concepts that are important for the remainder of the paper. Every variable, function, and statement is typed. The type i1 describes a 1-bit integer type, which can encode a Boolean value. Variable names are prefixed by a %. Lines such as entry: and condTrue__1: are labels and mark the beginning of a basic block. Each basic block contains a continuous sequence of statements. Statements may produce a value which is assigned to some variable. The basic block entry: marks the first basic block of the function. Examples for LLVM statements are or which computes the Boolean or of two variables and stores it into a result variable, or the br which jumps to some basic block unconditionally.

The translation makes use of static single assignment (SSA) φ-nodes [16], however, their implicitness make an automatic translation into logic networks difficult. We therefore make use of LLVM’s reg2mem transformation pass which introduces explicit memory instructions to explicitly store intermediate variables. Since our approach only parses the transformed result to create a logic network (and not to execute it as is), these instructions will not cause any overhead in memory access in the target program. The transformed function looks as follows:

```c
define internal i1 @Classical_Majority3(i1 x, i1 y, i1 z) {
  entry:
  %breg2mem = alloca i1, align 1
  %creg2mem = alloca i1, align 1
  %bred2mem = alloca i1, align 1
  %rend2mem = alloca i1, align 1
  %condContinue__1_crit_edge: %breg2mem = bitcast i12 0 to i12
  %a = or i1 x, i1 y
  store i1 %a, i1* %breg2mem, align 1
  %reload8 = load i1, i1* %breg2mem, align 1
  br i1 %reload8, label %condTrue__1, label %entry
  %condContinue__1_crit_edge:
  %reload7 = load i1, i1* %re2mem
  store i1 %reload7, i1* %re2mem11, align 1
  br label %condContinue__1
  %condTrue__1:
  %1 = or i1 x, i1 y, i1 z
  store i1 %1, i1* %reg2mem4, align 1
  %reload5 = load i1, i1* %reg2mem4, align 1
  store i1 %reload5, i1* %reg2mem15, align 1
  br label %condContinue__1
  %condContinue__1:
  %reload2 = load i1, i1* %reg2mem11, align 1
  store i1 %reload2, i1* %reg2mem, align 1
  %reload3 = load i1, i1* %reg2mem, align 1
  br i1 %reload3, label %condContinue__2, label %condContinue__1
  %condContinue__2:
  %2 = or i1 x, i1 y, i1 z
  store i1 %2, i1* %reg2mem4, align 1
  %reload = load i1, i1* %reg2mem, align 1
  store i1 %reload, i1* %reg2mem9, align 1
}
```
3.2 Optimizing the logic network

Only simple logic optimizations such as constant propagation (e.g., \( I \land x = x \)) or structural hashing [15] (never creating nodes with the same operator and the same operands twice) are applied when creating the initial logic network. We can apply more advanced logic optimization techniques to the resulting network. It is possible to give the developer control over which cost function to assume, or even which sequence of optimization techniques to apply. However, a typical optimization flow would target reducing the number of AND gates in the logic network in favor of XOR gates, since AND gates correspond to more complicated operations both in near-term and error-corrected quantum computing devices [2]. This cost function relates to the multiplicative complexity [23] of Boolean functions. Several algorithms to reduce the number of AND gates in logic networks have been proposed [4, 7, 24, 31, 32]. In our example workflow, we employ the cut rewriting technique described in [31]. When the number of primary inputs does not exceed 8, we first collapse all outputs into their truth tables and decompose them using Shannon’s decomposition rule until all internal logic nodes have 6 inputs, and use the database in [9] to map each node into its optimal XAG representation beforehand. Fig. 3 shows the logic network after optimization. Because the network has a single output and not more than 6 inputs, an optimum representation is found.

3.3 Compiling a logic network into QIR

There exist various techniques that describe how to map logic networks into quantum circuits, such as [17–20, 22, 28]. XAG-based methods are of particular interest due to recent optimization methods for multiplicative complexity and its close connection to resource cost of the quantum implementation [19, 20]. The mapping of an XAG into an LLVM function based on QIR is straightforward. One only needs function calls to allocate and deallocate qubits, as well as calls to CNOT and Toffoli operations. For each AND gate in the logic network, one computes the linear fanin in-place into two existing qubits and then applies a Toffoli gate controlled on these two qubits targeting a helper qubit. Eventually all outputs are copied out using CNOT gates, before all helper qubits are uncomputed by applying all operations in reverse order. For our running example the generated LLVM function looks as follows (note that we shortened some function names for the quantum operations for better readability):

```c
auto Majority3__body(int %a, int %b, int %c) { return %a \& \& %b \& \& %c;
}
```
4 CONCLUSION

In this paper, we described a quantum compilation flow based on Q#, QIR, and LLVM, which creates quantum operations based on classical function implementations. The approach leverages logic networks and recent results in logic optimization based on multiplicative complexity. Many extensions to the proposed flow are possible, e.g., space-efficient mapping of functions defined over integer values, or automatic approximation-aware mapping of functions defined over floating-point values. Further, it is of interest to consider mapping parameterized functions, e.g., which take a dynamic sized array as input, since no straightforward logic network representation exists for such cases.

REFERENCES

[1] Matthew Amy and Vlad Gheorghiu. 2020. stag—a full-stack quantum processing toolkit. Quantum Science and Technology 5, 3 (Jun 2020), 034016. https://doi.org/10.1088/2058-9565/ab9359
[2] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. IEEE Trans. on CAD of Integrated Circuits and Systems 32, 6 (2013), 818–830. https://doi.org/10.1109/TCAD.2013.2244643
[3] Charles Henry Bennett. 1989. Time/Space Trade-Offs for Reversible Computation. SIAM J. Comput. 18, 4 (1989), 766–776. https://doi.org/10.1137/0218053
[4] Anna Bernasconi, Stefano Cimato, Valentina Cipriani, and Maria Chiara Molteni. 2020. Multiplicative Complexity of Autosymmetric Functions: Theory and Applications to Security. In Design Automation Conference. IEEE, 1–6. https://doi.org/10.1109/DAC5072.2020.9218492
[5] Dominic W. Berry, Matthew A. Childs, and Robin Kothari. 2015. Hamiltonian Simulation with Nearly Optimal Dependence on All Parameters. In Foundations of Computer Science. IEEE Computer Society, 792–809. https://doi.org/10.1109/FOCS.2015.54
[6] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin T. Vechev. 2020. Silq: a high-level quantum language with safe uncomputability and intuitive semantics. In ACM SPlASH Conference on Programming Language Design and Implementation. ACM, 286–300. https://doi.org/10.1145/3385412.3386007
[7] Joan Boyar, Philip Matthews, and René Peralta. 2013. Logic Minimization Techniques with Applications to Cryptography. Journal of Cryptology 26, 2 (2013), 280–312. https://doi.org/10.1007/s00145-012-9126-7
[8] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. 2002. Quantum amplitude amplification and estimation. Contemp. Math. 305 (2002), 53–74.
[9] Çağdas Çalık, Méltem Sönmez Turan, and René Peralta. 2019. The multiplicative complexity of &-variable Boolean functions. Cryptography and Communications 11, 1 (2019), 93–107. https://doi.org/10.1007/s12095-018-0297-2
[10] Loy K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In Symposium on Theory and Computing. 212–219. https://doi.org/10.1145/237814.237866