Towards efficient automatic oracle synthesis and resource estimation using QDK and QIR

I-Tung Chen*,
Department of Electrical and Computer Engineering,
University of Washington, Seattle, WA, USA
Email: itungc@uw.edu
* These authors contributed equally

Chaman Gupta*,
Department of Material Science and Engineering,
University of Washington, Seattle, WA, USA
Email: chaman@uw.edu
* These authors contributed equally

***Abstract*—Quantum oracles are often treated as a black box in quantum circuits and require individual tailoring for each problem. Automatic oracle synthesis (AOS) provides a promising route to speed up the process of constructing a complex quantum oracle. Here, we expand the existing AOS in the Microsoft Quantum Development Kit (QDK) by adding new arithmetic operators supports, new input data types, and introducing new workflow that allows one to test and estimate the resource of oracles generated using AOS. We also optimize the AOS process by reducing the number of T gates using measurement-based uncomputation. Furthermore, we compare the resource requirement of the oracles generated using AOS to that of the existing QDK library oracles using Azure Quantum Resource Estimator. The results suggest that the oracles generated using AOS perform better in runtime, but use more qubits compared to the corresponding QDK library oracle. With the presented workflow, one can easily implement and estimate the quantum resource required for oracles with complex arithmetic functions. Finally, we present a strategy that can further reduce the number of physical qubits needed in AOS.**

I. Introductions

Quantum oracles are often elusive and needed to be hand-crafted in different quantum algorithms [1] [2], on the other hand, the implementation of the quantum building blocks following the oracles is usually described in detail (amplitude amplification in Grover's algorithm, quantum Fourier transform in quantum phase estimation). Automatic oracle synthesis (AOS) using Q# [3] is a promising route to implement arbitrary quantum oracles, which is deterministic, side-effect-free, and can be represented by arithmetic functions. In addition, Q# can be compiled into LLVM [4] quantum intermediate representation (QIR) [5], which allows AOS to be performed using logic networks such as XOR-AND-Inverter graphs (XAGs) [6].

Here, we expand the existing AOS from the Microsoft Quantum Development Kit (QDK) [7], [8], [9][8][9] with new arithmetic operators and 64-bit integer input type. Newly supported arithmetic operators include addition, multiplication, comparison, and remainder operators with 64-bit integer input type. The presented work here shows the full workflow of using AOS in the QDK and QIR that can help developers implement complex quantum oracles automatically.

II. Related work

Classiq [10] has demonstrated oracle synthesis and arithmetic oracles. In [11], the authors presented two methods for oracle synthesis, one uses minimum qubits for oracle generation while the second minimize the number of gates. In silq [12], automatic uncomputation is supported. In staq [13], the quantum circuit is synthesized through a QASM language, which is handled by the EPFL Logic Synthesis Libraries [6], same as what is used in this work. This work differs from the other works by presenting a complete workflow that allows one to generate and analysis the oracles with ease in QDK.

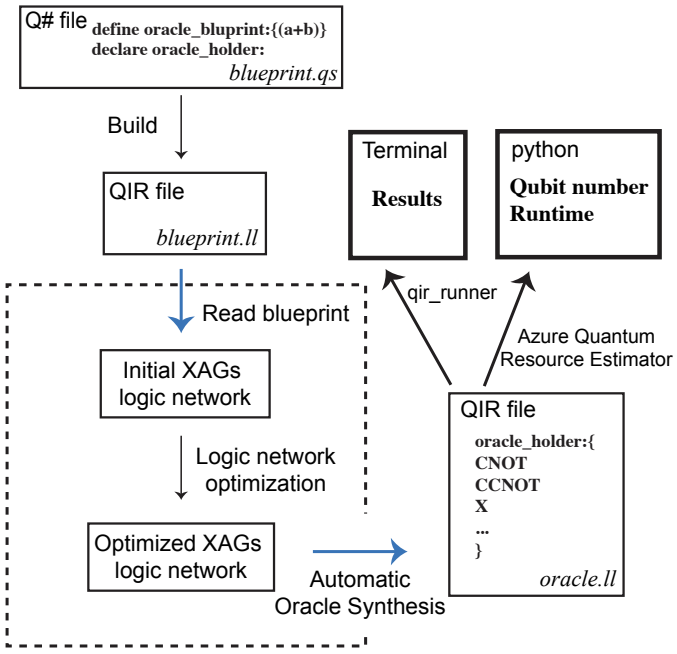

Fig. 1. **Compilation flow for automatic oracle synthesis.**

III. Workflow

The process of AOS [3], as shown in Fig. 1, can be described as the following. First, a desired blueprint is first defined in Q#, which serves as the basis for the quantum oracle implementation, and an empty holder is also declared to hold the forthcoming AOS codes. With the help of our expansion on AOS, one can write a quantum program as follows:

```
namespace OracleGenerator.Classical {
```

```
    internal function AddMultiply(x: Int, y: Int, z:
      Int): Int {
      return (x + y*z);
    }
  }
namespace Operation {
    operation AddMultiply(
      inputs: (Qubit[], Qubit[], Qubit[]),
      output: Qubit[]
    ): Unit {}

    @EntryPoint()
    operation Program(): Unit {
      use (x, y, z) = (Qubit[], Qubit[], Qubit[]);
      use v = Qubit[];
      AddMultiply((x, y, z), v);
    }
  }
```

The program contains an oracle blueprint AddMultiply that takes three integer arguments and returns an integer value that is calculated from the arithmetic functions defined in the Classical namespace. The operation AddMultiply, in the main Operation namespace, is empty and will be generated using AOS. We also compare the resources required for the oracles generated using AOS with similar existing Q# library functions. In addition, a measurement-based uncomputation [14] was incorporated to optimize the AOS process and reduce the quantum resources.

The Q# file is compiled into a QIR file (.ll) and read by the AOS program (the dotted box shown in Fig. 1), which generates and optimizes XAGs networks [15]. Finally, a reversible quantum logic circuit is generated using universal quantum gates (NOT, CNOT, and Toffoli gates) that fill up the holder in the QIR file. The resulted QIR file can be simulated using QIR-runner for verification and loaded by Azure Quantum Resource Estimator [16] for quantum resource estimation.

Our work's main contributions can be identified through the blue arrows, which indicate the extension of existing functions, and the thick arrows, which represent the inclusion of new workflows in AOS, as depicted in Fig. 1.

IV. Case study

In this section, we provide the workflow of executing AddInt oracle and Grover's algorithm as cases study to demonstrate our work. We do not cover the full LLVM (QIR) code in detail, but only describe important concepts in our work.

A. Case I: Arithmetic operations

In this case, the quantum oracle is designed to perform the addition operation, and we illustrate the concept of each step in the AOS. First, we define the blueprint for AOS in Q# as the following:

```
namespace OracleGenerator.Classical {
  internal function AddInt(a : Int, b : Int) : Int {
    return a + b;
  }
}
namespace Operation {
  operation AddInt(
    inputs : (Qubit[], Qubit[]),
    output : Qubit[])
    : Unit {}

  @EntryPoint()
  operation RunProgram() : Unit {
    use (a, b, f) = (Qubit[63], Qubit[63], Qubit
    [63]);

    AddInt((a, b), f);
  }
}
```

The blueprint, AddInt, is defined in the classical namespace and the corresponding empty oracle holder with the same name is also declared in the main program. We can simply call the quantum operation AddInt after the @Entry point, and the quantum operation will be applied when the oracle holder is filled with the AOS oracle.

Next, we read in the QIR file (built from Q) we have defined earlier. This contains the main contribution of our work, which includes the expansion of the supported arithmetic operators and input type. The two pieces of information in the defined blueprint are: the arithmetic operators and the input type, and we can extract them using LLVM, as shown in the below code snippet.

```
bool analyze_function_signature(llvm::Function const
    & f) const
{
    for (const auto& arg : f.args())
    {
        if (arg.getType()->isIntegerTy(64) || arg.
    getType()->isIntegerTy(1u))
        {
            continue;
        }
        return false;
    }
    auto const* retTy = f.getReturnType();
    return retTy->isIntegerTy(64)|| retTy->
    isIntegerTy(1u) || is_valid_tuple_pointer_type(
    retTy);}
```

The QIR reader first extracts the input data types and ensures that they are supported in the AOS. In this case, both 64-bit integer input and Boolean input are supported. Next, the type of operators can be determined using the following code. Here we only show the addition operator as an example.

```
case llvm::Instruction::Add:
{
    auto const* op0 = inst.getOperand(0u);
    auto const* op1 = inst.getOperand(1u);
    auto const* ty0 = op0->getType();
    auto const* ty1 = op1->getType();

    value_signals[&inst] = value_signals[inst.
    getOperand(0u)];

    if (ty0->isIntegerTy(64) && ty1->isIntegerTy(64)
    )
    {
        auto carry = ntk.get_constant(false);
        carry_ripple_adder_inplace(ntk,
    value_signals[&inst], value_signals[inst.
    getOperand(1u)], carry);
    }
```

When the operator is addition, a carry_ripple_adder_inplace is performed between the two consecutive 64-bit integer types.

Following the determination of the input types and the operator types, an initial XAGs logic network is generated, and this initial XAGs network is further optimized using the Mockturtle logic network optimizer [6]. The initial XAGs network is verified using a Verilog simulator to make sure the arithmetic function conversion from QDK to XAGs is correct.

Finally, the quantum oracle is mapped from the optimized XAGs to quantum gates that fill up the oracle holder. In this step, we made another contribution to the AOS workflow, in which we incorporate the measurement-based uncomputation during the conversion from XAGs to quantum gates. Measurement-based uncomputation can effectively reduce the number of Toffoli gates when uncomputing a quantum circuit. In addition to NOT, CNOT, and Toffoli gates in the original AOS workflow, h, measure, and cz instances are added to qir_context to replace the Toffoli gates when uncomputing the quantum circuit, as shown below.

```
h = module.getOrInsertFunction("
    __quantum__qis__h__body", llvm::Type::getVoidTy(
    ctx), qubitPtrTy);
measure = module.getOrInsertFunction(
    "__quantum__qis__m__body", llvm::Type::getVoidTy
    (ctx), qubitPtrTy);
cz = module.getOrInsertFunction("
    __quantum__qis__cz__body", llvm::Type::getVoidTy
    (ctx), qubitPtrTy, qubitPtrTy);
```

Then, when the AOS is uncomputing the circuit, we performed the measurement-based uncomputation instead of using Toffoli gates, as shown in the following:

```
if (compute == 1)
{
    builder.CreateCall(qir.CCNOT(), {node_to_value[
    diff[0].front()], node_to_value[diff[1].front()
    ], temporary});
}

if (compute == 0)
{
    builder.CreateCall(qir.H(), {node_to_value[diff
    [0].front()]});
    builder.CreateCall(qir.Measure(), {node_to_value
    [diff[0].front()]});
    builder.CreateCall(qir.CZ(), {node_to_value[diff
    [0].front()], node_to_value[diff[1].front()]});
}
```

The oracle generated using AOS is saved in the oracle holder block in the QIR file and is ready to be used. And its output can be simulated and verified using the QIR runner, which is a new workflow presented in this work.

B. Case study II: Grover's algorithm integrating with AOS

Now, we apply the exact same workflow as before, but we implement an oracle for Grover's algorithm to search for a missing ISBN digit of a book. This case study is referenced from [17]. Here, the oracle uses the $(x + a \cdot y) \mod N$ operation to determine which digit (0 to 9) satisfies the ISBN rule, where a and N are constant inputs, and x and y are qubit registers. We can simply define the oracle blueprint as

```
namespace OracleGenerator.Classical {
  internal function addmod(x: Int, y: Int): Int {
    let (a, N) = (6, 11);
    return ((x + ((a*y)%N))%N);
  }
}
```

We can use this oracle to search for the missing ISBN digit and the resulting output is shown below:

```
ISBN with missing digit: [0, 3, 0, 6, -1, 0, 6, 1,
    5, 2]
Equation: (9 + 6y) mod 11 = 0, Missing digit: 4
Full ISBN: [0, 3, 0, 6, 4, 0, 6, 1, 5, 2]
The missing digit was found in 1 attempts.
```

where –1 represents the missing digit in the ISBN, and we can see the missing digit is found in 1 attempt.

With the new capabilities of new input types and operators, any discrete function that can be encoded in integers and implemented using AOS. With these enhancements, the process of generating oracles using the expanded AOS becomes streamlined, optimized, and user-friendly. This simplification empowers developers to implement complex arithmetic functions without requiring in-depth knowledge of the library functions that map such arithmetic functions to qubit registers.

V. AOS Resource Estimation

In this section, we compare the resource needed for the oracles generated using AOS with similar Q# library oracles using Azure Quantum Resource Estimator, as shown in the table below. The library oracles used here are AddI, MultiplyI, and MultiplyAndAddByModularInteger, which corresponds to AOS implementations x+y, x∗y, and (x+a∗y) mod N, respectively. Here, x and y represent qubit registers inputs while a and N are constants, which do not count as inputs. We include the constants to make the oracle generated by AOS has the closest form to the library oracle. The only difference between the two is that the library oracles use in-place computation, where the output value is stored in one of the input registers, while the oracles generated using AOS need an extra register to store it. The resource estimation result is shown in Fig. 2. We estimate the resources on six different qubit parameters [16]: us_e3(e4), ns_e3(e4), and Maj_e4(e6), which have the operational runtime and fidelity that may correspond to future ion-based qubits, superconducting qubits, and Majorana qubits, respectively.

As shown in Fig. 2, the library oracles generally require fewer qubits, but a longer runtime compared to oracles generated using AOS, which can be explained by the in-place computation in library oracles. We also see a reduction in the runtime between the optimized AOS (green bar) and the original AOS (orange bar), which is because of the measurement-based uncomputation reduces the number of T gates. We also observe a trade-off between the runtime and number of qubits for different oracles, as shown in Fig. 2(a), (c), (e). Finally, we also estimated the resources required to run Grover's algorithm (ISBN digit search) using oracles generated by AOS and compare that to the library oracle, the result shows AOS method uses more qubits but significantly shorter runtime, as shown in Table I. Note that we used the ns_e3 qubit parameter here.

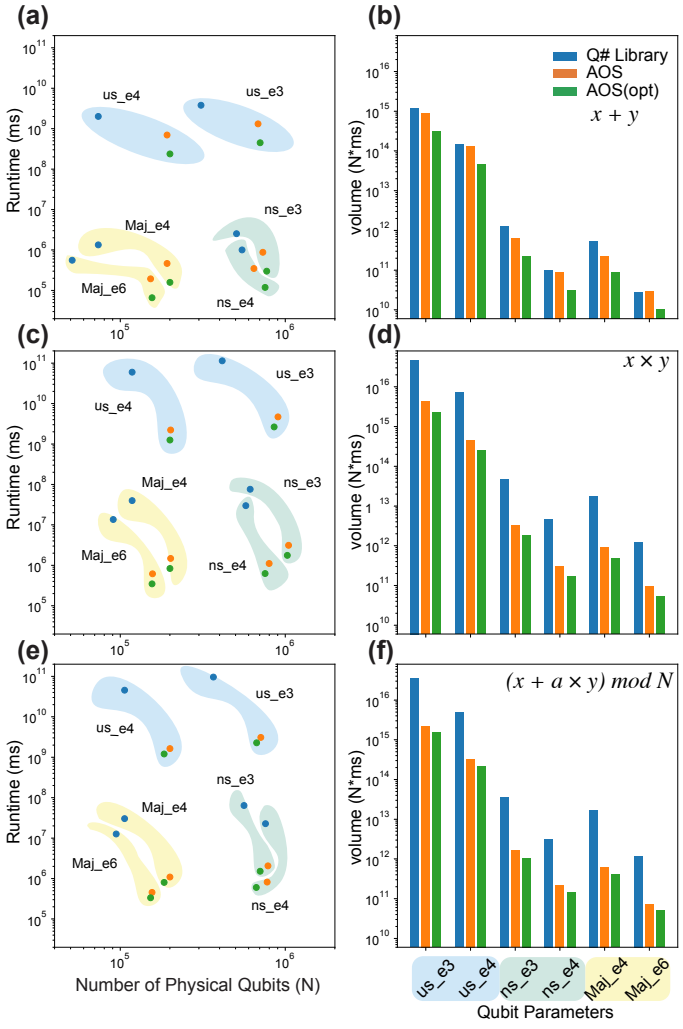

Fig. 2. **Resources required for library functions in Q# compared with their respective AOS functions. (a)(c)(e)** The quantum resource needed for $x + y$, $x \times y$, and $(x + a \times y)\ mod\ N$ arithmetic functions, respectively. Error budgets used here are $1e-5$, $1e-5$, and $1e-4$ for cases (a), (c), and (e), respectively. **(b)(d)(f)** The quantum volume comparison of $x + y$, $x \times y$, and $(x + a \times y)\ mod\ N$ arithmetic functions, respectively.

TABLE I
GROVER'S ALGORITHM RESOURCE ESTIMATION

Algorithm	Physical Qubits	Total Runtime
AOS oracle	621,740	4ms 752us
Library oracle	572,850	77ms 646us

VI. Conclusions and Outlook

In summary, we extended the AOS support of 64-bit integer and integer arithmetic operations. We also detail the workflow of AOS using QDK, through which the readers can easily follow. In addition, we incorporate Azure Quantum Resource Estimator to estimate the resource needed of the oracles generated using AOS. The results show a trade-off between the number of qubits and the runtime between library oracles and oracles generated using AOS, even for complex composite functions, oracles generated using AOS lower the runtime significantly. The presented workflow also allows the developers to speed up the process of constructing and testing new quantum oracles. The path forward is to include the support of floating point and fixed point data types and incorporate the in-place computation to reduce the number of qubits required for the AOS. The source code and the jupyter notebook used in this project can be found here [18].

VII. Acknowledgements

We thank Mariia Mykhailova and Dr. Mathias Soeken from Microsoft Quantum for mentoring this project, the fruitful weekly discussions, and the detailed feedback on the drafting of this work. We also thank Dr. Sara Mouradian, QuantumX, and AQET at UW for the opportunity to work on this project.

References

[1] D. W. Berry, A. M. Childs, and R. Kothari, "Hamiltonian simulation with nearly optimal dependence on all parameters," in *2015 IEEE 56th annual symposium on foundations of computer science*, pp. 792–809, IEEE, 2015.
[2] S. P. Jordan, "Fast quantum algorithm for numerical gradient estimation," *Physical review letters*, vol. 95, no. 5, p. 050501, 2005.
[3] M. Soeken and M. Mykhailova, "Automatic oracle generation in microsoft's quantum development kit using qir and llvm passes," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 1363–1366, 2022.
[4] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.
[5] QIR Alliance, *QIR Specification*, 2021. Also see https://qir-alliance.org.
[6] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage, *et al.*, "The epfl logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2018.
[7] I. L. Markov and M. Saeedi, "Constant-optimized quantum circuits for modular multiplication and exponentiation," 2015.
[8] I. L. Markov and M. Saeedi, "Faster quantum number factoring via circuit synthesis," *Phys. Rev. A*, vol. 87, p. 012310, Jan 2013.
[9] G. Meuli, M. Soeken, E. Campbell, M. Roetteler, and G. D. Micheli, "The role of multiplicative complexity in compiling low t-count oracle circuits," 2019.
[10] Classiq, "Classiq arithmetic oracle, https://docs.classiq.io/0-13/user-guide/builtin-functions/arithmetic/arithmetic-oracle.html."
[11] J. M. Henderson, E. R. Henderson, A. Sinha, M. A. Thornton, and D. M. Miller, "Automated quantum oracle synthesis with a minimal number of qubits," 2023.
[12] B. Bichsel, M. Baader, T. Gehr, and M. Vechev, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, (New York, NY, USA), p. 286–300, Association for Computing Machinery, 2020.
[13] M. Amy and V. Gheorghiu, "ttstaq/tt—a full-stack quantum processing toolkit," *Quantum Science and Technology*, vol. 5, p. 034016, jun 2020.
[14] C. Gidney, "Halving the cost of quantum addition," *Quantum*, vol. 2, p. 74, jun 2018.
[15] G. Meuli, M. Soeken, and G. De Micheli, "Xor-and-inverter graphs for quantum compilation," *npj Quantum Information*, vol. 8, no. 1, p. 7, 2022.
[16] M. E. Beverland, P. Murali, M. Troyer, K. M. Svore, T. Hoefler, V. Kliuchnikov, G. H. Low, M. Soeken, A. Sundaram, and A. Vaschillo, "Assessing requirements to scale to practical quantum advantage," 2022.
[17] Microsoft, "Use the Q# libraries."
[18] "GitHub repository of this project, https://github.com/itungc/automatic-oracle-synthesis-using-qdk-and-qir.git,"