The Journal of Supercomputing (2024) 80:11682–11703
https://doi.org/10.1007/s11227-023-05879-9

QPU integration in OpenCL for heterogeneous programming

Jorge Vázquez-Pérez[1] · César Piñeiro[1] · Juan C. Pichel[1] · Tomás F. Pena[1] · Andrés Gómez[2]

Accepted: 23 December 2023 / Published online: 31 January 2024

Abstract

The integration of quantum processing units (QPUs) in a heterogeneous high-performance computing environment requires solutions that facilitate hybrid classical–quantum programming. Standards such as OpenCL facilitate the programming of heterogeneous environments, consisting of CPUs and hardware accelerators. This study presents an innovative method that incorporates QPU functionality into OpenCL, standardizing quantum processes within classical environments. By leveraging QPUs within OpenCL, hybrid quantum–classical computations can be sped up, impacting domains like cryptography, optimization problems, and quantum chemistry simulations. Using Portable Computing Language (Jääskeläinen et al. in Int J Parallel Program 43(5):752–785, 2014. https://doi.org/10.1007/s10766-014-0320-y) and the Qulacs library (Suzuki et al. in Quantum 5:559, 2021. https://doi.org/10.22331/q-2021-10-06-559), results demonstrate, for instance, the successful execution of Shor's algorithm (Nielsen and Chuang in Quantum computation and quantum information, 10th anniversary edn. Cambridge University Press, Cambridge, 2010), serving as a proof of concept for extending the approach to larger qubit systems and other hybrid quantum–classical algorithms. This integration approach bridges the gap between quantum and classical computing paradigms, paving the way for further optimization and application to a wide range of computational problems.

Keywords QPU · Hybrid programming · OpenCL · Qulacs · PoCL

1 Introduction

Computation has seen a significant surge in demand over recent years. The rise of disciplines such as big data analysis, artificial intelligence, and automation (related to IoT), among others, has underscored the need to enhance efficiency in both time and energy consumption.

Extended author information available on the last page of the article

This is why hardware acceleration has become a topic of interest in the HPC landscape, with tools arising such as the one employed in this work, portable computing language (PoCL) [1]. This approach leverages specialized computer hardware designed to perform specific functions more adeptly than generic software on a traditional central processing unit (CPU). In essence, tasks that software can handle on a standard CPU can also be managed by custom hardware or a mix of the two. Thus, to amplify computing tasks, one might optimize the software, hardware, or both. While software improvements can speed up development and simplify updates, they could introduce computational overhead. In contrast, a hardware-centric approach can yield faster processing, reduced power usage, and superior parallel processing. Quantum computing, in particular, and all its software tools (as the Qulacs library employed in this article [2]) aim to transform the latter by tapping into quantum mechanics principles to process data in ways conventional computers cannot, as algorithms such as Shor's have shown [3], positioning it as a leading accelerator in today's tech landscape.

Integrating this new computational paradigm as an accelerator into the established HPC ecosystem poses significant challenges. This work will outline various strategies. Currently, quantum computing and HPC integrations often involve a QPU communicating with a local system online. While direct, this setup faces speed and task handling challenges. A more sophisticated architecture proposes connecting individual computing nodes directly to a QPU, enabling swift communication and improved parallel processing—a more suitable approach for accelerator devices [4]. As technology advances, compatibility with other accelerators like GPUs and FPGAs becomes increasingly crucial.

The motivation behind this work is to explore the integration of QPUs in contemporary HPC environments, specifically by merging OpenCL [5] with quantum computing. OpenCL is an open standard for parallel programming of heterogeneous systems, allowing developers to harness the power of both CPUs and various accelerators for computational tasks. Given the early stages of QPUs, full integration into HPC environments remains elusive. However, through simulations, this work seeks to provide a glimpse into how quantum computers might function as accelerators in present-day HPC nodes, shedding light on future advancements in this domain. These proof-of-concept explorations form a central component of our discussion.

In summary, by combining OpenCL and quantum simulations, the aim is to envision the role of quantum computers as accelerators within today's HPC systems. This research explores the challenges and prospects of integrating quantum computing into the current HPC landscape and lays the groundwork for future technological intersections and advancements.

The rest of this article is organized as follows. In Sect. 2, the current and anticipated roles of quantum computers within contemporary HPC environments are elucidated. Subsequently, in Sect. 3, the heterogeneous platform paradigm is introduced to provide context for the synergy between OpenCL and quantum computing. Central to this research, Sect. 4 presents a proof of concept illustrating the integration of QPUs within the OpenCL standard, using simulations of the quantum phase estimation (QPE) circuit and Shor's algorithm circuit as examples. The paper concludes with an evaluation of the results and a discussion on future directions.

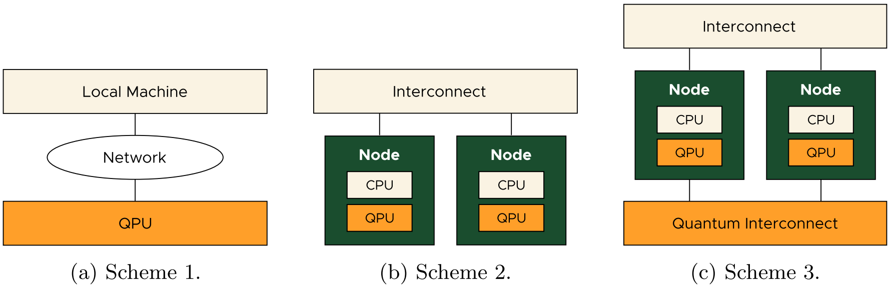
2 Quantum computers in high-performance computing

2.1 QPUs as accelerators

A lot of speculation has been made in the last few years about quantum computation, from the quantum supremacy statement veracity to the feasibility of developing a quantum computer that yields an acceptable error rate. But one thing is certain: quantum computation is not foreseen to take the place of classical computation, at least not in the near future. There are several reasons [6, 7] that justify this statement:

1. *Memory* One of the key differences between classical computation and quantum computation is the decoherence of data. While classical computation is able to store data for arbitrarily long time, quantum computation and, therefore, qubits have small time windows where information is not lost, i.e., decoherence does not destroy information [8]. Of course, time units depend strictly on the specific technology of the quantum computer. Some researchers have theorized about quantum memory random access [9], but this kind of theoretical models are far away from being feasible.
2. *Approximate computing* Quantum computing is not able to yield an approximate result of a problem just as classical computers do. Classical computers can give an approximate result of a problem with certainty, contrary to quantum computers which give an approximate result with high probability. The stochastic nature of quantum computers must never be forgotten and plays a really important part in analyzing the quantum algorithms and why they provide an exponential speed up.
3. *Software* Software is essential for quantum computers [7]. Virtually all quantum computation models necessitate the use of classical control. This is due to the challenges in achieving dependable measurements and fault-tolerant computations with quantum devices that are prone to errors, especially in the absence of a reliable method to sequence operations and make decisions for error correction. As a result, all recognized software toolchains operate under the assumption of a *quantum co-processor* model. In this model, several classical devices are responsible of managing and directing the operations of the QPU, similar to how a classical microprocessor today manages and interacts with the GPU on a graphics co-processor card. While actual implementations may vary, software developers can conceptualize this as a single microprocessor delivering instructions to the quantum co-processor in every cycle.

In addition, quantum computers are inherently exceptional operating on some specific tasks in which classical computers are not that efficient. To be more precise, bounded-error quantum polynomial time (BQP) is the class of decision problems solvable by a quantum computer in polynomial time, with an error probability of



(a) Scheme 1. (b) Scheme 2. (c) Scheme 3.

Fig. 1 Different schemes of quantum integration on HPC environments [4]

at most $\frac{1}{3}$ for all instances [3]. On the other hand, in computational complexity theory, P, also known as PTIME or DTIME($nO(1)$), is a fundamental complexity class that contains all decision problems which can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time. It is known that $P \subset$ BQP because it has been proven that every classical circuit can be simulated by a quantum circuit [3]. The contrary is likely false, relying on the fact that $P =$ NP is believed to be false, and quantum computers can solve in polynomial time some NP problems such as integer factorization, discrete logarithms, among others. In some of these problems, the quantum advantage turns out to be exponential in comparison with its classical counterpart.

These two main characteristics, the need of classical computation in order to execute quantum computation and the exponential advantage, are the reasons why quantum computers are considered as **accelerators**, just as if quantum computers were FPGAs or GPUs. This HQCC model, using QPU as an accelerator has its pinnacle example in variational quantum Eigensolver (VQE) [10], which represents a hybrid algorithmic approach that leverages the capabilities of both classical and quantum computing systems to ascertain the ground state of a specified physical system. Initiated with an educated guess or ansatz, the quantum processor is tasked with the computation of the expectation value of the system in relation to an observable, frequently the Hamiltonian. Subsequently, a classical optimizer is employed to refine the initial guess. The underlying principle of this algorithm is rooted in the variational method of quantum mechanics.

2.2 Execution schemes for QPUs

Once the nature and purpose of quantum computers as accelerators are understood, the need to define how these devices should be integrated into an HPC environment becomes important. With this goal in mind, an examination of the current and future approaches of the integration will be presented. Figure 1 will serve as a guideline.

Current approaches emphasize enabling remote access to QPUs via public networks as a significant aspect of their integration into HPC systems, evident in Fig. 1a. This method permits users to utilize quantum resources without being physically present at the quantum hardware location. Nonetheless, it also introduces challenges like ensuring security, managing network latency and fulfilling infrastructure requirements. To address these challenges, various techniques and frameworks, including virtualization, containerization and network protocols, have been implemented. These strategies aim to enable seamless remote access to QPUs and ensure their efficient and secure use within the HPC environment.

Quantum computers are considered among the most promising hardware accelerators. Similar to the utilization of other accelerators, the goal is to deploy them as efficiently and optimally as possible. This mirrors the case with GPUs, which are integrated within the same nodes as CPUs to minimize latency and ensure peak performance. Such an integration goal is also envisioned for QPUs, evident in Fig. 1b. Integration strategies span from networking the quantum device directly with the motherboard to a future scenario where the CPU and quantum processor coexist on the same die. Although on-die integration offers the best performance potential, realizing it requires significant advancements in quantum technologies. An alternative approach involves motherboards with dedicated sockets for quantum accelerators. This provides benefits like minimized latency, board lanes optimized for the necessary bandwidth, and improved access to shared resources such as memory. As the quantum computing field evolves, deeper integration of these components is anticipated to unveil the maximum capabilities of HPC, allowing users to capitalize on the benefits of quantum acceleration [11].

A novel macroarchitecture is also represented in Fig. 1c. It expands upon the distributed design to encompass a distributed quantum computing system. This approach leverages a quantum interconnect, unlocking a multitude of new application possibilities by enabling entangling operations between nodes. However, to be able to achieve these kind of architecture with such a sophisticated communication as entanglement provides imply new requirements and advances that are out of the scope of this work.

3 Integrating QPUs in OpenCL

As previously mentioned, hardware accelerators are being found as a good option to release some pressure from the hectic need of computation that we are living on nowadays. This has various implications, but the main one is: How should these devices be integrated in HPC environments? Extracting the most out of them in terms of computational performance is, obviously, one of the main concerns, but not the only one. There is also a need to define how devices will communicate with the host (the main CPU responsible for handling calls and executing the core program), how portability concerns will addressed, and, most importantly, how programmers and developers will handle these questions.

That is precisely what the OpenCL standard tries to give a response to. OpenCL supports a broad range of applications, from embedded systems to high-performance computing, by providing a low-level, high-performance and portable abstraction. It allows developers to write parallel code using a subset of ISO C99 and offers an intermediate language for parallel execution. With consistent numerical

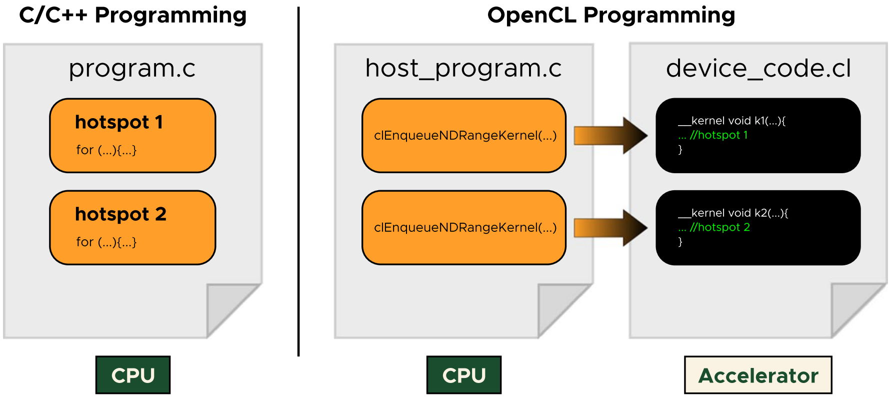


Fig. 2 Comparison between common programming and OpenCL programming models

requirements and seamless integration with APIs like OpenGL, OpenCL serves as the foundation for a parallel computing ecosystem [12]. By empowering developers to harness the potential of heterogeneous processors, OpenCL plays a vital role in emerging interactive graphics applications, blending general parallel compute algorithms with graphics rendering pipelines. Through its core specification, handheld/embedded profile and optional extensions, OpenCL equips software developers to unlock the full capabilities of heterogeneous processing platforms, enabling high-performance applications across various devices. And to be able to do this, OpenCL defines the following main standards:

1. *A communication API* This API is the responsible of establishing and performing communication between the host and the devices. As long as a device implements the necessary functions of the standard, the communication using OpenCL is possible.
2. *A cross-platform programming language* This is the language in which all kernels are going to be written. In the context of OpenCL, a kernel is a small program or function that runs on a computing device like a GPU or CPU. It performs specific tasks in parallel, meaning it can process multiple pieces of data at the same time. The kernel is the core part of the code that gets executed on the hardware to carry out computations. That is why a specification of the language is required in order to be able to execute a kernel in every device considered by, achieving the portability sought by the OpenCL standard. As it has been peeked quickly before, this language is a subset of the ISO C99 language. The restrictive nature of this subset is related to the need of allowing only operations executable in every device considered (CPU, GPU, FPGA…).

OpenCL also defines an intermediate cross-platform language and an extension standards, but these are out of the scope of this work.

To elucidate the performance characteristics of OpenCL programs, Fig. 2 serves as an illustrative guide. In conventional programming paradigms utilizing languages such as C and C++, execution is predominantly CPU-centric, as depicted on the left-hand side of the figure. While some programs exploit the multicore architecture of modern CPUs or even offload certain computational tasks to a GPU through vendor-specific APIs and libraries (e.g., CUDA), this approach diverges from the objectives of OpenCL. OpenCL aims for a more versatile computational model, as represented on the right-hand side of Fig. 2.

On the one hand, the OpenCL's API standardizes the communication protocol, enabling seamless interaction with a variety of computing devices, irrespective of their inherent nature. This abstraction layer erases the need for device-specific considerations in the code, with the exception of the compilation and execution phase done in the host side, where the target device must be explicitly selected. Furthermore, OpenCL's API provides the flexibility to specify the execution environment's characteristics, however, it is pertinent to note that the present discussion will not dig into the general-purpose parallelism capabilities that OpenCL offers, as the field of quantum computing is still in a premature stage with respect to parallelism and high-performance techniques.

On the other hand, code intended for execution on an accelerator must be delineated separately from the host code stream. This separation can be achieved either by placing the accelerator code in a distinct file, as illustrated in Fig. 2, or by embedding it as a string within the host file. The choice between these two approaches is a matter of modularity that the programmer must determine. A non-negotiable aspect, however, is the programming language in which the accelerator code must be written in: OpenCL C, as defined in the previously mentioned standard. It is crucial to highlight that the functions designated for execution on the device are explicitly marked with the `__kernel__` directive, representing the code declaration of the kernel concept.

3.1 OpenCL and quantum computing

Now that the heterogeneous computing paradigm has been presented using OpenCL as a means to standardize ideas, and quantum computation has been defined as a tool to accelerate specific processes where classical computation lacks efficiency, it is time to discuss how these two disciplines can be combined and how each of them can harness its power with the help of the other.

First, the fact that quantum computers are considered as accelerators explains why the use of OpenCL was initially considered. As depicted in Fig. 2, accelerators play a central role in the OpenCL scheme, which recognizes that modern computation should not only have the capability but also the necessity to leverage all available devices to maximize resource efficiency. This perspective aligns with how quantum computers should be viewed: as accelerators. Therefore, integrating them into this computational model represents a significant step toward incorporating quantum computers into our heterogeneous computing environments.

Secondly, the most relevant practical quantum algorithms today are primarily iterative, featuring alternating quantum components where the concept of "quantum supremacy" is applied, along with classical counterparts responsible for optimizing the parameter gates and preparing the circuit for the next execution. These are the variational quantum algorithms [13]. Examples of these type of algorithms are the VQE [10] or the quantum approximate optimization algorithm (QAOA) [14]. For example, this execution model is particularly intriguing for distributing the workload between a GPU, responsible for optimizing the parameters, and a QPU, dedicated to applying quantum acceleration.

Of course, not everything in this combination of schemes is a perfect match. The introduction of quantum computation represents a wholly new paradigm at the computational table, which poses challenges for the portability aspect that OpenCL highly values. There are several reasons for this incompatibility, one of the main ones being that the operations used in quantum computation are not even considered in the OpenCL C language. Moreover, an issue that may not have significant implications in classical computing, such as executing code on an FPGA that was originally designed for a GPU, takes on greater significance when QPUs are introduced into the discussion. Executing classical code on a quantum computer serves little purpose. Quantum algorithms require an entirely different approach compared to classical algorithms, involving distinct programming flows, operations, and more. This is one of the major challenges when integrating quantum computers into the OpenCL framework.

4 Proof of concept: OpenCL for QPUs

In addition to the issues highlighted at the end of the previous section, there is another significant obstacle preventing us from integrating a complete QPU into this scheme: the current lack of fully functional quantum computers. To include a device in the OpenCL scheme, it is essential to possess a comprehensive understanding of the device's architecture, enabling the establishment of compilation rules. Without knowledge of how memory is accessed, how quantum gates are transpiled, which backend the QPU supports, and so forth, it becomes impossible to define compilation procedures

As a result, the option of employing simulated quantum computers becomes a viable choice, similar to its use in other contemporary paradigms that incorporate quantum computation. As a preliminary step in this direction, rather than incorporating a QPU as a CPU with emulation tools, a proof of concept was initiated using a conventional CPU and a kernel code that implements all the necessary functions for quantum simulation.

To accomplish this, the first requirement is the implementation of OpenCL, as it was explained in Sect. 3. OpenCL is a standard that defines a set of communication functions within its API and provides a cross-platform programming language. However, it does not prescribe how these functions should be implemented. In this work, the open-source implementation known as PoCL was utilized, as described in [1]. The version 3.1 of PoCL was the one chosen.

Additionally, because there was a need to simulate quantum circuits within a kernel, a set of functions defining the quantum simulation was necessary to complete the job. There was two options: either implement a custom library following the

OpenCL C language specification or adapt an already existing library to conform to the aforementioned language specifications. The chosen option was the latter due to the availability of the Qulacs library [2]. This library includes a C implementation as well as C++ and Python implementations, providing a solid foundation for enabling quantum simulation within an OpenCL kernel.

4.1 OpenCL restrictions

Numerous difficulties were encountered in this seemingly easy task, as discussed in the previous section, which involved adapting Qulacs to meet the OpenCL C requirements. These difficulties included the following:

1. *Function recursion* This type of operation is allowed in OpenCL C. However, recursion, while technically feasible, imposes a significant computational cost on GPUs. This is due to the need to allocate stack space for thousands of threads and account for recursion during the challenging process of register allocation. Additionally, the issue of thread divergence further complicates matters. When transitioning to FPGA synthesis, the problem becomes insurmountable. Given these circumstances, the most straightforward solution was to completely eliminate recursion. This is precisely what PoCL does; it forbids its use and generates a compilation error.
2. *Standard math libraries* The nature of quantum computing necessitates a significant reliance on linear algebra and complex numbers. Typically, these mathematical constructs are implemented using common C math libraries like `math.h`. However, standard libraries are not permissible in OpenCL due to its prior compilation with the common GCC compiler, which does not adhere to the requirements of the OpenCL C specification. Consequently, custom operations and data types had to be implemented specifically for this work. This included the development of complex number representations and all the operations associated with this data type.
3. *Double pointers* In C, when working with matrices, double pointers are commonly used to access dynamically allocated memory. However, in OpenCL C, the use of double pointers is not allowed due to the complexities that arise when dealing with multi-level pointers across different memory spaces in heterogeneous computing environments. OpenCL is designed to target a wide range of devices with diverse memory architectures, and supporting multi-level pointers could introduce inconsistencies and inefficiencies. By restricting this feature, OpenCL ensures portability, efficient data transfer, and simplified memory management, enabling developers to write optimized code that can run effectively on various hardware architectures. In this work, alternative techniques were employed, such as flattening arrays, to effectively handle multi-dimensional data.
4. *Dynamic memory allocation* Arguably, one of the most crucial functionalities of the C language, custom dynamic memory allocation, is prohibited in OpenCL C. This prohibition stems from several compelling reasons, including the necessity for deterministic memory access patterns, concerns about potential memory

management overhead, incompatibility with heterogeneous memory architectures, and the inherent risk of data races and synchronization issues. OpenCL kernels are meticulously designed to execute in a parallel and deterministic manner across multiple processing elements, demanding predictable memory access patterns at compile time. The introduction of dynamic memory allocation would introduce unpredictability and hinder memory access optimizations, potentially compromising performance. Moreover, efficiently managing dynamic memory in the context of heterogeneous environments with varying memory architectures, as previously encountered with double pointers, would prove exceptionally challenging. The associated bookkeeping overhead and synchronization requirements in multi-threaded execution could further exacerbate performance concerns.

Instead, OpenCL promotes explicit memory management through statically allocated memory objects like buffers and images, ensuring both predictability and efficiency in memory access. However, for the specific requirements of this project, dynamic memory allocation was an imperative despite the efficiency trade-off. To implement dynamic memory management, the KMA model was adopted, as described in [15]. This model serves as a dynamic memory manager explicitly designed for OpenCL, accommodating the unique demands of our computational environment.

5. *Specification of address space* There was also the need of changing address space as needed to fit in the `constant`, `private` and `global` scheme that OpenCL requires. This would not be needed if the generic address space was implemented, but in the version of PoCL used (3.1) the generic address space was not implemented and so pertinent address space specification was required which, because of the classical C nature of Qulacs, was not defined.

4.2 Examples

To comprehensively represent our research, this work will present two distinct examples. Firstly, a simulation of the quantum phase estimation (QPE) [3] algorithm will be conducted due to its simplicity and applicability, making it an ideal introductory algorithm for simulations. Subsequently, the Shor algorithm [16] will be introduced in a manner analogous to the QPE. The objective of this comparison is to illustrate that even more intricate algorithms can be effectively simulated using the Qulacs library adapted for the OpenCL C language.

It should be noted that the algorithms will not be fully implemented within the scope of this paper. Specifically, with regard to the QPE, this research demonstrates the effective simulation of the circuit but does not delve deeply into post-processing of the data to estimate the phase. Instead, an illustrative example will be provided to demonstrate the accuracy of the algorithm's results. Similarly, in the case of Shor's algorithm, the primary focus of this study is not on the factorization of integers but rather on ensuring the correct execution and outcomes of the quantum component, namely the accurate estimation of the period.

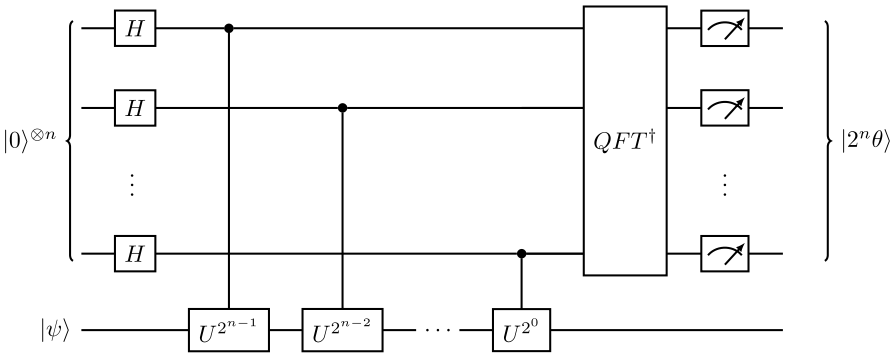


Fig. 3 QPE circuit

4.2.1 QPE algorithm

The QPE algorithm is one of the most well-known quantum algorithms [3]. Given a unitary operator U, the algorithm estimates θ in $U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$. So, basically, it estimates the phase of an eigenvalue of a given gate. The circuit responsible of this estimation can be seen in Fig. 3. The algorithm employs a two-register quantum system: one to hold the phase estimation and another initialized to the eigenstate $|\psi\rangle$. It applies controlled-U operations, with U raised to exponentially increasing powers based on the qubit position, creating a superposition of phase states. An inverse quantum Fourier transform (QFT†) on the first register converts these phases into a binary representation that can be measured to yield an estimate of θ, as it will be shown at the end of this section. The precision of QPE and its probability of success improve with more qubits, showcasing the algorithm's ability to leverage quantum mechanics for tasks that challenge classical computers. The mathematical proof of the result will be omitted, as it falls outside the scope of the article.

Regarding the gate employed, it should be mentioned that the job was executed employing a phase gate P as the U gate of the circuit for simplicity purposes, because using $|1\rangle$ as the eigenvector for the algorithm satisfies $P|1\rangle = e^{i\theta}|1\rangle$, as it shows a simple application of the gate to the state:

$$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = e^{i\theta} \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

In Fig. 4, the host code is depicted as a standard OpenCL code.[1] This OpenCL program begins by establishing its operational context (lines 1–4), searching for a compatible device using the `search_device` function, and setting up a command queue for executing instructions. It then prepares for computation by incorporating

[1] This code and all the following ones can be found in https://github.com/jorgevazquezperez/OpenCLQPU.

```
 1  int main(int argc, char *argv[]) {
 2      cl::Device device = search_device();
 3      cl::Context context(device);
 4      cl::CommandQueue queue(context, device);
 5  
 6      Sources src;
 7      add_kernel_files(src, "../../../", "../qpe.cl");
 8      cl::Program program(context, src);
 9      const char *includes = "-I../../../src/malloc";
10      program.build(includes);
11  
12      size_t n_counts = 3;
13      size_t count_dim = pow(2, n_counts);
14      COUNT counts[count_dim];
15  
16      cl::Buffer heap(kma_create(device.get(), context.get(), queue.get(),
            program.get(), 2048));
17      cl::Buffer countsbuf(context, CL_MEM_READ_WRITE, sizeof(COUNT) *
            count_dim);
18  
19      cl::Kernel qpe(program, "qpe", nullptr);
20      qpe.setArg(0, heap);
21      qpe.setArg(1, countsbuf);
22      qpe.setArg(2, &n_counts);
23  
24      queue.enqueueWriteBuffer(countsbuf, CL_TRUE, 0, sizeof(COUNT) * count_dim
            , counts);
25      queue.enqueueNDRangeKernel(qpe, cl::NDRange(1), cl::NDRange(1));
26      queue.enqueueReadBuffer(countsbuf, CL_TRUE, 0, sizeof(COUNT) * count_dim,
             counts);
27      queue.finish();
28  
29      print_probabilities(counts, count_dim);
30      return 0;
31  }
```

Fig. 4 Host code for QPE

kernel files which include the modified Qulacs library (line 7) and including necessary external sources as it is the allocation operations (line 9), followed by the compilation of the program with the aforementioned resources (line 10). It specifies the dimension for the number of qubits n that are going to be used to estimate the phase using the variable `n_counts`, which is then used to calculate the dimension over the $\mathbb{R}$ space, i.e., 2^n, and this dimension, in turn, is used for defining the buffer in which the measured results are going to be stored (lines 12–14). Subsequently, the program allocates memory buffers on the device for memory allocation, with the `heap` buffer, and result storage, with the `countsbuf` buffer (lines 16–17). The QPE kernel, which performs the main computation, has its arguments set (lines 19–22), including the allocated buffers and the size parameter. Execution is managed through a series of queue commands that write to the buffer, initiate the kernel, and read back the results (lines 24–27). To ensure all processes are complete, it synchronizes with `queue.finish()` (line 28). Finally, the program processes and

```
 1  __kernel void qpe(HEAP heap, COUNT __global* counts, size_t __global *
        n_counts) {
 2      int dim = pow(2, *n_counts + 1);
 3      CTYPE __global *state = allocate_quantum_state(dim, heap);
 4      initialize_quantum_state(state, dim);
 5
 6      int i, j, exp;
 7      double phase_input = 1./2;
 8      CTYPE matrix[4] = {{1, 0}, {0, 0}, {0, 0}, {0, 0}};
 9
10      for(i = 0; i<3; i++) {
11          H_gate(i, state, dim);
12      }
13
14      X_gate(3, state, dim);
15
16      for(i = 0; i<3; i++) {
17          exp = 1 << (3-i-1);
18          matrix[3] = double_to_complex(cos(2*PI*phase_input*exp), sin(2*PI*
                phase_input*exp));
19          single_qubit_control_single_qubit_dense_matrix_gate(i, 1, 3, matrix,
                state, dim);
20      }
21
22      inverse_qft(0, 3, 0, state, dim);
23
24      UINT __global *index_list =
25          (UINT __global *)malloc(heap, sizeof(UINT) * (*n_counts));
26      for(int i=0; i<(*n_counts); i++){
27          index_list[i] = i;
28      }
29      get_probabilities(index_list, *n_counts, state, dim, counts);
30  }
```

Fig. 5 Kernel code for QPE

displays the results with the `print_probabilities` function (line 29), before terminating with a return statement (line 30).

In examining the kernel code, Fig. 5 illustrates the resemblance between the kernel program and a conventional Qulacs program scripted in C. The adapted Qulacs functions are highlighted in italics in the code. One can see that the kernel takes in the heap, a global counts array and the `n_counts` variable as inputs (line 1), being coherent with the arguments set in the host code in Fig. 4. The kernel begins by setting the dimensionality of the quantum state as a power of two based on the number of counts (line 3) and then allocates and initializes a quantum state within the provided heap memory (lines 3–4) using the adapted Qulacs functions `allocate_quantum_state` and `initialize_quantum_state`, respectively. Subsequently, the kernel defines a phase input and a matrix for quantum operations (lines 6–8). It applies a Hadamard gate `H_gate` to the first three qubits (lines 10–12), setting the state into superposition. It then performs an operation with an `X_gate` on the fourth qubit (line 14), to prepare the state for the subsequent controlled

Table 1 QPE results obtained with the possible measured values accompanied by their corresponding probability

Values	Probability
000	3.7494e−33
001	4.39269e−33
010	7.4988e−33
011	2.56025e−32
100	1.00000
101	2.56025e−32
110	7.4988e−33
111	4.39269e−33

operations, as it was explained at the beginning of this section. The kernel continues with a loop applying a controlled gate, where it adjusts the phase based on the iteration (lines 16–19), using the adapted `single_qubit_control_single_qubit_dense_matrix_gate` function from Qulacs. After completing the controlled operations, it applies an inverse quantum Fourier transform `inverse_qft` to the qubits first `n_count` qubits (line 22), which is the final step in QPE to extract the phase information. Finally, the kernel prepares an index list and allocates memory for it on the heap (lines 24–26). It populates this list with indices (line 27), which is then used for measuring the quantum state. The `get_probabilities` function (line 29) then extracts the probabilities of the quantum state's outcomes, which in a normal appliance of the QPE algorithm would be an actual measurement, but, for illustrative purposes, this work chooses to get the actual probabilities of the possible measurements.

In this example, as displayed in Fig. 5, the value of the phase was $\theta = \frac{1}{2}$. Given the inherent characteristics of the algorithm, the phase can be estimated using the relationship $\theta = \frac{value}{2^n}$. Here, *value* denotes the quantity with the maximal probability; while, n represents the precision used, quantified by the number of qubits read. Subsequent to the execution of the kernel on the device and the retrieval of the buffer containing the outcomes, as illustrated in Fig. 4 and labeled as `countsbuf`, the results shown in Table 1 were obtained.

And so, the most probable value is 4 (`100` in binary format), and then $\theta = \frac{4}{8} = \frac{1}{2}$. In the given scenario, the outcome is evident as the phase, denoted by θ, adheres to the relationship $\theta = \frac{value}{2^n}$. Conversely, in distinct scenarios, the probability tends to converge around the closest $\frac{value}{2^n}$ corresponding to the actual phase. Those scenarios are not considered on the post-processing of the data for being out of the scope of the program, as it was previously mentioned at the beginning of the Sect. 4.2. All in all, the QPE algorithm has been successfully simulated utilizing the OpenCL framework.

4.2.2 Shor's algorithm

Conversely, Shor's algorithm [16] exemplifies the pinnacle of quantum computational capabilities. It entails an intricate integration of classical techniques, such

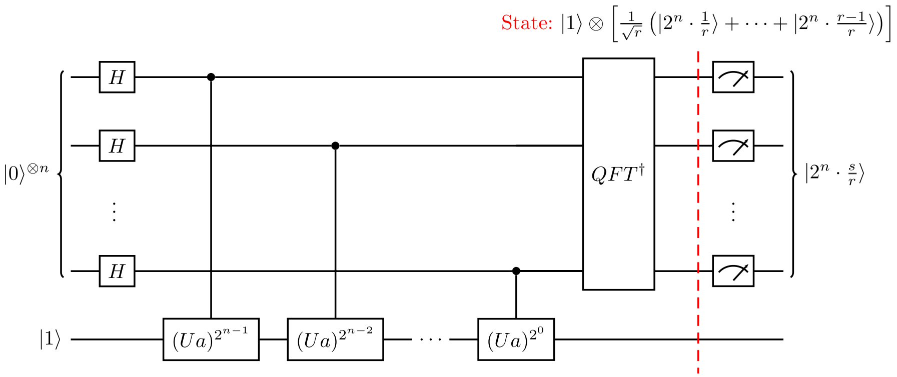

State: $|1\rangle \otimes \left[\frac{1}{\sqrt{r}}\left(|2^n \cdot \frac{1}{r}\rangle + \cdots + |2^n \cdot \frac{r-1}{r}\rangle\right)\right]$
$|0\rangle^{\otimes n}$
H
$|1\rangle$
$(Ua)^{2^{n-1}}$
$(Ua)^{2^{n-2}}$
$(Ua)^{2^0}$
$QFT^\dagger$
$|2^n \cdot \frac{s}{r}\rangle$

Fig. 6 Shor's circuit (color figure online)

as the continued fraction algorithm, with a sophisticated quantum circuit. In fact, Shor's algorithm is a seminal quantum computational method designed for efficiently factoring large numbers, a problem traditionally considered challenging for classical computers. By harnessing quantum parallelism, this algorithm can determine the factors of a composite number in polynomial time, thereby showing a potential threat to classical encryption systems, particularly RSA, which rely on the difficulty of factoring large numbers.

More specifically, Shor's algorithm centers around the modular exponential function $f(x) = a^x \mod N$, where a is randomly chosen and N is the number wished to be factored. The algorithm proceeds as follows:

1. *Computation of the greatest common divisor (GCD)* By using the Euclidean algorithm, the algorithm finds the GCD of a and N. If the GCD is not 1, it has found a factor, and ends. If not, the algorithm moves to the second step.
2. *QPE* This is the only quantum portion of the algorithm. It uses the quantum circuit presented in Fig. 6 to evaluate the modular exponential function across a superposition of states. It can be noticed that it is the same circuit of Fig. 3 but with $|\psi\rangle = |1\rangle$ and with the U operator satisfying:

   $$U|x\rangle = |ax \mod N\rangle \equiv Ua|x\rangle.$$

   It must be remarked that the randomly chosen value a is fed into the U gate of the QPE algorithm. As it is presented in Fig. 6, the final state before the measurements (highlighted with a red dashed line) will be of the form:

   $$\frac{1}{\sqrt{r}}\left(|2^n \cdot \frac{1}{r}\rangle + \cdots + |2^n \cdot \frac{r-1}{r}\rangle\right)$$

   and so, after measurement, the state obtained is of the form $|2^n \cdot \frac{s}{r}\rangle$, with $0 < s < r$, getting the phase $\theta = \frac{s}{r}$.
3. *Classical post-processing with continued fractions* Once the phase θ is extracted from the quantum circuit, classical algorithms come back into play. The period r is

not directly observed but inferred from the output phase using the continued fraction algorithm. This classical algorithm is used to analyze the output and deduce the period from the quantum measurement. If r is odd or if $a^{r/2} \equiv -1 \mod N$, the algorithm moves back to the first step. Otherwise, it proceeds to the next step.

4. *Factorization* With the period r in hand satisfying the necessary conditions (it is even and $a^{r/2} \not\equiv -1 \mod N$), the algorithm uses classical computations (like Euclid's algorithm for computing the GCD) to factor N. Specifically, the factors of N can often be found by taking the GCD of N with $a^{r/2} \pm 1$ and, if they are not, the algorithm restarts.

In the current study, it is performed a simulation of a circuit comprising three counting qubits and four auxiliary qubits, totaling seven qubits. A more generalized algorithm implementation would demand excessive computational resources and time, diverging from the intended scope of this work. It should be noted that counting qubits are specifically employed to find the period via the continued fraction algorithm.

After presenting Shor's algorithm, the discussion shifts to the main objectives of the study. In examining Fig. 7, one observes minimum alterations in the host program relative to the QPE implementation. Notably, the kernel name is shifted from the QPE one to the Shor one and the post-processing of the data returned by the OpenCL kernel now employs the continued fraction algorithm to discern the period which is done in the `print_guess` function. But, excluding that part, the code maintains the same structure and calls as the one for the QPE.

Pertaining to the kernel code, Fig. 8 delineates the circuit implementation utilizing the same adapted Qulacs library as in the QPE instance and, again, the calls to the adapted function are highlighted in italics. As it happened with the host code for QPE and Shor's, both kernels are really similar. The only notable change is the use of the gate facilitating modular multiplication instead of the phase gate employed in the QPE circuit. This gate is denoted as `c_amod15` in the code (line 14), and is a bespoke construction leveraging the capabilities of the Qulacs library. This library allows for the integration of any valid gate operation, provided its matrix is unitary, by handing several tools to the programmer, which were used in this work to achieve the desired functionality. It should be clarified that the study adopts a constrained version of Shor's algorithm, with a predetermined value of $N = 15$ (hence the name of the gate `c_amod15`). This selection is predicated on fostering clarity and facilitating a comparison with results found in the Qiskit [17] textbook, where an analogous circuit is employed. As previously explained, this study aims to illustrate the efficient integration of quantum processes into heterogeneous platforms. Consequently, attempting to simulate the entirety of Shor's algorithm, given its inherent complexity, could compromise the primary objective.

The possible outcomes of the algorithm are delineated in Table 2. Observing Table 2a, one notes that states `000`, `010`, `100`, and `110` exhibit identical probability distributions. This suggests that each corresponding guess bears an equivalent likelihood of selection. Taking into account that the correct period is $r = 4$, one can observe in Table 2b that the probability of selecting a correct response

```
int main(int argc, char *argv[]) {
    cl::Device device = search_device();
    cl::Context context(device);
    cl::CommandQueue queue(context, device);

    Sources src;
    add_kernel_files(src, "../../../", "../shor.cl");
    cl::Program program(context, src);

    const char *includes = "-I../../../src/malloc";
    program.build(includes);

    size_t n_counts = 3;
    size_t count_dim = pow(2, n_counts);
    COUNT counts[count_dim];

    cl::Buffer heap(kma_create(device.get(), context.get(), queue.get(), program.get(), 2048));
    cl::Buffer countsbuf(context, CL_MEM_READ_WRITE, sizeof(COUNT) * count_dim);

    cl::Kernel shor(program, "shor", nullptr);
    shor.setArg(0, heap);
    shor.setArg(1, countsbuf);
    shor.setArg(2, &n_counts);

    queue.enqueueWriteBuffer(countsbuf, CL_TRUE, 0, sizeof(COUNT) * count_dim, counts);
    queue.enqueueNDRangeKernel(shor, cl::NDRange(1), cl::NDRange(1));
    queue.enqueueReadBuffer(countsbuf, CL_TRUE, 0, sizeof(COUNT) * count_dim, counts);
    queue.finish();

    print_probabilities(counts, count_dim);
    print_guess(counts, n_counts, count_dim);
    return 0;
}
```

Fig. 7 Host code for Shor's algorithm

stands at 0.5, evidenced by measurements `010` or `110`. Conversely, measurements `000` or `100` would not yield accurate results: the former due to $s = 0$ and the latter because it produces r, albeit a factor, given $s = 2$. As was said when the algorithm was explained, if instead of `010` or `110`, the algorithm was to yield `000` or `100` then it would be repeated until a correct period is found. The remaining step of the algorithm, which is finding the actual factor is left for completion as it is a classical process with no interest for this work.

When juxtaposing these findings with those exposed in the Qiskit textbook, a clear congruence emerges. This consistency not only underscores the accuracy of the results but, akin to the QPE example, substantiates the efficacy of integrating quantum algorithms into the OpenCL pipeline, which was the main objective all along this study.

```
__kernel void shor(HEAP heap, COUNT __global* counts, size_t __global *
    n_counts) {
    int a = 7, i;
    UINT dim = pow(2, *n_counts + 4);
    CTYPE __global *state = allocate_quantum_state(dim, heap);
     initialize_quantum_state(state, dim);

    for(i = 0; i < *n_counts; i++){
        H_gate(i, state, dim);
    }

    X_gate(*n_counts, state, dim);

    for(i = 0; i < *n_counts; i++){
        c_amod15(i, state, a, pow(2, i), *n_counts, dim, heap);
    }

    inverse_qft(0, *n_counts, 1, state, dim);

    UINT __global *index_list =
        (UINT __global *)malloc(heap, sizeof(UINT) * (*n_counts));
    for(int i=0; i<(*n_counts); i++){
        index_list[i] = i;
    }
    get_probabilities(index_list, *n_counts, state, dim, counts);
}
```

Fig. 8 Kernel code for Shor's algorithm

Table 2 The results show the possible measured values and their probability while the period guess shows which fraction and guess for r correspond to each phase obtained

Results		Period guess		
Values	Probability	Phase	Fraction	Guess for r
000	0.25	0	$\frac{0}{1}$	1
001	0	0.25	$\frac{1}{4}$	4
010	0.25	0.5	$\frac{1}{2}$	2
011	0	0.75	$\frac{3}{4}$	4
100	0.25			
101	0			
110	0.25			
111	0			

5 Conclusions

In conclusion, this work explored the integration of a contemporary computational paradigm—heterogeneous platform computing—with quantum computation. Such an integration is reflective of ongoing efforts to harness the strengths of both quantum and classical computation. The overarching aim, as elaborated in this paper, is the seamless synergy between classical and quantum computation, allocating tasks based on the nature and strengths of each.

As an initial endeavor, this work aims to give a glimpse of the incorporation of quantum computation within these heterogeneous frameworks. The objective extends beyond simply adapting quantum computation to fit these schemes; it seeks to ensure that existing frameworks and libraries recognize and account for quantum computers as viable computational entities. This integration is not merely theoretical. Practical illustrations, particularly the QPE and Shor's algorithm, exemplify the potential role of QPUs within heterogeneous settings like OpenCL. These examples not only demonstrate the feasible integration but also validate its execution through circuit simulations. Actual QPU implementation, as posited, hinges on refining compilers to produce quantum–machine–readable code.

It is pertinent to note that subsequent steps in this research may not be exclusively tied to OpenCL. OpenCL was chosen for this investigation due to its comprehensive documentation and active community, making it a representative heterogeneous platform standard for the study's objectives. Notably, such standards were originally formulated without anticipating the integration of quantum computers, leading to certain challenges as exposed in Sect. 3.1. Nonetheless, there are some following steps that should be approached in order to achieve a seamless integration of QPUs into HPC environments, i.e., heterogeneous platforms:

1. Development of a driver for QPUs. QPUs, with their distinct quantum architecture, require specialized drivers to maximize their potential within these platforms. A tailored driver facilitates seamless integration of QPUs alongside classical computational units, ensuring efficient task distribution and optimization. By bridging the technical gap between classical and quantum systems, such a driver plays a pivotal role in unlocking the full potential of these combined computational environments.
2. Kernel code compilation into a format interpretable by quantum machinery, or into a quantum intermediate code. While there are established examples of such developments in the literature such as Qiskit, ProjectQ [18], Cirq [19] or QCOR, among others (being QCOR really interesting for its ability of generating various quantum machine codes depending on the technology used [20]), integration into a heterogeneous platform remains a gap. Ideally, a single codebase should serve diverse hardware, be it a QPU, GPU, or CPU, with each device tasked to translate the code into its device-specific instructions.

Moving forward, the confluence of quantum and classical computation within heterogeneous platforms heralds a new frontier in computational capabilities. While challenges persist, they are not insuperable. The advancements highlighted in this study, and the outlined future directions, pave the way for a collaborative computing era, where quantum and classical systems work in tandem to address problems of unprecedented complexity. Embracing the potential of this fusion, an important impact is anticipated on various scientific and technological domains.

Author contributions JV-P wrote the main manuscript and prepared all figures and tables. All authors reviewed the manuscript and correct errors and incorrections found in it.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by MICINN through the European Union NextGenerationEU recovery plan (PRTR-C17.I1), and by the Galician Regional Government through the "Planes Complementarios de I+D+I con las Comunidades Autónomas" in Quantum Communication. Simulations on this work were performed using the Finisterrae III Supercomputer, funded by the project CESGA-01 FINISTERRAE III. This work was also supported by the Ministry of Economy and Competitiveness, Government of Spain (Grant Numbers PID2019-104834GB-I00, PID2022-141623NB-I00 and PID2022-137061OB-C22), Consellería de Cultura, Educación e Ordenación Universitaria (accreditations ED431C 2022/16 and ED431G-2019/04), and the European Regional Development Fund (ERDF), which acknowledges the CiTIUS-Research Center in Intelligent Technologies of the University of Santiago de Compostela as a Research Center of the Galician University System.

Data availability Specific data are not required to reproduce this study. The work itself explains which input data are used and how to generate it.

Declarations

Conflict of interest The authors have no conflicts of interest that may have affected the content of this work.

Ethics approval Not applicable.

References

1. Jääskeläinen P, La Lama CS, Schnetter E, Raiskila K, Takala J, Berg H (2014) PoCL: a performance-portable OpenCL implementation. Int J Parallel Program 43(5):752–785. https://doi.org/10.1007/s10766-014-0320-y
2. Suzuki Y, Kawase Y, Masumura Y, Hiraga Y, Nakadai M, Chen J, Nakanishi KM, Mitarai K, Imai R, Tamiya S, Yamamoto T, Yan T, Kawakubo T, Nakagawa YO, Ibe Y, Zhang Y, Yamashita H, Yoshimura H, Hayashi A, Fujii K (2021) Qulacs: a fast and versatile quantum circuit simulator for research purpose. Quantum 5:559. https://doi.org/10.22331/q-2021-10-06-559
3. Nielsen MA, Chuang IL (2010) Quantum computation and quantum information, 10th anniversary edn. Cambridge University Press, Cambridge
4. Humble TS, McCaskey A, Lyakh DI, Gowrishankar M, Frisch A, Monz T (2021) Quantum computers for high-performance computing. IEEE Micro 41(5):15–23. https://doi.org/10.1109/MM.2021.3099140
5. Stone JE, Gohara D, Shi G (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng 12(3):66–73. https://doi.org/10.1109/MCSE.2010.69

6. Resch S, Karpuzcu UR (2019) Quantum computing: an overview across the system stack. arXiv preprint arXiv:1905.07240. https://doi.org/10.48550/arXiv.1905.07240
7. Chong FT, Franklin D, Martonosi M (2017) Programming languages and compiler design for realistic quantum hardware. Nature 549(7671):180–187. https://doi.org/10.1038/nature23459
8. Saki AA, Alam M, Ghosh S (2019) Study of decoherence in quantum computers: a circuit-design perspective. arXiv preprint arXiv:1904.04323. https://doi.org/10.48550/arXiv.1904.04323
9. Giovannetti V, Lloyd S, Maccone L (2008) Quantum random access memory. Phys Rev Lett. https://doi.org/10.1103/physrevlett.100.160501
10. Tilly J, Chen H, Cao S, Picozzi D, Setia K, Li Y, Grant E, Wossnig L, Rungger I, Booth GH, Tennyson J (2022) The variational quantum eigensolver: a review of methods and best practices. Phys Rep 986:1–128. https://doi.org/10.1016/j.physrep.2022.08.003
11. Ruefenacht M, Taketani BG, Lähteenmäki P, Bergholm V, Kranzlmüller D, Schulz L, Schulz M (2022) Bringing quantum acceleration to supercomputers. Technical report, Leibniz Supercomputing Centre. https://www.quantum.lrz.de/fileadmin/QIC/Downloads/IQM_HPC-QC-Integration-Whitepaper.pdf
12. KOW Group (2023) The OpenCL Specification. Version 3.0. https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf
13. Cerezo M, Arrasmith A, Babbush R, Benjamin SC, Endo S, Fujii K, McClean JR, Mitarai K, Yuan X, Cincio L, Coles PJ (2021) Variational quantum algorithms. Nat Rev Phys 3(9):625–644. https://doi.org/10.1038/s42254-021-00348-9
14. Farhi E, Goldstone J, Gutmann S (2014) A quantum approximate optimization algorithm. arXiv preprint arXiv:1411.4028. https://doi.org/10.48550/arXiv.1411.4028
15. Spliet R, Howes L, Gaster BR, Varbanescu AL (2014) KMA: a dynamic memory manager for OpenCL. In: Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU-7), pp 9–18. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2588768.2576781
16. Shor PW (1997) Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J Comput 26(5):1484–1509. https://doi.org/10.1137/s0097539795293172
17. Qiskit contributors (2023) Qiskit: an open-source framework for quantum computing. https://doi.org/10.5281/zenodo.2573505
18. Steiger DS, Häner T, Troyer M (2018) Projectq: an open source software framework for quantum computing. Quantum. https://doi.org/10.22331/q-2018-01-31-49
19. Cirq Developers (2023) Cirq. Zenodo. https://doi.org/10.5281/zenodo.8161252
20. Mintz TM, Mccaskey AJ, Dumitrescu EF, Moore SV, Powers S, Lougovski P (2020) QCOR: a language extension specification for the heterogeneous quantum–classical model of computation. ACM J Emerg Technol Comput Syst (JETC) 16(2):1–17

Authors and Affiliations

Jorge Vázquez-Pérez[1] · César Piñeiro[1] · Juan C. Pichel[1] · Tomás F. Pena[1] · Andrés Gómez[2]

✉ Jorge Vázquez-Pérez
jorgevazquez.perez@usc.es

César Piñeiro
cesaralfredo.pineiro@usc.es

Juan C. Pichel
juancarlos.pichel@usc.es

Tomás F. Pena
tf.pena@usc.es

Andrés Gómez
agomez@cesga.es

[1] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Galicia, Spain

[2] Centro de Supercomputación de Galicia (CESGA), 15705 Santiago de Compostela, Galicia, Spain