A pragma based C++ framework for hybrid quantum/classical computation

Arnaud GAZDA[1] and Océane KOSKA[1,2]

[1]Eviden Quantum Lab, Les Clayes-sous-Bois, France
[2]Laboratoire Interdisciplinaire des Sciences Numériques,
Gif-sur-Yvette, Saclay, France

March 29, 2024

Abstract

Quantum computers promise exponential speed ups over classical computers for various tasks. This emerging technology is expected to have its first huge impact in High Performance Computing (HPC), as it can solve problems beyond the reach of HPC. To that end, HPC will require quantum accelerators, which will enable applications to run on both classical and quantum devices, via hybrid quantum-classical nodes. Hybrid quantum-HPC applications should be scalable, executable on Quantum Error Corrected (QEC) devices, and could use quantum-classical primitives. However, the lack of scalability, poor performances, and inability to insert classical schemes within quantum applications has prevented current quantum frameworks from being adopted by the HPC community.

This paper specifies the requirements of a hybrid quantum-classical framework compatible with HPC environments, and introduces a novel hardware-agnostic framework called Q-Pragma. This framework extends the classical programming language C++ heavily used in HPC via the addition of pragma directives to manage quantum computations.

1   Introduction

Quantum computing is a paradigm of computer science that uses quantum physics to solve some computationally hard problems faster than a classical computer. These problems, such as cracking encryption [32] or simulating quantum chemistry systems [22, 7], are currently solved using High Performance Computing (HPC) systems, but can be tackled more efficiently using quantum algorithms, suggesting that quantum computing will revolutionize HPC. However, these quantum algorithms are not expected to replace existing classical applications; they will be integrated inside existing

1

softwares to avoid some heavy classical computations. For example, the application that cracks encryption will use a quantum processor to find an integer divisor while parsing a SSL certificate or decoding an encrypted message will require classical resources. Applications using quantum resources will be hybrid, requiring both quantum and classical resources. Moreover, the classical code used in these hybrid applications is likely already implemented. Then, extending classical applications to integrate quantum kernels is a crucial step in the process of making quantum devices useful.

This kind of integration has already been done in classical computing, using more than one type of processor or core. For example, in general-purpose computing on Graphical Processing Units (GPU), the Central Processing Unit (CPU) delegates some tasks to the GPU acting as an accelerator [19]. The emergence of new programming tools such as CUDA [30] and OpenMP [10, 24] have made classical hybridization possible, based on a well-defined hardware connection between the GPU and the CPU. Similarly, tasks could be offloaded to a Quantum Processing Unit (QPU), by taking into account the interaction between classical and quantum hardware. The main challenge would be to connect the quantum computing systems with classical ones, and to define the programming interface to communicate with such quantum resources.

Proposition of architectures integrating quantum devices into a HPC node[1] were recently published [4, 20] (these nodes are referred to as "hybrid quantum-classical nodes"). In these designs, the classical resource (named *Host*) is placed close to the quantum device to reduce latency and to enhance the quantum computation efficiency. Reducing the latency improves the computation and allows for efficient data-transfers between the components of the node. In these designs, the Host is directly connected to a controller, which is a piece of classical software and electronics responsible for manipulating the quantum state within the quantum device.
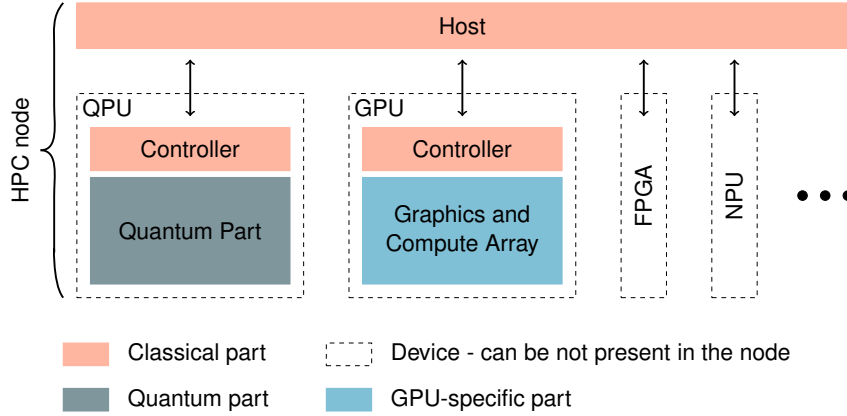

Figure 1: Hybrid HPC node - composed of a Host and one or several devices. The QPU is connected to the Host like any other device.

In Figure 1, the QPU controller receives instructions from the Host to execute them

---

[1]A node is the minimal entity on which an Operating System (OS) can be installed

on the quantum part. It schedules these instructions, and can execute classical computations that interact with the quantum part. Incorporating classical computations alongside quantum computation is used to manage Quantum Error Correction (QEC), but can also be used to optimize a quantum routine. For example, this method is used to reduce the gate count of arithmetic operations [14]. The incorporation of classical computation directly impacts the design of a hybrid quantum-HPC framework.

Existing quantum frameworks (like OpenQASM 3 [9], Q# [36], and Scaffold [1]) do not take advantage of this hardware integration, by restricting CPU/QPU interactions. Moreover, running a quantum routine within an existing application appears challenging, as the discussion on the interoperability between HPC and quantum computing remains limited. Then, redesigning classical-quantum programming is a foremost requirement, but this redesign can also simplify the way hybrid applications are implemented. To do so, defining the specifications of a hybrid quantum-HPC framework is necessary.

This paper proposes a definition of a hybrid quantum-HPC framework based on 7 criteria, inspired by well-established classical and quantum hybrid frameworks. Based on these criteria, a new C++ quantum-HPC framework called "Q-Pragma" is introduced. This framework integrates modern programming concepts to satisfy HPC constraints and to simplify the integration of quantum kernels in existing applications. Nevertheless, these constraints are not an obstacle to use Q-Pragma in a non-HPC context, and can even simplify the development of hybrid algorithms. For instance, advanced quantum algorithms can be written using only few lines of code (see Appendix A). A core concept of Q-Pragma is to use pragma directives to add quantum-classical hybridization capabilities to C++. Q-Pragma is a hardware-agnostic framework built on existing HPC design patterns, making it portable, scalable, and usable with classical HPC frameworks.

In the next section, the term "hybrid quantum-HPC" will be used, since this section focus on the integration of a quantum device in a HPC system. Subsequently, the term "hybrid quantum-classical" will be preferred to discuss about hybridization in general (and not only HPC).

2 Defining a hybrid quantum-HPC framework

Developing a hybrid quantum-HPC framework seamlessly interoperable with HPC languages is of paramount importance. Such interoperability would guarantee compatibility and facilitate smooth integration with current HPC infrastructures and software ecosystem. Additionally, these specifications play a critical role in shaping the framework functionalities and performance.

HPC is already hybrid, as many existing applications rely on GPUs. This section focus on the existing classical hybrid frameworks, to highlight HPC-friendly design patterns which can be used in a quantum-HPC framework. Then, this section focus on existing quantum frameworks to identify features that need to be implemented in quantum-HPC frameworks.

2.1 Classical hybrid computing

Within the realm of HPC, the most prominent languages are Fortran [3], C [23], and C++ [35]. Fortran, the oldest among them, maintains its active presence in HPC, primarily due to its utilization in legacy codes. Meanwhile, C stands out for its precise memory management and system operations, making it particularly powerful for performance optimization. Additionally, a vast range of libraries and tools like BLAS [6] and LAPACK [2] are available for C, enriching its development ecosystem. On the other hand, despite its greater complexity compared to C and Fortran, C++ offers the advantages of object-oriented programming, significantly enhancing code maintainability in large-scale HPC applications. Furthermore, C++ has template metaprogramming capabilities that enable compile-time code generation, thus enhancing performance further. These three programming languages possess interoperability, meaning that a function written in any of these languages can be invoked or utilized within another language. Creating a framework that works easily with today's HPC languages allows us to use pre-existing tools including parallel programming tools and optimized libraries.

In classical hybrid computing, interoperability with one of these programming languages prevails. There are two types of classical hybrid computing frameworks, frameworks based on a new programming language, and frameworks extending an existing HPC language.

Hybrid frameworks based on a new programming language
Some hybrid applications already use third-party languages to run code on specific devices. For example, CUDA [30] and OpenCL [34] are programming languages designed to take advantage of GPUs.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA, specifically designed to exploit the computational power of GPUs for general-purpose computing tasks. CUDA provides a C-like programming interface, allowing developers to express GPU computations within their existing C or C++ codebase.

Similarly OpenCL is an open standard parallel programming framework that targets heterogeneous computing environments like CPUs, GPUs and FGPAs. OpenCL also supports several programming languages such as C, C++ and Python.

This design pattern provides the concept of *kernel* (i.e. a device-specific function which can be called from a HPC language) and provides tools to allocate memory on

the device. This design pattern has been chosen by some quantum frameworks, including CUDA Quantum [11], Q# [36], OpenQASM 3 [9], and Scaffold [21]. Howeover, with the exception of CUDA Quantum, none of these programming languages incorporate the concept of kernel, preventing these frameworks from being used in HPC environments.

Hybrid frameworks extending existing programming languages
Another option is to enhance an existing language to streamline integration. Among conventional hybrid programming solutions, OpenMP [10, 24] (Open Multi-Processing) stands out as a prominent programming model engineered to seamlessly integrate GPU-specific code within established programming languages like C, C++, or Fortran. OpenMP is based on a collection of compiler directives, library functions, and environment variables. These directives, highlighted by pragmas, allow programmers to define code sections for GPU execution without having to learn a new programming language. They serve the purpose of adding device specific instructions as well as specifying the code and memory locality (i.e. selecting the device on which a function is executed, or a variable stored). This design pattern has been chosen by QCOR [26], by adding function attribute specifiers to the C++ language.

These two classical design patterns address a critical concern of classical hybrid environments, namely code and memory locality control. This concern should be addressed by quantum-HPC frameworks through the following specification:

Code and memory locality
A quantum-HPC framework provides tools to control code and memory locality.

Existing classical HPC frameworks are extended with device-specific features to encompass the full spectrum of capabilities offered by the device. Thus, the capabilities of a QPU must be defined in order to build a quantum-HPC framework.

2.2   Quantum computing features for HPC

Quantum computing will impact HPC. To harness this acceleration, quantum computing features for HPC must be listed. A set of hardware capabilities, through the seven DiVicenzo's criteria [12], has defined what a universal quantum computer is. This section defines a list of software capabilities to better take advantage of a such device. An overview of these specific capabilities has never been done before. Inspired by the state of the art, this subsection outlines the main features of a quantum-HPC framework. It involves using millions of physical qubits, correcting errors for accurate results, and spreading computations across multiple QPUs for better processing. These requirements will need specific capabilities from the QPU.

Dynamic interaction   *Classical memory or quantum memory can be allocated on a QPU and manipulated from the Host, at runtime.*

Hybrid quantum-HPC applications require their classical part to interact with their quantum part. To interact with a quantum computation, the Host needs to send and receive information from the QPU. The Host should then be able to access QPU's memory, by reading and writing data directly on the QPU.

Scalability *Hybrid quantum-HPC frameworks should support algorithms with arbitrary big number of qubits, or instructions.*

Pratical quantum-HPC applications, like Shor algorithm running on an error-corrected QPU [15], would require millions of physical qubits (the *scalability* is a hardware requirement, since it is a DiVicenzo's criteria). Then, any hybrid quantum-HPC framework must be able to handle such large amount of physical qubits. Nowadays, the circuit formalism used to describe quantum algorithms represents, on a timeline, the different operations that will be applied on the quantum state. This formalism is static and is used by most of today's quantum frameworks like myQLM [13], Qiskit [31], and Cirq [8]. Nevertheless, the circuit formalism is not designed to represent billions of operations acting on millions of qubits, as it would be necessary to create a data structure of several gigabytes to represent a single instance of the problem. Moreover, the integration of classical operations that have an impact on the quantum computation is not compatible with this formalism.

Typing *Quantum registers are typed to simplify the manipulation of huge structures.*

Today, the majority of classical programming languages utilizes typing, where source code handles not just booleans but also more complex data types like integers, floating points, and arrays. Employing typing in a language streamlines the source code and empowers users to create sophisticated algorithms. The use of typing in a framework is essential to ensure scalability. A quantum-HPC framework should offer predefined quantum data types while also permitting users to define their own custom quantum structures.

Reversibility *Pure quantum operations are reversible.*

Quantum reversibility is a fundamental concept in quantum computing, referring to the ability to reverse quantum operations and undo their effects [29]. Any quantum routine written in a hybrid framework should be reversible.

Controllability *Pure quantum operations are controllable.*

In quantum computing, control gates such as Toffoli gates have a crucial function as they facilitate the creation of entanglement [29]. This entanglement, along with the concept of superposition, empowers quantum computers to execute advanced computations that greatly surpass the classical computer capabilities. The ability to control a quantum routine constitutes a fundamental concept within a quantum-HPC framework.

Safe uncomputation *Quantum registers can be reset to $|0\rangle$ state without any measurement.*

Unlike classical computing, resetting a qubit is dangerous (reset being composed of a measurement, and optionally a X gate). A qubit may be entangled with another one, so resetting a qubit could have an impact on other qubits. Safe uncomputation is a concept defined in existing quantum frameworks like Quipper [17] or Silq [5], relying on the reversibility of quantum unitary operations to reset a qubit to the $|0\rangle$ state.

None of the existing quantum frameworks fulfill the aforementioned requirements,

see Table 1. A new C++ framework for hybrid quantum-classical computing fulfilling all these requirements is introduced. This framework, compatible with HPC environments, is composed of both a C++ library, as well as a compiler plugin providing C++ pragmas to control the quantum computation.

✓ Fully supported
~ Partially supported

Framework	Locality	Dynamic interaction	Scalability	Typing	Reversibility	Controllability	Safe uncomputation
Cirq [8]					✓	✓	
CUDA Quantum [11]	~				✓	✓	~
myQLM [13]				✓	✓	✓	~
OpenQASM 3 [9]		✓	✓		✓	✓	
ProjectQ [33, 18]		✓	✓		✓	✓	~
Q# [36]		✓	~		✓	✓	~
QCOR [26]	~				✓	✓	
Qiskit [31]					✓	✓	
Quipper [17]					✓	✓	✓
Scaffold [1]						✓	
Silq [5]				✓	✓	✓	✓
CUDA [30]	✓	✓	✓	✓			
OpenCL [34]	✓	✓	✓	✓			
OpenMP [10, 24]	✓	✓	✓	✓			

Table 1: Features implemented by existing hybrid frameworks - the first group corresponds to quantum frameworks, the second one corresponds to classical frameworks (see Appendix B for details).

3   Q-Pragma: a C++ hybrid framework

This section introduces a new framework, called "Q-Pragma" that bridges the classical and quantum computing worlds, even within HPC ecosystems. Q-Pragma is a C++-based framework that not only empowers classical applications with quantum capabilities but also satisfies the *code and memory locality* requirement. This framework constitutes a first step towards the development of hybrid quantum-classical applications, and adds the power of quantum computing to the computational efficiency and scalability offered by HPC systems.

At its core, this framework comprises a versatile C++ library designed to facilitate the seamless integration of quantum operations into classical C++ code. This integration enables developers to explore quantum algorithms and harness quantum hardware resources without completely overhauling their existing classical codebases. Moreover, C++ pragmas specifically tailored to extend the language expressive power are introduced, allowing for the efficient description of quantum operations within classical code, thus forming a bridge between the two computing paradigms.

In Q-Pragma, the *dynamic interaction* requirement is satisfied through the memory sharing enabled by the HPC link between Host and QPU [20]. This is discussed in Subsection 3.2.1.

Additionally, the QPU connected to the Host is directly targeted by Q-Pragma, simplifying hybrid application development. This targeting relies on a C++ library available on the node. This library provides a connection with the QPU, and instructions are directly streamed through this library. This fulfills the *scalability* requirement. Streaming instructions avoid the drawbacks of the circuit structure, permitting the framework to manipulate large scale quantum computations, as well as allowing classical computation to interact with the quantum part. Streaming instructions is a known pattern used by OpenQASM 3 [9] or ProjectQ [33, 18].

The subsequent subsections delve into the architecture, design principles, and key features of this C++ framework. They focus on the typing system introduced by Q-Pragma, the concept of quantum routine and the directives extending the C++ language.

3.1 Q-Pragma: A quantum library

Q-Pragma provides a C++ library defining new quantum types as well as the concept of quantum routines. The new quantum types guarantee compliance with the *typing* and *safe uncomputation* requirements, while the quantum routine concept meets the *reversibility* and *controllability* requirements.

3.1.1 Quantum types

In order to make Q-Pragma intuitive and well integrated with classical C++, new quantum types have been added to the framework. These quantum types can describe arbitrary data-structures, including user defined quantum structures. Q-Pragma quantum types can be grouped into two categories:

- Quantum boolean, named "qbool" in Q-Pragma. A quantum boolean can be seen as a unique pointer, pointing to a quantum memory address. This "qbool" is a quantum register of size 1 and is the elementary brick of the quantum type design.

- Quantum array, an array of quantum bools having a fixed size. Pure quantum types defined in this framework inherit from a quantum array.

A pure quantum register has a fixed size. This size is defined by a constant value and not by a classical variable stored in a classical register, thereby making the register a pure quantum object. This follows the philosophy of C++ types: any C++ type has a fixed size. Moreover, pure quantum functions built upon these fixed-size quantum registers can be optimized at compile-time. Dynamic-sized quantum types are built using a classical C++ container (e.g. vector, list). They are composed of both classical and quantum registers, and as such are called hybrid structures.

The safe uncomputation concept stems from the reversibility of quantum unitary operations to reset a quantum register to the $|0\rangle$ state. It has been introduced in some quantum frameworks like Quipper [17] or Silq [5]. Q-Pragma provides a safe uncomputation mechanism based on the initialization of a quantum object. Each quantum type handles its own "initialization" (i.e. computing the state preparation - done by the constructor of the quantum type) and "deinitialization" (i.e. uncomputing the state preparation - done by the destructor of the quantum type, ignored if a measurement occurred on this register). Any quantum operation executed on the quantum register must be uncomputed to get $|0\rangle$ state. Through a *get_init* method one can cast a classical type value into a state preparation routine. This routine is used to initialize the state and then stored to undo this state preparation at delete time.

Listing 1 illustrates the safe uncomputation mechanism. The *qreg* quantum register is initialized to state $|1\rangle$ (to do so, an implicit X gate is applied on the qubit). At the end of the scope, the *qreg* register is no longer accessible, so Q-Pragma reset the register to state $|0\rangle$ (by applying a X gate on the qubit).

```
1  {
2      // Initialize a quantum register based on
3      // a classical value
4      // This initialization creates a state
5      // preparation (here applying a X gate)
6      qbool qreg = true;
7
8      ...
9
10     // Object "qreg" is deleted, the safe
11     // uncomputation mechanism applies a
12     // X gate to reset this register to |0⟩
13 }
```

Listing 1: Safe uncomputation mechanism

Similarly, the *cast_measure* method consists in casting a quantum measurement into a classical interpretation of it. This approach forms an explicit connection between classical and quantum types. For instance, measuring a qbool gives a bool while measuring a quantum array leads to a classical array of bools. If one creates a new quantum type, one can arbitrarily choose the way classical and quantum data interact through this new type.

```
1  // State preparation
2  quint8_t quint = 12UL;
3
4  ...
5
6  // Measurement of a quantum integer
7  uint8_t cuint = measure_and_reset(quint);
```

Listing 2: Casting a quantum integer into a classical integer - *A quantum integer being an array of qbools, casting the measurement consists in translating an array of bool (i.e. bitstring) into a classical integer*

Q-Pragma provides three built-in quantum types: the *qbool* (quantum bool or qubit), the *quint_t* (quantum unsigned integer) and the *qint_t* (quantum signed integer). The *quint_t* and the *qint_t* are inspired from the C++ classical types *uint_t* and *int_t* respectively. These two types belong to the *quantum array* category, meaning that *quint_t* and *qint_t* are fixed-sized arrays of qbools, their size being given by a template parameter. Note that any quantum type can be created following the *quint_t* conception.

A quantum integer can be initialized from a classical integer, and measuring a quantum integer returns a classical integer. Listing 2 illustrate this behavior, the quantum register *quint* will be initialized with the quantum state $|00001100\rangle$, which is the binary encoding for 12; at the end of the scope, the quantum register *quint* is measured in the *Z*-basis, and the outcome bitstring is casted into a classical integer. To implement this behavior, quantum type *quint_t* overloads the *get_init* and

cast_measure methods, enabling this type to be instantiated from an unsigned integer or a positive signed integer.

```
1  /* QUINT_T */
2  quint8_t quint_a;
3  quint8_t quint_b = 42;
4
5  // Apply operations
6  quint_a ^= (quint_b + 7);
7
8  // Measure a register
9  uint8_t cuint_a = measure_and_reset (quint_a);
```

Listing 3: Usage of a 8 qubits quantum integer

A *quint_t* behaves as a classical integer, enabling bitwise operations and some arithmetic operations, as shown in Listing 3. This can be done by overriding C++ operators. For instance, overriding the "+" operator enables the addition between *quint_t* and other quantum or classical integers. The same mechanism applies for *qbool* and *qint_t*.

Any built-in Q-Pragma quantum type behaves as its equivalent classical type. A quantum type can be initialized using its equivalent classical type and is manipulated using the same set of operations, as long as the operations remain reversible, as shown in Listing 3 or Listing 4. Measuring a quantum type returns its classical equivalent. Moreover, any quantum type can be initialized using a C++ expression.

```
1  // Initializing two qbools from classical bools
2  qbool a = true , b = false;
3  // Initializing a qbool from a C++ expression
4  qbool c = a | b;
```

Listing 4: Initialization of a quantum type using a C++ expression

C++ operators can be used to define complex quantum routines using only few lines of code. In addition, this section shows that Q-Pragma adheres to several requirements described in the previous section, namely the *typing* and *safe uncomputation* requirements.

3.1.2 Quantum routines

A quantum routine is a pure quantum function acting on a fixed-size quantum register. It is by definition controllable and reversible. A quantum routine is either:

- A native gate, also called "basic gate" in Q-Pragma. These basic gates are the usual gates used in quantum computation, including Pauli gates X, Y and Z, Toffoli gates, CNOT gates, and others.
- A sequence of other quantum routines, each acting on a subset of qubits.

A routine being a function, it can be built using the same syntax as any C++ function. A directive `#pragma quantum routine` must be used to declare this function as a quantum routine. The behavior of this pragma is explained in the next section.

A quantum routine can be executed using the *operator()* method, to be called using the same syntax as any C++ function. Nevertheless, a quantum routine features extra methods to change the behavior of this function, as shown in Listing 5:

- Method *dag()* is used to call the inverse version of the routine.
- Method *ctrl()* is used to call the control version of the routine. The size of the register passed in argument defines the number of controls.
- Method *ctrl_dag()* is used to call the control version of the inverse. The size of the register passed in argument defines the number of controls.

```
1  qbool q0, q1, qc;
2
3  // Apply the routine
4  my_routine(q0, q1);
5
6  // Apply the reverse routine
7  my_routine.dag(q0, q1);
8
9  // Apply the controlled routine
10 // The routine is controlled by a single qubit
11 my_routine.ctrl(qc, q0, q1);
12
13 // Apply the reverse and controlled routine
14 my_routine.ctrl_dag(qc, q0, q1);
```

Listing 5: Usage of a quantum routine

The quantum routine ensures that Q-Pragma supports the *reversibility* and *controllability* requirements.

3.2 Q-Pragma: extending C++

C++ pragma directives are powerful tools used to specify several compilation features, while maintaining compatibility with the C++ language. Q-Pragma relies on these directives to add quantum programming capabilities to C++. This is achieved using five new pragma directives, allowing hybrid quantum-classical code implementation while remaining consistent with C++, and comprehensible by C++ developers.

These pragmas are used to:

- manage code and memory locality, i.e. *quantum scope* and *quantum move* directives.
- control quantum instructions, i.e. *quantum ctrl* directive.
- define quantum routines, i.e. *quantum routine* directive.
- manage the uncomputation, i.e. *quantum compute* directive.

3.2.1 Managing code and memory locality

The *pragma quantum scope* is the first pragma directive of our framework. It indicates that the following scope has to be run on the quantum device. The QPU has classical capabilities through the controller (see Figure 1). Q-Pragma is based on the assumption that this controller is programmable. Therefore, a quantum scope can contain pure quantum operations, classical instructions, and non-reversible quantum operations like measurements. The *pragma quantum scope* compiles the source code to be offloaded on the QPU, reducing the amount of data transferred, and thus increasing performances. Any function defined on the Host can be called from the quantum device, and a function defined on the quantum device can be called from the Host (quantum gates can then be executed from the Host). This involves a PCI Express communication between QPU and Host, and is therefore slower than executing a local function.

```
1  #pragma quantum scope
2  {
3      std::array<int, 4UL> counters;
4      qbool q0, q1;
5
6      for (int i = 0 ; i < 100 ; ++i) {
7          // Generate a Bell pair
8          H(q0);
9          CNOT(q0, q1);
10
11         // Measure
12         bool meas0 = measure_and_reset(q0);
13         bool meas1 = measure_and_reset(q1);
14
15         // Store measurement in the map counter
16         ++counters[meas1 << 1 + meas0];
17     }
18 }
```

Listing 6: Example of a *pragma quantum scope*

Listing 6 illustrates a *pragma quantum scope* directive: the sample of code inside the scope is executed on the QPU, not on the Host. This pragma directive is not mandatory and does not change the program output, but serves to improve performances: the Host communicates only once with the quantum device (to start the scope) instead of a hundred times (to allocate memory, to apply quantum gates or to measure a qubit).

Any piece of code written inside a *pragma quantum scope* directive is executed on the quantum device, otherwise the code is executed on the Host. If a variable is defined in a *pragma quantum scope* directive, the memory is allocated on the quantum device, otherwise the memory is allocated on the Host. A *qbool* can be allocated and manipulated from the Host, as this structure is a classical object pointing to a quantum memory address (a *qbool* can be seen as a unique pointer). Host variables can be accessed in a *pragma quantum scope*, using the memory sharing enabled by the PCI Express protocol. Nevertheless, variable access between Host and QPU is inherently slow. To compensate for this, Q-Pragma handles the memory through data management directives. Some variables can be temporarily moved from the Host to the QPU in a quantum scope, using the *with* keyword. At the beginning of the scope, the listed variables are moved to the device, and moved back to the Host at the end of the scope. Listing 7 shows how to temporarily move the *my_ int* variable on the QPU.

```
1  qint8_t my_int;
2
3  #pragma quantum scope with (my_int)
4  {
5      my_int += 13;
6  }
```

Listing 7: Example of a *pragma quantum scope with*

Moreover, the *pragma quantum move* directive handles manual data transfer from Host to device through the *toDevice* specifier, and from device to Host through the *toHost* one. In this case, the developer has total control over the data flow between Host and device as shown in Listing 8.

```
1  std::array<qbool, 8UL> qreg;
2  bool condition = ...;
3
4  #pragma quantum scope
5  {
6      if (condition) {
7          #pragma quantum move toDevice(qreg)
8          for (const auto & qubit : qreg) {
9              H(qubit);
10         }
11         #pragma quantum move toHost(qreg)
12     } else {
13         // DO SOMETHING with no memory movement
14     }
15 }
```

Listing 8: Example of a *pragma quantum move* directive

These two pragma directives provide fine control over both code and memory locality. One can precisely select on which device a function is executed, or a variable stored. The syntaxes of these pragma directives are specified in Figure 2.

⟨pragma_scope⟩ ::= "#pragma quantum scope" ⟨scope_options⟩?
⟨scope_options⟩ ::= "with" ⟨var_list⟩

⟨pragma_move⟩ ::= "#pragma quantum move" (⟨move_dir⟩ ⟨var_list⟩)+
⟨move_dir⟩ ::= "toDevice" | "toHost"

⟨var_list⟩ ::= "(" ⟨var_name⟩ ("," ⟨var_name⟩)* ")"

Figure 2: Grammar of *pragma quantum scope* and *pragma quantum move* using BNF with regex

3.2.2 Controlling quantum instructions

The *pragma quantum ctrl* directive is used to control quantum instructions, using a *qbool* or a C++ expression. If a C++ expression is used, a temporary *qbool* is created and initialized using the C++ expression. Listing 9 highlights this behavior: a temporary *qbool* is instantiated (with the value $my\_int == 42$), and this *qbool* is used to control the scope; thanks to the *safe uncomputation* mechanism, this temporary *qbool* is reset to state $|0\rangle$ at the end of the scope.

```
1  qint8_t my_int;
2  // Some code acting on my_int
3  // ...
4
5  qbool q0, q1;
6  #pragma quantum ctrl (my_int == 42)
7  {
8      H(q0);
9      CNOT(q0,q1);
10     RZ(M_PI / 4.)(q1);
11 }
```

Listing 9: Example of a *pragma quantum ctrl* controlled by a C++ expression

Only pure quantum operations are impacted by a control. Control have no impact on classical operations, and quantum measurements are forbidden. A *pragma quantum ctrl* is similar to a quantum *if* statement in the sense that the controlled quantum instructions affect the quantum state only if the control qubit evaluates to *true*. Nevertheless, unlike the C++ *if* statement, instructions inside a *pragma quantum ctrl* are always executed in their controlled version. Using a different syntax between the C++ *if* statement and the *pragma quantum ctrl* avoids confusion. The syntax of this pragma is defined by Figure 3.

⟨pragma_ctrl⟩   ::=   "#pragma quantum ctrl" ⟨ctrl_options⟩
⟨ctrl_options⟩   ::=   "(" ⟨var_name⟩ | ⟨quantum_condition⟩ ")"

Figure 3: Grammar of *pragma quantum ctrl* using BNF with regex

This pragma meets the *controllability* requirement of a quantum-classical framework compatible with HPC systems.

3.2.3   Defining quantum routines

The *pragma quantum routine* directive is used to create quantum routines using the same syntax as a C++ function. A quantum routine is composed of purely quantum operations, and is reversible. Therefore:

- a routine takes only purely quantum objects as arguments.

- measurement-based operations are not allowed.

- a routine returns nothing (i.e. is void type), since no classical data is either passed as input or created (no measurement).

To meet the *reversibility* and *controllability* requirements, a routine is callable, reversible and controllable, as shown in Listing 10. Since a quantum routine is a pure quantum function, the routine is compiled for the QPU: executing a quantum routine from the Host requires a single communication request between the Host and the QPU, regardless of the quantum routine size.

```
1  #pragma quantum routine
2  void my_routine(const qbool & q0, const qbool & q1)
3  {
4      H(q0);
5      Z.ctrl(q0, q1);
6  }
7
8  int main() {
9      qbool q0, q1, qc;
10     // Apply the routine
11     my_routine(q0, q1);
12     // Apply the reverse routine
13     my_routine.dag(q0, q1);
14     // Apply the controlled routine
15     my_routine.ctrl(qc, q0, q1);
16     // Apply the reverse and controlled routine
17     my_routine.ctrl_dag(qc, q0, q1);
18 }
```

Listing 10: Example of the creation and calls of a quantum routine using the *pragma quantum routine* directive

In Listing 10, the *my_routine* routine is compiled for the QPU, i.e. the routine object only exists on the QPU. The *main* function allocates three qubits and performs four calls to *my_routine*. Since the sample of code is executed on the Host (due to the lack of *pragma quantum scope* directive), each allocation and each routine call sends a request to the QPU. A response is sent back by the QPU after each request, to notify the Host that the request has been executed on the QPU (so the Host can execute the next instruction). Then, executing Listing 10 requires 7 requests to the QPU.

As any C++ function, quantum routines can be templated. For instance, one can create a templated Quantum Fourier Transform (QFT), as shown in Listing 11. The template parameter *SIZE* is reused to define the size of the argument *qreg*, but also inside the routine as a stop condition for the *for* loops.

```
1  #pragma quantum routine
2  template <uint64_t SIZE>
3  void qft(const std::array<qbool, SIZE> & qreg) {
4      for (uint64_t idx = 0ul ; idx < SIZE ; ++idx) {
5          H(qreg[idx]);
6
7          for (uint64_t ctr = idx + 1 ; ctr < SIZE ;
8               ++ctr) {
9              // Compute angle
10             double angle =
11                 M_PI / (1 << (ctr - idx));
12
13             // Apply gate
14             PH(angle).ctrl(qreg[ctr], qreg[idx]);
15         }
16     }
17 }
18
19 int main() {
20     qint8_t qreg;
21     qft<8UL>(qreg);
22 }
```

Listing 11: QFT on an array, using Q-Pragma

Moreover, quantum routines can be parameterized. According to our routine definition, the R_X gate is not a routine as it depends on a classical angle (a routine takes only pure quantum objects as argument). On the other hand, $R_X(\theta)$ is a quantum routine since its angle is known, and the gate itself does not depend on any classical input. In Q-Pragma, R_X is a parameterized routine, i.e. an object which, binded with classical parameters, becomes a routine. Variational algorithms can be implemented with parameterized routine in Q-Pragma.

A parameterized routine can be built using the *pragma quantum routine*, by listing the binded variables in the directive. Listing 12 provides an example of a parameterized routine.

```
1  #pragma quantum routine (double angle0, double angle1)
2  void param_routine (const qbool & q0,
3                      const qbool & q1) {
4      RX(angle0).ctrl(q0, q1);
5      RZ(angle1).ctrl(q0, q1);
6  }
7
8  int main () {
9      qbool q0, q1;
10     param_routine(M_PI/3.0, M_PI/6.0)(q0, q1);
11 }
```

Listing 12: Parameterized routine using the *pragma quantum routine*

The requirements introduced with the concept of typed arguments in C++ functions do not necessarily apply to quantum routines generated with a *pragma quantum routine*. In fact, as long as the quantum arguments have exactly the same size as the one given in the function declaration, no error will be raised. This brings flexibility, as shown in Listing 13.

```
1  #pragma quantum routine
2  void bell_pair (const qbool & q0, const qbool & q1)
3  {
4      H(q0)
5      CNOT(q0, q1);
6  }
7
8  int main () {
9      qbool q0, q1;
10     bell_pair(q0, q1);  // Correct
11
12     std::array<qbool, 2> qreg;
13     bell_pair(qreg);  // Still correct
14
15     qint_t<2> qint;
16     bell_pair(qint);  // Still correct
17 }
```

Listing 13: Example demonstrating the flexibility of quantum routine call arguments

However, strict typing can be enforced over the quantum routines using the *typed* flag. This flag ensures that the quantum types given in the quantum routine declaration are exactly the same as the one given in the routine call, as shown in Listing 14.

```
1  #pragma quantum routine typed
2  void bell_pair(const qbool & q0, const qbool & q1)
3  {
4      H(q0)
5      CNOT(q0, q1);
6  }
7
8  int main() {
9      qbool q0, q1;
10     bell_pair(q0, q1);  // Correct
11
12     std::array<qbool, 2> qreg;
13     bell_pair(qreg);  // ERROR - does not compile
14
15     qint_t<2> qint;
16     bell_pair(qint);  // ERROR - does not compile
17 }
```

Listing 14: Example of a typed quantum routine

Additionally, a pure quantum type has a static size in Q-Pragma. Therefore, quantum routines cannot be created using hybrid types such as lists of qubits. To overcome this limitation, dynamic quantum routines have been introduced to create routines acting on a dynamic-sized quantum registers. A dynamic routine can be created using the "dynamic" flag, as shown in Listing 15. Dynamic routines are always typed.

```
1  #pragma quantum routine dynamic (double angle)
2  void rx_wall(const std::vector<qbool> & vector) {
3      for (auto & qubit: vector) {
4          (RX(angle))(qubit);
5      }
6  }
7
8  int main() {
9      std::vector<qbool> qvect(12);
10     rx_wall(0.6)(qvect);
11 }
```

Listing 15: Example of a dynamic quantum routine

Since dynamic and parameterized quantum routines rely on variables defined at runtime, they can only be created at runtime. Fixed-sized quantum routines are created at compile-time, and are then optimized by the compiler. The syntax of *pragma quantum routine* is defined in Figure 4.

$$\begin{array}{lll}
\langle\text{prag\_routine}\rangle & ::= & \text{``\#pragma quantum routine''} \langle\text{flag}\rangle?\ \langle\text{bound\_vars}\rangle? \\
\langle\text{flag}\rangle & ::= & \text{``dynamic''} \mid \text{``typed''} \\
\langle\text{bound\_vars}\rangle & ::= & \text{``(''} \langle\text{variable}\rangle\ (\text{``,''} \langle\text{variable}\rangle\ )^*\ \text{``)''} \\
\langle\text{variable}\rangle & ::= & \langle\text{cpp\_type}\rangle \langle\text{var\_name}\rangle
\end{array}$$

Figure 4: Grammar of *pragma quantum routine* using BNF with regex

3.2.4 Managing the uncomputation

The *safe uncomputation* requirement is a key element of a quantum-classical framework. Q-Pragma provides a *pragma quantum compute* directive in which pure quantum gates are executed. This directive enables the automatic uncomputation of these pure quantum operations as soon as the end of the current scope is reached. For instance, implementing a R_{ZZ} gate relies on an ancilla qubit which must be reset to its $|0\rangle$ state, this can be implemented using *pragma quantum compute* as shown in Listing 16.

```
1  #pragma quantum routine (double angle)
2  void RZZ(const qbool & qb1, const qbool & qb2) {
3      // Define ancilla
4      qbool ancilla;
5
6      {
7          #pragma quantum compute
8          {
9              CCNOT(qb1, qb2, ancilla);
10         }
11
12         // Apply RZ gate
13         (RZ(angle))(ancilla);
14
15         // Automatic uncomputation of CCNOT
16         // "ancilla" is then reset to |0⟩
17     }
18 }
```

Listing 16: Example of a parameterized routine using the *pragma quantum routine* directive

This directive has been designed to meet the *safe uncomputation* requirement, but can also be used to implement a routine matching the pattern $U \cdot A \cdot U^\dagger$. This allows additional optimization techniques. For instance, the computation and uncomputation parts are never controlled, even if they are in a *pragma quantum ctrl* context. The syntax of this directive is shown in Figure 5.

$$\langle\text{pragma\_compute}\rangle \quad ::= \quad \text{``\#pragma quantum compute''}$$

Figure 5: Grammar of *pragma quantum compute* using BNF with regex

4 Evaluation of Q-Pragma

Q-Pragma has been designed to simplify the development of quantum algorithms. The article presenting the SILQ framework [5] compares some frameworks to show how suitable they are for implementing various quantum algorithms. This benchmark has been extended to add Q-Pragma figures, as shown in Figure 6.

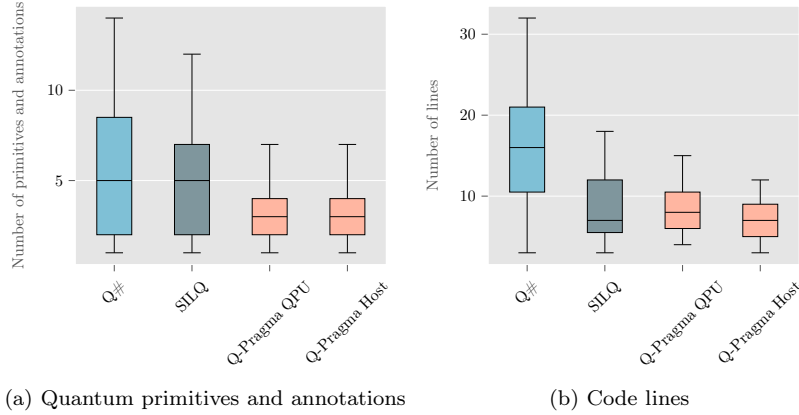(a) Quantum primitives and annotations   (b) Code lines

Figure 6: Evaluation of Q-Pragma compared to Q# and SILQ. The comparison between these frameworks is based on the number of quantum primitives and annotations used (Figure 6a) and the code length (Figure 6b) - Q-Pragma allows these algorithms to be implemented on the Host or on the QPU (by using a *pragma quantum scope* directive). Each algorithm of this benchmark has been implemented twice (one implementation executing code on the Host, the other executing code on the QPU). See Appendix C for details.

Typing provides a higher-level interface for manipulating quantum registers, simplifying the development of quantum algorithms. Thus, complex quantum algorithms can be written using fewer quantum primitives (see Figure 6a), and using less line of code (see Figure 6b).

5 Conclusion

Q-Pragma enables quantum code integration in classical applications, and remains compatible with HPC environments. It has been designed as an Open Standard, to ensure that these applications will be executable on real QPUs. This also guarantees that Q-Pragma can evolve, and support new programming features future hybrid code may require. Q-Pragma attempts to meet all the requirements current and future QPUs will establish, facilitating the development of hybrid quantum-classical applications able to reach quantum advantage.

Since the relevance of a framework can only be assessed through practice, our aim is to develop post-NISQ quantum algorithms using Q-Pragma. For example, we would

like to implement several linear algebra algorithms, such as QSVT [16, 25], since their classical equivalents are of great interest in HPC.

Furthermore, Q-Pragma has only been connected to quantum emulators, and not yet to real quantum hardware. We would like to work with QPU makers to demonstrate a real use case of Q-Pragma.

Q-Pragma also introduces new compilation constraints. Current quantum compilers process quantum circuits, acting on a known set of qubits. Q-Pragma defines quantum routines, but the register on which a routine will be applied is not known at compile time. This makes it difficult to consider topological constraints when compiling a Q-Pragma program. We would like to work on compilation within the Q-Pragma framework, whether in a QEC scope or not.

This Q-Pragma framework has been implemented in C++17, and relies on a Clang plugin. Some common quantum algorithms are implemented in Q-Pragma, code examples are given in the Appendix A.

6   Acknowledgement

References

[1] Ali Abhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, Frederic Chong, Margaret Martonosi, Martin Suchara, Ken Brown, Massoud Pedram, and Todd Brun. Scaffold: Quantum programming language, 07 2012.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] John Backus. The history of fortran i, ii and iii. *Annals of the History of Computing*, 1(1):21–37, 1979.

[4] K. Bertels, A. Sarkar, T. Hubregtsen., M. Serrao, A. A. Mouedenne, A. Yadav, A. Krol, and I. Ashraf. Quantum computer architecture: Towards full-stack quantum accelerators. In *2020 Design, Automation & Test in Europe Conference & Exhibition (2020)*. IEEE, March 2020.

[5] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin 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*. ACM, June 2020.

[6] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[7] Hans Hon Sang Chan, Richard Meister, Tyson Jones, David P. Tew, and Simon C. Benjamin. Grid-based methods for chemistry simulations on a quantum computer. *Science Advances*, 9(9), mar 2023.

[8] Cirq Developers. Cirq, 2023.

[9] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, September 2022.

[10] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[11] NVIDIA CUDA Quantum development team. Cuda quantum, 2023.

[12] David P. DiVincenzo. The physical implementation of quantum computation. *Fortschritte der Physik*, 48(9–11):771–783, September 2000.

[13] Eviden Quantum Lab. myQLM: Quantum Computing Framework, 2020-2023.

[14] Craig Gidney. Windowed quantum arithmetic. *arXiv: Quantum Physics*, May 2019.

[15] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, April 2021.

[16] András Gilyén, Yuan Su, Guang Hao Low, and Nathan Wiebe. Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 2019.

[17] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoî t Valiron. Quipper. *ACM SIGPLAN Notices*, 48(6):333–342, jun 2013.

[18] Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. A software methodology for compiling quantum programs. *Quantum Science and Technology*, 3(2):020501, February 2018.

[19] Qihang Huang, Zhiyi Huang, Paul Werstein, and Martin Purvis. Gpu as a general purpose computing resource. In *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 151–158, 2008.

[20] Travis S. Humble, Alexander McCaskey, Dmitry I. Lyakh, Meenambika Gowrishankar, Albert Frisch, and Thomas Monz. Quantum computers for high-performance computing. *IEEE Micro*, 41(5):15–23, 2021.

[21] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. ScaffCC: Scalable compilation and analysis of quantum programs. *Parallel Computing*, 45:2–17, jun 2015.

[22] Ivan Kassal, James D. Whitfield, Alejandro Perdomo-Ortiz, Man-Hong Yung, and Alá n Aspuru-Guzik. Simulating chemistry using quantum computers. *Annual Review of Physical Chemistry*, 62(1):185–207, may 2011.

[23] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson, 2006.

[24] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, feb 2009.

[25] John M. Martyn, Zane M. Rossi, Andrew K. Tan, and Isaac L. Chuang. Grand unification of quantum algorithms. *PRX Quantum*, 2(4), dec 2021.

[26] Alexander Mccaskey, Thien Nguyen, Anthony Santana, Daniel Claudino, Tyler Kharazi, and Hal Finkel. Extending c++ for heterogeneous quantum-classical computing. *ACM Transactions on Quantum Computing*, 2(2):1–36, 2021.

[27] Mariia Mykhailova and Martin Roetteler. Microsoft q# coding contest - summer 2018 - main contest july 6-9, 2018, 2018.

[28] Mariia Mykhailova and Martin Roetteler. Microsoft q# coding contest - winter 2019 - main contest march 1-4, 2019, 2018.

[29] Michael Nielsen and Isaac Chuang. Quantum information theory. In *Quantum Computation and Quantum Information*, pages 528–607. Cambridge University Press, June 2012.

[30] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[31] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[32] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, oct 1997.

[33] Damian S. Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: an open source software framework for quantum computing. *Quantum*, 2:49, January 2018.

[34] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May 2010.

[35] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, 3rd edition, 1997.

[36] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, feb 2018.

A Examples

A.1 Bell pair

The Bell pair algorithm is a quantum routine composed of two quantum gates. These quantum gates are part of Q-Pragma standard library. We provide in Listing 17 an example of such implementation.

```
1  #pragma quantum routine
2  void bell_pair(const qbool & qb0,
3                 const qbool & qb1) {
4      H(q0);
5      CNOT(q0, q1);
6  }
```

Listing 17: Example - Bell pair

A.2 Uniform superposition

A uniform superposition $[0 , 256[$ can be created using wall of Hadamard gates acting on 8 qubits, as $256 = 2^8$. On the other hand, a uniform superposition $[0 , 200[$ requires an additional post-selection, as shown in Listing 18.

```
1  quint8_t quantum_int;
2  qbool ancilla;
3
4  do {
5      // Create a uniform superposition
6      reset(quantum_int);
7      wall::H<8UL>(quantum_int);
8
9      #pragma quantum ctrl(quantum_int >= 200)
10     X(ancilla);
11 } while (measure_and_reset(ancilla));
```

Listing 18: Example - Uniform superposition $[0 , 200[$

A.3 Shor Algorithm

Shor algorithm [32] relies on advanced primitives like the Modular Exponentiation. In Q-Pragma, this primitive can be easily implemented using C++ operators. This is demonstrated in the Shor algorithm implementation in Listing 19. This example only focuses on the quantum part of Shor algorithm.

```
1  // Define the size of the register used to encode
2  // a quantum (unsigned) integer
3  #define SIZE ...
4
5  using namespace qpragma;
6
7  uint64_t find_divisor(uint64_t to_divide) {
8      // Step 1: find a random number
9      std::random_device rd;
10     std::uniform_in_distribution<uint64_t> distrib(
11         2UL, to_divide - 1UL
12     );
13     uint64_t random_number = distrib(rd);
14
15     // If random_number is not coprime with
16     // to_divide, then we found a divisor
17     if (
18         auto gcd =
19             std::gcd(random_number, to_divide);
20         gcd != 1UL
21     )
22         return gcd;
23
24     // Step 2: Perform the quantum part of Shor
25     uint64_t measurement = 0UL;
26
27     #pragma quantum scope with (to_divide, random_number)
28     {
29         quint_t<SIZE> first_register;
30         wall::H<SIZE>(first_register);
31
32         quint_t<SIZE> second_register =
33             qpragma::pow(random_number,
34                          first_register)
35             % to_divide;
36         reset(second_register);
37
38         qft<SIZE>(first_register);
39         measurement =
40             measure_and_reset(first_register);
41     }
42
43     // Step 3: perform the classical part of
44     // Shor algorithm. This part is purely
45     // classical and rely on the continuous
46     // fraction algorithm
47     ...
48 }
```

Listing 19: Example - Shor algorithm

B Features implemented in existing frameworks

This section explains the reasoning behind Table 1 and is composed of two subsections. The first subsection focuses on quantum frameworks, and the second subsection focuses on well-established hybrid CPU-GPU frameworks.

B.1 Quantum frameworks

Locality *A quantum-HPC framework provides tools to control code and memory locality.*

Most of today's quantum frameworks do not provide control over QPU-Host interactions, meaning that the code written in these frameworks can only be executed on a quantum device. Therefore, a user cannot choose whether a function should be executed on the Host or on the QPU. Then, most of today's quantum frameworks do not implement the *locality* feature.

Nevertheless, CUDA Quantum [11] and QCOR [26] introduce the concept of quantum kernels, to specify that a function should be executed on a quantum device (otherwise, the function is executed on the Host). Nevertheless, the *locality* feature is not entirely supported as these frameworks do not provide tools to control the memory locality.

Dynamic interaction *Classical memory and quantum memory can be allocated on a QPU and manipulated from the Host, at runtime.*

Most of quantum frameworks rely on the concept of quantum circuit to describe a quantum computation. In these frameworks, a circuit describes the entire quantum computation: each circuit describes an independent computation, and the quantum memory is reset after the circuit has been executed. Moreover, a quantum circuit being static, a classical computation cannot interact with it during the execution. As a result, quantum frameworks relying on a quantum circuit structure prevent any interaction between the classical and quantum parts of the system, so do not implement the *dynamic interaction* feature.

In most of quantum frameworks, applying a gate is equivalent to adding it into an underlying circuit, but this is not mandatory. For instance, the instruction can be streamed to the QPU and the gate applied right away, allowing a classical program to interact with the QPU. Several methods have been implemented by OpenQASM 3 (see [9], Section 2.4), ProjectQ (see [33], Section 4.1.2) and Q# (see examples from Q# documentation) to support the *dynamic interaction* feature.

All other frameworks listed in Table 1 rely on a circuit structure to describe a quantum computation, so do not implement the *dynamic interaction* feature.

Scalability *Hybrid quantum-HPC frameworks should support algorithms with arbitrary big number of qubits, or instructions.*

The circuit structure, used by most of today's quantum frameworks, list all quantum operations that will occur during a computation. This structure does not scale

well as the number of qubits continues to increase. OpenQASM 3 [9] and ProjectQ [33, 18] rely on scalable structures (see *dynamic interaction* paragraph) to describe a quantum computation.

Q# [36] is the only remaining framework which does not rely on a quantum circuit. Nevertheless, due to the lack of information concerning the internal structure used by this framework, it is not possible to verify whether this framework support the *scalability* feature or not.

Typing   *Quantum registers are typed to simplify the manipulation of huge structures.*

Typing registers add a new level of abstraction to simplify the manipulation of large data structures, simplifying the design of large scale quantum algorithms. NISQ frameworks focusing on short-term algorithms, they often do not implement this feature. As a consequence, only SILQ [5] and myQLM [13] implement the *typing* feature.

Reversibility   *Pure quantum operations are reversible.*

The *reversibility* feature is key to simplify the development of quantum algorithms. This feature is already well integrated in existing quantum frameworks. Scaffold [21] is the only framework listed in Table 1 which does not fulfill this requirement (reversing a routine in Scaffold implies to rewrite the routine by hand, as shown in their tutorials).

Controllability   *Pure quantum operations are controllable.*

To simplify the development of quantum-HPC algorithms, a quantum routine should be controllable by one or several qubits. All the programming frameworks listed in Table 1 implement the *controllability* feature, by providing either a control function / method, or by relying on a *if* statement.

Safe uncomputation   *Quantum register can be reset to $|0\rangle$ state without measurement.*

The safe uncomputation concept has been introduced by Quipper [17] and embraced by SILQ [5]. Additional frameworks like CUDA Quantum [11], myQLM [13], ProjectQ [33], or Q# do not full-fill this requirement but provides some uncomputation features that enable to uncompute safely a quantum register.

B.2   Classical frameworks

Locality   *A hybrid framework provides tools to control code and memory locality.*

CUDA   Any function defined in CUDA [30] is, by default, executed on the Host. Some function attribute like _ _ *Host* _ _ or _ _ *device* _ _ can be used to specify if a function should be executed on the Host or on the GPU. A Host function is only callable from the Host, and a device function is only callable from the device. The _ _ *global* _ _ function attribute is used to define a kernel, which is a function

executed on the device but callable from the Host. By using these attributes, one can control the code locality.

CUDA is based on C++. Then, any C++ function allocating memory on the Host is available in CUDA. Moreover, CUDA provides some functions, like *cudaMalloc*, to allocate memory on the device. By using C++ and CUDA functions, one can control the memory locality.

OpenCL   An OpenCL-based program is developed using a classical programming language (like C) and OpenCL [34]. Any function defined in the classical language is executed on the Host, and any function written in OpenCL is executed on the GPU. OpenCL defines the *kernel* function attribute to define a kernel, which is a function executed on the device but callable from the classical programming language. By using these two programming languages, one can control the code locality.

OpenCL provides also a library to allocate memory on the GPU, using a classical programming language, to control the memory locality.

OpenMP   OpenMP [10, 24] extends C++. Any C++ function is executed on the Host. OpenMP relies on the *omp target* pragma directive to define a kernel, which is a piece of code executed on the device. By using this pragma directive, one can control the code locality.

OpenMP does not provide tools to directly allocate memory on the GPU. Nevertheless, the *omp target* pragma directive can move data from the Host to the device (and vice-versa), to control the memory locality.

Dynamic interaction   *Classical memory and GPU memory can be allocated on a GPU and manipulated from the Host, at runtime.*

As shown above, CUDA [30], OpenCL [34], and OpenMP [10, 24] provide tools to allocate memory at runtime. This memory can be updated at runtime like any Host variable (the GPU being connected using a PCIe link, the GPU memory is mapped and accessible from the Host).
Moreover, kernels can be started at runtime, fulfilling the *dynamic interaction* feature.

Scalability   *Hybrid CPU-GPU frameworks should support algorithms with arbitrary memory size, or instructions.*

CUDA [30], OpenCL [34], and OpenMP [10, 24] are used in production to solve complex problems requiring the manipulation of large amounts of data. The *scalability* feature is already proven by practice.

Typing   *GPU registers are typed to simplify the manipulation of huge structures.*

CUDA [30], and OpenMP [10, 24] are based on the C++ language. Any C++ class or type can be used on the GPU using these frameworks.

OpenCL [34] is based on C language. Any C type is already defined in OpenCL.

C   Evaluation of Q-Pragma

This paper provides a first evaluation of Q-Pragma compared to Q# [36] and SILQ [5]. This comparison shows that implementing algorithms using Q-Pragma requires less lines of code, in average. Table 2 summarizes the code length, and the number of quantum primitives / annotations needed to solve various problems defined for the Microsoft Q# coding contests (Summer 2018 [27] and Winter 2019 [28]). This evaluation has already be done in [5] for both Q# and SILQ.

		Summer 2018														
		A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	D1	D2	D3	E1	E2
Q#	Lines	9	12	32	24	12	16	9	19	11	28	11	15	9	23	21
	QPA	1	2	4	6	1	1	5	14	2	3	1	4	3	8	3
SILQ	lines	5	6	12	12	3	9	4	5	3	7	7	7	7	7	5
	QPA	2	5	5	7	2	2	3	5	2	5	3	3	1	9	5
Q-Pragma	Lines	3+1	6+1	8+1	9+1	4+3	5+3	4+3	9+3	4+3	9+3	5+1	5+1	6+0	8+3	9+4
	QPA	2+0	3+0	4+0	2+1	1+0	1+0	2+1	3+1	2+1	3+1	1+0	1+0	1+0	5+1	3+1

(a) Summer 2018 comparison

		Winter 2019											
		A1	A2	B1	B2	C1	C2	C3	D1	D2	D3	D4	D5
Q#	Lines	3	20	21	30	18	27	19	3	12	5	21	10
	QPA	1	8	11	14	6	10	9	1	8	5	11	9
SILQ	lines	10	10	17	15	7	11	7	4	15	18	17	15
	QPA	4	8	12	12	2	2	2	2	5	7	7	8
Q-Pragma	Lines	6+0	8+1	17+4	12+3	10+1	9+1	7+1	4+1	7+1	6+1	7+1	9+0
	QPA	4+0	2+0	10+1	8+1	4+0	1+0	2+0	2+0	3+0	4+0	4+0	7+0

(b) Winter 2019 comparison

		Mean
Q#	Lines	16.30
	QPA	5.59
SILQ	lines	9.07
	QPA	4.81
Q-Pragma	Lines	7.26+1.67
	QPA	3.15+0.33

(c) Mean

Table 2: Evaluation of Q-Pragma compared to SILQ and Q#, using algorithms from Q# coding contests (Summer 2018 [27] and Winter 2019 [28]). The comparison is based on the code length, and the number of quantum primitives / annotations (QPA) used. For Q-Pragma, the code has been implemented on the Host or on the QPU: the "+" symbol shows the figures for both the Host and the QPU implementation (e.g. $17+4$ means that the Host implementation requires 17 lines of code, while the QPU implementation requires $17 + 4 = 21$ lines of code). The green cells show the lowest value for each column.

To ensure a fair evaluation, the criteria defined in [5] (Appendix H) have been reused, and any unreadable code-shortening transformation have been avoided. The Q-Pragma primitives and annotations are:

- quantum usual gates (H, X, CNOT ...) and their derivatives (*wall*, *ctrl*, *dag* and *ctrl_dag*),
- measurements and resets,
- pragma directives.

Quantum registers are not considered as a primitive / annotation, matching a criteria defined in [5] for Q# (qubits are not considered as a primitive / annotation). According to this protocol, Table 2 shows that, in average, Q-Pragma implementations are shorter, and use less quantum primitives / annotations than Q# or SILQ. However, the readability of the code is key, to ensure maintainability. In the following, some implementations are provided. Q# implementations are available in [27] and [28] while some of SILQ implementations are available on their website (https://silq.ethz.ch/examples).

C.1   Creating a superposition (Summer 2018 - A2)

The goal is to generate the following quantum state on n qubits:

$$\frac{1}{\sqrt{2}} \left(|0\rangle + |b\rangle\right),$$

where $b \in \{0,1\}^n$ and $b_0 = 1$, so b can also be considered as an integer such that $1 \leq b < 2^n$.

```cpp
#include "qpragma.h"

using namespace qpragma;

#pragma quantum routine (uint64_t bstate)
template <uint64_t SIZE>
void solve(const qbool & head,
           const quint_t<SIZE> & tail) {
    H(head);

    #pragma quantum ctrl(head)
    tail += (bstate >> 1);
}
```

Listing 20: Implementation of $\frac{1}{\sqrt{2}} \left(|0\rangle + |b\rangle\right)$ state using Q-Pragma

Listing 20 shows a solution for this problem using Q-Pragma. This example highlights how typing registers simplifies the code development. The quantum register is splitted in two (*head* and *tail*). The variable b being an integer, it can be added to the *tail* register, as this register is a quantum integer. This idea not only shortens the code, but also makes it clearer. In this example, the first qubit is initialized with the $|+\rangle$ state, and then controls the operation updating *tail* in-place: this naturally creates the superposition of the states $|0\rangle |0...0\rangle$ and $|1\rangle |b_1...b_{n-1}\rangle$. Because $b_0 = 1$, this routine creates the superposition $|0\rangle$ and $|b\rangle$. Note that this routine can be called on a unique quantum register, as this routine is not *typed*.

C.2   Creating a W state (Summer 2018 - A4)

The goal is to generate a W state on N qubits (where N is a power of 2):

$$W_N = \frac{1}{\sqrt{N}} \left(|100\cdots 0\rangle + |010\cdots 0\rangle + \cdots + |000\cdots 1\rangle\right)$$

```
1  #include "qpragma.h"
2
3  using namespace qpragma;
4
5  // In this sample of code, LOG is expected
6  // to be equal to log2(N)
7  #pragma quantum routine
8  template <uint64_t LOG, uint64_t SIZE = (1 << LOG)>
9  void solve(const quint_t<SIZE> & qreg) {
10     quint_t<SIZE> anc;
11     wall::H<LOG>(anc);
12
13     for (uint64_t idx = 0 ; idx < SIZE ; ++idx) {
14         qreg[idx] ^= (anc == idx);
15
16         #pragma quantum ctrl (qreg == 1 << idx)
17         anc -= idx;
18     }
19 }
```

Listing 21: W state implementation using Q-Pragma

In Listing 21, the creation of the W state relies on an array of ancilla qubits initialized with a uniform superposition. These ancilla qubits represent all the possible indexes for the 1 in the final W state. At the end of the routine, these ancilla qubits need to be reset to state $|0\rangle$. To do so, the routine implements the following steps:

- Create state $\sum |i\rangle |0\rangle$ by allocating the *anc* register and applying a wall of Hadamard gate on it.
- Perform the transformation $|i\rangle |0\rangle \rightarrow |i\rangle |2^i\rangle$ (this is done by the first line of the *for* loop).
- Implement the transformation $|i\rangle |2^i\rangle \rightarrow |0\rangle |2^i\rangle$ to reset the *anc* register to $|0\rangle$ (done by the *pragma quantum ctrl* in the *for* loop).

These steps are used to create the state $\sum |0\rangle |2^i\rangle$, which correspond to the W state. The two last steps can be implemented using a single *for* loop (instead of 2), shortening the implementation.

C.3   Pattern of increasing blocks (Winter 2019 - D2)

The goal is to implement a unitary operation which is represented by a square matrix with increasing blocks. For example, a such 3-qubits matrix should have the following shape:

$$\begin{pmatrix} * & & & & & & & \\ & * & & & & & & \\ & & * & * & & & & \\ & & * & * & & & & \\ & & & & * & * & * & * \\ & & & & * & * & * & * \\ & & & & * & * & * & * \\ & & & & * & * & * & * \end{pmatrix}$$

```cpp
#include "qpragma.h"

using namespace qpragma;

#pragma quantum routine
template <uint64_t SIZE>
void solve(const std::array<qbool , SIZE-1> & most,
           const qbool & tail) {
    if constexpr (SIZE > 1UL) {
        wall::H<SIZE - 1>.ctrl(tail, most);
        solve<SIZE - 1>.ctrl((qbool) not tail, most);
    }
}
```

Listing 22: Increasing blocks pattern implementation using Q-Pragma

Listing 22 provides an implementation of the solution given in [28]. This example relies on recursive calls of the *solve* routine. The routine arguments are *most* and *tail*, which corresponds respectively to the whole array except for the last qubit, and the last qubit itself. When the quantum routine *solve⟨n⟩* is called with a quantum array of n-qubits, this array is automatically splitted in the array *most* (of size $n-1$) and the qubit *tail* by Q-Pragma. Consequently, it makes the recursive call easier.