Special Session Paper

HiQ-ProjectQ: Towards user-friendly and high-performance quantum computing on GPUs

1st Damien Nguyen
Zurich Research Center
Data Center Technology Laboratory
2012 Laboratories, Huawei
Zurich, Switzerland
damien1@huawei.com

2nd Dmitry Mikushin
Zurich Research Center
Data Center Technology Laboratory
2012 Laboratories, Huawei
Zurich, Switzerland
dmitry.mikushin@huawei.com

3rd Yung Man-Hong
Central Research Institute
Data Center Technology Laboratory
2012 Laboratories, Huawei
Shenzhen, China
yung.manhong@huawei.com

***Abstract*—In this work, we present some of the latest efforts made at Huawei Research in order to improve the overall performance of ProjectQ, the quantum computing framework used as the foundation of our quantum research. Since a few years, performance assessment of the framework using profiling tools has shown that a significant portion of the compilation time is spent in doing memory management linked to the lifetime and access of Python objects. The main purpose of this work is therefore aimed at addressing some of these limitations by introducing a new C++ processing backend for ProjectQ, as well as starting a complete rewrite of the simulator code already written in C++.**

The core of this work is centered around on a new C++ API for ProjectQ that moves most of the compiler processing into natively compiled code. We achieve this by providing a new compiler engine to perform the conversion from Python to C++, which ensures that we retain maximum compatibility with existing user code while providing significant speed-ups. We also introduce some of our work aimed at porting the existing C++ code to offload the more demanding calculations onto GPUs for better performance.

We then investigate the performance of this new API by comparing it with the original ProjectQ implementation, as well as some other existing quantum computing frameworks. The preliminary results show that the C++ API is able to considerably reduce the cost of compilation, to the point that the compilation process becomes mostly limited by the performance of the simulator.

Unfortunately, due to the some last minutes developments and the time frame required for the submission of the present manuscript, we are unable to provide conclusive benchmark results for the GPU-capable implementation of the simulator. These will most likely be presented during the keynote presentation at the DATE conference in 2021.

***Index Terms*—quantum computing, compilation, projectq, hiq**

I. Introduction

Quantum computers promise to offer significant advantages in solving a variety of problems that are demonstrably hard for classical computers [1]–[4]. While for some classes of problems, e.g. in the field of condensed-matter physics, the mapping between the algorithms and the intrinsic structure of the problem is relatively straightforward to generate, it is generally not the case. Indeed, the languages used to design and describe both classical and quantum algorithms are usually aimed at ensuring correctness or facilitating the derivation of some complexity estimates rather than caring about the actual implementation of the algorithm on some particular hardware. It is up to quantum computing software frameworks to translate the *human-readable* language into efficient *machine* code to run on simulators or actual quantum computing hardware.

Over the years, the most common approach has been to embed the quantum-specific extensions into a general-purpose programming language. This makes adoption from users already familiar with these languages easier, as well as facilitating development of the quantum computing software framework by relying on existing interpreters or compilers. A large number of quantum front-ends are based on Python [5]–[8] due to its ease of use and prevalence in the scientific community worldwide. Yet some have been built on top of other languages, such as C++ [8], [9], Julia [10] or C#/F# [11]. In this work, we will focus on ProjectQ [12], a quantum computing software framework which embeds a quantum computing language in Python. ProjectQ was designed to have a device independent high-level description of quantum algorithms with an intuitive syntax and a modular compiler design [13], coupled with a highly optimised full amplitude simulator written in C++ for better performance.

The purpose of the present paper is to discuss some of the latest developments at Huawei Research that are aimed at improving the overall performance of ProjectQ during the compilation and optimisation of a quantum circuit. This is mainly achieved by introducing a new C++ compilation backend that guarantees almost total compatibility with existing user code. We also introduce some preliminary work on improving the C++ simulator performance by adding support for GPU computations using the Nvidia CUDA programming framework.

A. Current state of ProjectQ

As it stands today, ProjectQ is mostly developed in Python with the addition of a high-performance C++ simulator with emulation capabilities. While this offers great flexibility in development and facilitates its adoption by users, it comes with some associated costs. Most notably, Python, being an interpreted language, has some inherent performance penalties when compared to similar code written in other languages that are natively compiled, e.g. in C++. Moreover, the nature of Python's memory and object management prevents important optimizations opportunities, such as is the case with constant folding for example. In ProjectQ, these performance penalties

978-3-9819263-5-4/DATE21/©2021 EDAA

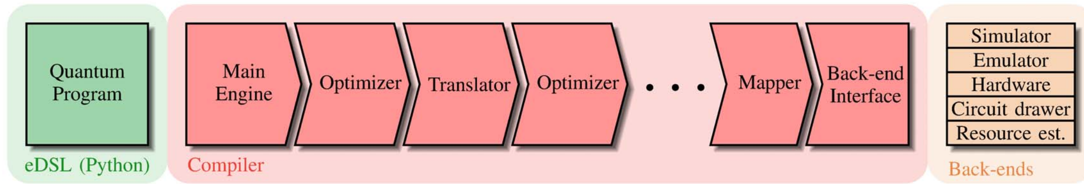

Fig. 1. Typical ProjectQ workflow. A quantum program (in green) gets converted into a list of commands that get processed by a number of compiler engines (in red), before reaching a backend (in yellow) for final processing. Image reproduced from [14] with permissions from its authors.

can particularly be felt when profiling real-world-like example quantum circuits. For compute-bound problems, while we expect majority of time to be spent within the simulator, a significant amount of time is spent in managing memory object to guarantee that all pre- and post-conditions are followed at each step of the compilation process.

Contrary to many of its competitors, ProjectQ has not adopted a *quantum circuit as an object* approach to compiling quantum circuit. In this approach, quantum circuit compilation (also called *transpilation* by some software frameworks [5]) is mainly performed by first building a quantum circuit object and then applying some transformation passes to it. This approach is particularly suited for today's Noisy-Intermediate-Scale Quantum (NISQ) devices, where resources are typically limited and circuits are relatively short with little conditional branching. Experiments on NISQ devices tend to be short with qubit allocation taking place at the start of the experiments with measurements performed at the end, although that is slowly changing as devices become more capable.

ProjectQ, on the other hand, views a quantum circuit as a series of classical and quantum instructions that flow between a set of compiler engines that can perform certain actions on them (Fig. 1), such as gate decomposition, qubit mapping or optimising gates. Customisation of the compiler and optimiser procedure is done by dynamically modifying the list of compiler engines before or during the compilation process itself. This has allowed some flexibility in building the quantum circuit, such as providing an easy syntax for the *Compute/Uncompute* pattern for example. Another key difference to the *quantum circuit as an object* approach is that qubit allocation, deallocation and measurement may occur at any point within the program, which gives the user greater control over the management of the qubit resources.

The current ProjectQ already provides some limited C++ interface in the form of a high-performance simulator. It was developed to be run exclusively on CPUs, heavily relying on the AVX2 compiler intrinsics instructions set where possible. Additionally, the calculations were optimised to make use of fused multiply-accumulate instructions by storing the real and imaginary part of complex numbers in the same AVX registers in order to increase throughput performance [15], particularly in cases for matrix operations that require both the matrix itself and its hermitian form.

B. Current state of GPU Computing

General-purpose GPU computing has evolved rapidly for over a decade. The modern solutions offer various integer and floating-point compute modes, potentially prospective for sustained accuracy conservation. For the scope of this work, however, we will focus on classical double precision computations. The latest NVIDIA Tesla A100 GPU has double-precision peak compute throughput of $9.7\,\mathrm{TFLOPS}$ [16] (Fig. 2).

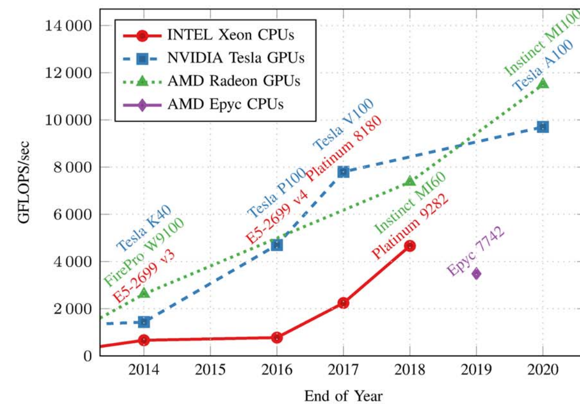

Fig. 2. Comparison of double-precision computing performance (data taken from [16])

High-end GPUs are most efficient on extremely massive and massively parallel workloads. For instance, an NVIDIA V100 can sustain up to 1344 persistent blocks of 128 threads with 32 registers per thread. Although nested parallelism and streaming are supported in modern GPU software, the best scheduling is usually achieved for by uniformly distributing the workload onto the GPU compute grid. This is a situation typically encountered in full amplitude quantum simulators when applying a gate to some qubits as multiple independent matrix vector multiplications are required.

II. HiQ-ProjectQ: a new take on ProjectQ

HiQ-ProjectQ is the name of the internal fork of ProjectQ currently being worked on at Huawei Research. It includes some additional features compared to the original ProjectQ code, some of which are available on its GitHub page [17], while others are in the process of being released open-source

and will follow shortly. Note that this includes all the new material that will be presented here at the time of this writing.

A. Moving to C++

One of the goal while developing HiQ-ProjectQ is to improve the performance of the quantum compiler itself, as it is the foundation onto which most of the other quantum projects build upon. We plan on achieving this by moving as much of the compilation process in C++ as feasibly possible, while keeping the Python interface mainly as a front-end with only the bare of essentials. This is inline with the direction in which most software packages for quantum computing seem to go into; e.g. Qulacs [8]. Ideally, we would do all the processing in C++ or some other natively compiled language to get over constraints imposed by the Python interpreter. While this is desirable long-term it is outside of the scope of the present work. Nonetheless, work on the C++ API does lay down some of the ground work for future developments that will aim to go in that direction.

Due to the modular structure of the ProjectQ compiler (Fig. 1), writing the new C++ processing backend mainly implies creating a new compiler engine to convert ProjectQ commands and compiler engines objects from Python to C++. Once the conversion has occurred, the rest of the compilation, optimisation, mapping and simulation processes are run in C++. In practice, we also implemented a new main engine, which is the main point of entry in any ProjectQ program, in order to make the new API easier to use by current ProjectQ users. Additionally, we need to adapt some of the backends in order to ensure maximum compatibility with existing code.

With these changes in place, most ProjectQ programs today only need the replacement of any occurrence of `MainEngine` with `CppMainEngine` in order to use the new C++ API. In particular, all the basic gates, and some more advanced gates such as `TimeEvolution` and `QubitOperator`, and compiler engines currently included in ProjectQ are supported by the new C++ API without requiring any changes in user code. There are a few limitations due to the Python to C++ conversion process that we hope to address in the future, such as supporting user-defined gate classes or decomposition rules, and perhaps even compiler engines using either just-in-time compilation or possibly through other means. Furthermore, support for submitting ProjectQ quantum circuits to web-based backends (such as `IBMBackend` or `AQTBackend`) is currently not included, although that could be easily remedied by either also porting that code to C++ or by providing a C++ to Python conversion engine.

Through the compiler engine list that is passed as an argument to the main engine, the user is able to customise the exact point at which the Python to C++ conversion will occur by specifying where the `Python2CppEngine` is located. This also serves as a marker for HiQ-ProjectQ to know which compiler engines need to be emulated in C++, which includes the backend. This also ensures that the new API is capable of running any current ProjectQ object, even with custom user-defined gates or compiler engines, provided that the relevant engines or gate decompositions occur before the Python to C++ conversion.

For the C++ part of the compilation process, our current implementation is based on the Tweedledum quantum circuit library [18]. While this library is still in active development and not fully stable yet, a new beta after a complete rewrite has recently been published. Though incomplete, it already provides more than the basic functionality required for our needs, which include the processing and compiling quantum circuits such as creating quantum circuit, supporting custom gates, various optimisation passes as well as qubit mapping and circuit synthesising algorithms. However, since Tweedledum and ProjectQ have different quantum circuit representations, a considerable part of the new API is aimed at handling qubit allocations and deallocations, as well as handling the mapping between qubits identifiers between Pytho and C++ in order to ensure that gate operations get applied on the correct qubit and that measurement results are properly propagated. We also ensure that the compilation process is only performed once on each gate instruction. In the current version of the C++ API, this was achieved by a new class designed to manage the internal quantum circuit representation.

All in all, our efforts aim to minimize disruptions to user code, while maximizing the performance gains during compilation and simulation.

B. Quantum simulations on GPUs

The main computing task of the simulator is to represent the application of a quantum gate to some qubits, which typically takes the form of a computing kernel comprising of multiple matrix-vector multiplications for full amplitude simulations. The parallelisation of the current qubit processing kernels has been implemented in a relatively straightforward way to work on multicore CPUs, namely using a list of multidimensional nested loops which are then collapsed using OpenMP directives (Fig. 3 top). In order to port the simulator code for GPU computations in an optimal fashion, most of the work has been aimed at replacing the nested loop structure with a single loop with identity stepping of the loop index, effectively achieving the same result as the collapse OpenMP directives. In order to optimise for performance, we also make sure that the index type is chosen so as to be the minimum required to span all the ranges required by the calculation (Fig. 3).

In practice, kernel loop iterations operate on a small input square matrix `m`, a small input vector `d` and a large output vector `psi` which represent the wavefunction. In place of an explicit for-loop construct, we decided on using a standard STL *for_each* construct iterating over an artificial *counting iterator* similar to the ones existing in the Boost and Thrust libraries, which provides the functor with an integer index to be used for accessing array elements. Starting with C++17, most standard STL algorithms accept an additional parameter to support parallelism in the form of an execution policy, which is for our particular case the C++ parallel STL `std::par_unseq` policy as all kernel iterations are effectively independent of one another and can be run in any order.

This approach has the advantage of unifying the parallel loop implementation across multiple targets with or without multi-threading support enabled. In addition, NVIDIA recently

```
...
if (ctrlmask == 0){
    #pragma omp for collapse(6) schedule(static)
    for (UINT i0 = 0; i0 < n; i0 += 2 * ds[0])
        for (UINT i1 = 0; i1 < ds[0]; i1 += 2 * ds[1])
            for (UINT i2 = 0; i2 < ds[1]; i2 += 2 * ds[2])
                for (UINT i3 = 0; i3 < ds[2]; i3 += 2 * ds[3])
                    for (UINT i4 = 0; i4 < ds[3]; i4 += 2 * ds[4])
                        for (UINT i5 = 0; i5 < ds[4]; ++i5)
                            K::core(...);
}
...
```

```
inline void kernel_loop(...) {
    const BITMASK upperBound =
        (CHAR_BIT * sizeof(BITMASK) == maskBitSize) ?
        ~BITMASK(0) : (BITMASK(1) << maskBitSize) - 1;

    BITMASK bitMask[N + 1];
    for (int jj = 0; jj < N + 1; jj++)
        bitMask[jj] = BITMASK((1 << bitSize[jj]) - 1);

    #pragma omp for
    for (BITMASK ii = 0; ii < upperBound; ii++)
        kernel_body<...>(ii, bitMask, bitSize,
        psi, m, ctrlmask, d, ds, de);

    // Run the last iteration separately to avoid
    // the possible index type overflow.
    kernel_body<...>(upperBound, ...);
}
```

```
inline void kernel_body(...) {
    BITMASK bits = ii;
    UINT i = 0;
    for (int jj = 0; jj < N; jj++)
    {
        i += (bits & bitMask[jj]) * ds[jj];
        bits >>= bitSize[jj]; // popcnt(bitMask[jj])
    }
    i += bits & bitMask[N];

    if constexpr(CTRLMASK == 0)
        K::core(psi, i, d, m);
    else
        if ((i & ctrlmask) == ctrlmask)
            K::core(psi, i, d, m);
}
```

Fig. 3. Comparison of the original ProjectQ multidimensinal loop nest (multiloop) code *(top)* and linearized loop with packed multiindex extraction *(bottom)*. To simplify the present figure, the approach based on the STL `std::for_each` algorithm discussed in the text is not represented here and replaced with a traditional for-loop.

introduced the *nvc++* compiler as part of its HPC SDK. This new compiler takes advantage of the freedom given to compiler and library developers to implement parallel STL algorithms that are using an execution policy in order to generate device-specific code (in the case of GPUs), as well as automatically handling any memory transfers between CPU and GPU memory. This can happen without requiring any `__device__` annotations or special markings for the GPU code, which practically means that a single code base can be run on a variety of platforms and targets with little to no cost to the user.

In our case, the parallelism support is provided by the GCC compiler, the NVIDIA *nvc++* compiler, and the Intel oneAPI *DPC++* compiler for multicore CPUs, NVIDIA GPUs and Intel GPUs, respectively. Moreover, OpenCL devices could be supported by means of the SYCL Parallel STL [19]. The GCC implementation is backed by the Thread Building Blocks (TBB) library. TBB provides out-of-order multithreaded iterator with workload balancing for x86 and ARM-based platforms. The *nvc++* compiler is provided by the novel NVIDIA HPC SDK, a successor of PGI OpenACC compiler. Besides supporting OpenACC, it has been extended to provide parallel STL and partial OpenMP 5 support. In turn, the parallel STL implementation of *nvc++* is based on the Thrust backend, a GPU-specific variant of STL. Intel oneAPI suite internally combines SYCL with GPU/FPGA execution policy. Embedded GPUs (e.g. ARM Mali) could be covered by using some generic SYCL implementations, such as Codeplay ComputeCPP combined with a vendor-specific runtime.

Some targets can additionally leverage vectorisation within a kernel loop iteration. Specifically, AVX and Advanced SIMD instructions can be used to vectorise `psi` computation on Intel and ARM targets, respectively. Unlike CPU targets, GPU targets generaly operate on scalar values, with an exception of NVIDIA. NVIDIA GPU stands out by supporting 128-bit vectorised memory load/stores, which could be used to reduce the instruction pressure. We intend to unify all variants of vector support by means of the XSIMD framework [20].

Package	Version
qiskit	v0.23.1
qiskit-aer	v0.7.1
qiskit-terra	v0.16.1
Qulacs-GPU	v0.2.0
ProjectQ	v0.5.1

TABLE I
VERSION OF PACKAGES USED IN THE PRESENTED BENCHMARKS

III. Results & Discussion

In this section, we aim to show the performance of our new implementations compared to the existing one, as well as show some comparisons with some existing software. The benchmarks were run on a Linux desktop computer equipped with an Intel Xeon 6238T with 192 GiB of DDR4 memory and an Nvidia Quadro GV100 graphics card. The benchmarks were compiled using GCC 10.2.0 with multithreading enabled using OpenMP, CUDA 11.1 for GPU-enabled packages and run using the pytest-benchmark package under Python 3.8.6. Each package was installed inside its own virtual environment

to ensure that the tests can be run in isolation and avoid outside influences. The versions of the software packages used to run the benchmarks are summarised inside Tab. I. All software were compiled with multi-threading and GPU support where applicable and the benchmarks run with the number of threads capped at the physical number of cores present on the machine. All the benchmarks were repeated at least 5 times in order to gather some statistics and unless stated otherwise, the data presented thereafter is the minimum time required to execute each run.

Unfortunately, due to the some last minutes developments and the time frame required for the submission of the present manuscript, we are unable to provide conclusive benchmark results for the GPU-capable implementation of the simulator. These will most likely be presented during the keynote presentation at the DATE conference in 2021.

The main benchmark consists in generating a random circuit comprised of random single-qubit Pauli-X rotations and controlled-X gates as described in Alg. 1. For each software package, we ran the benchmark for up to 25 qubits (Fig.4) with the number of repetitions M was set to 100 for all experiments.

Algorithm 1: Random circuit algorithm

```
input : N qubits, an integer M
// Apply random Pauli-X rotations
foreach qubit do
 | Rx(random_angle(), qubit)
end
for i ← 0 to M do
 | // Apply pairwise CNOT gates
 | for i ← 0 to N do
 | | CNOT(qubit [i], qubit [(i+1)%N]);
 | end
 | // Apply random Pauli-X rotations
 | foreach qubit do
 | | Rx(random_angle(), qubit)
 | end
end
```

The results clearly show that a significant overhead is incurred by ProjectQ when compared to HiQ-ProjectQ, as we seem to gain about an order of magnitude in speedup for up to about 16 qubits. As both frontends are still Python-based, this gain in performance mainly comes from less overhead in running natively compiled code versus interpreted code. Note that some of the improvements also come from some improved versions of the compilations algorithms used in the C++ version, e.g. with circuit optimisation. Also, since both implementations use the same C++ simulator, the speedup diminishes when the number of qubits increases as the simulation cost dominates the overall compilation process.

When comparing with other frameworks such as Qulacs, it is important to note that the current implementation of the ProjectQ C++ simulator operates on gate matrices directly and does not possess optimised compute kernels for each basic gate type (e.g. Pauli-X, Hadamard, etc.). This can likely account for some of the differences observed in the performance data. The implementation of such changes is also something we are considering for future works.

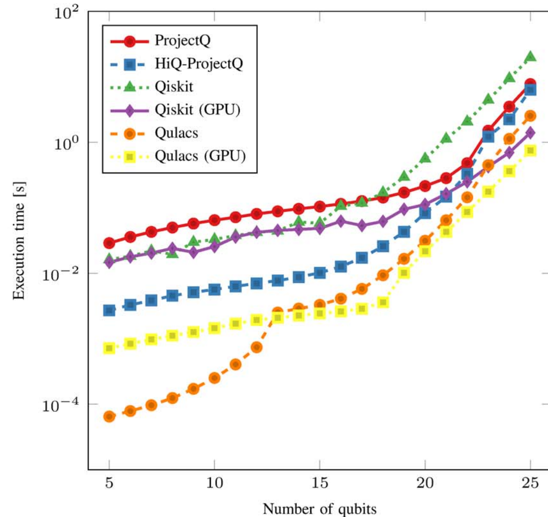

Fig. 4. Random circuit (Alg. 1) benchmark results with various software packages (Tab. I).

The second set of benchmarks that we have run consists in applying the Grover search algorithm with an alternating bit oracle with up to 21 qubits (Fig. 5) in order to showcase the impact of our new implementations in some more real-world conditions. Similarly to the random circuit case, HiQ-ProjectQ consistently outperforms the original ProjectQ in all of our runs. This is again linked to faster processing and better optimisation algorithms when using the C++ implementation.

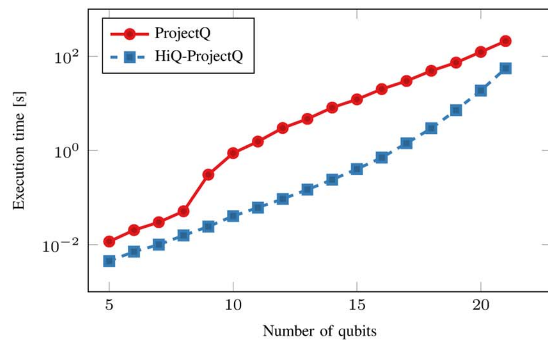

Fig. 5. Executing the Grover search algorithm for an alternating bit oracle.

IV. Conclusion

In this paper, we presented some of the latest developments at Huawei that are aimed at improving the overall performance of our Huawei's quantum computing software framework based

on ProjectQ. Our new implementation of ProjectQ, named HiQ-ProjectQ, shows that at least an order of magnitude in the compilation duration can be gained when using our new C++ implementation with little to no changes in existing user code. Furthermore, we have started a complete rewriting of the simulator to allow for both CPU and GPU execution backend in a single code base without compromising on performance. Although this will require investigations that are not finished at the time of this writing, future work will be aimed at continuously improving the C++ simulator, as well as integrating the C++ HiQ-ProjectQ with other simulation paradigms such as single-amplitude simulation based on tensor contractions [21], [22].

Acknowledgment

We would like to thank our colleagues at Huawei for helping review and edit this manuscript, in particular Romain Moyard who joined us recently for his insights.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, p. 1484–1509, Oct. 1997. [Online]. Available: https://doi.org/10.1137/S0097539795293172

[2] L. K. Grover, "Quantum computers can search arbitrarily large databases by a single query," *Phys. Rev. Lett.*, vol. 79, pp. 4709–4712, Dec 1997. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.79.4709

[3] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Phys. Rev. Lett.*, vol. 103, p. 150502, Oct 2009. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.103.150502

[4] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," 2014.

[5] "Qiskit aqua." [Online]. Available: https://qiskit.org/aqua/

[6] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "t—ket⟩: a retargetable compiler for nisq devices," *Quantum Science and Technology*, vol. 6, no. 1, p. 014003, Nov 2020. [Online]. Available: http://dx.doi.org/10.1088/2058-9565/ab8e92

[7] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, M. S. Alam, S. Ahmed, J. M. Arrazola, C. Blank, A. Delgado, S. Jahangiri, K. McKiernan, J. J. Meyer, Z. Niu, A. Száva, and N. Killoran, "Pennylane: Automatic differentiation of hybrid quantum-classical computations," 2020.

[8] "Qulacs," 2018.

[9] T. Nguyen, A. Santana, T. Kharazi, D. Claudino, H. Finkel, and A. McCaskey, "Extending c++ for heterogeneous quantum-classical computing," 2020.

[10] X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, "Yao.jl: Extensible, efficient framework for quantum algorithm design," *arXiv:1912.10877*, 2019.

[11] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q#: Enabling scalable quantum computing and development with a high-level dsl," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, ser. RWDSL2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3183895.3183901

[12] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: an open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, Jan. 2018. [Online]. Available: https://doi.org/10.22331/q-2018-01-31-49

[13] T. Häner, D. S. Steiger, K. Svore, and M. Troyer, "A software methodology for compiling quantum programs," *Quantum Science and Technology*, vol. 3, no. 2, p. 020501, feb 2018. [Online]. Available: https://doi.org/10.1088/2058-9565/aaa5cc

[14] Steiger, Damian and Häner, Thomas and Troyer Matthias, "ProjectQ: Software for quantum computing," Presentation at the Quantum Systems and Technology conference at Monte Verità, 5 2018.

[15] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126947

[16] K. Rupp, "CPU, GPU and MIC hardware characteristics over time," 2013. [Online]. Available: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

[17] Huawei-HiQ, "HiQ-ProjectQ." [Online]. Available: https://github.com/HuaweiHiQ/ProjectQ

[18] B. Schmitt, "Tweedledum." [Online]. Available: https://github.com/boschmitt/tweedledum

[19] K. Group, "Sycl parallel STL," 2019. [Online]. Available: https://github.com/KhronosGroup/SyclParallelSTL

[20] J. M. et al., "Xsimd: C++ wrappers for simd intrinsics," 2020. [Online]. Available: https://github.com/xtensor-stack/xsimd.git

[21] J. Gray, "quimb: a python library for quantum information and many-body calculations," *Journal of Open Source Software*, vol. 3, no. 29, p. 819, 2018.

[22] J. Gray and S. Kourtis, "Hyper-optimized tensor network contraction," 2020.