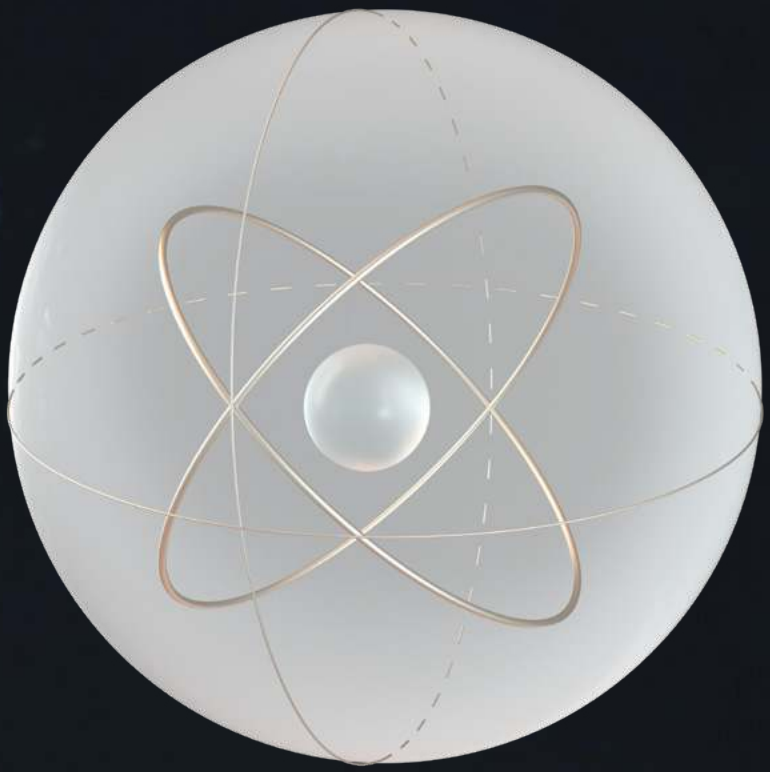
QUANTUM MACHINES

USE CASE

Quantum Orchestration for Neutral Atoms

Discover a robust way to arrange, control, and experiment with neutral atoms using the Quantum Orchestration Platform



ATOM ARRANGEMENT

Neutral atom-based quantum computers use single atoms trapped in an array of optical tweezers as qubits. The atoms are probabilistically loaded into an array of traps (P~50%) from a background cloud of atoms. The trap is so tight that trapping two atoms in a single trap results in the loss of both, ensuring that each trap site has either zero or one atom. This means that preparing a programmable initial configuration of 2D atom arrays requires shuffling atoms around after their initial loading. A tightly packed 2D array of atoms (seen in Figure 1) is a typical target configuration but other options are used [1] as they can be more suited for particular kinds of computations or simulations.

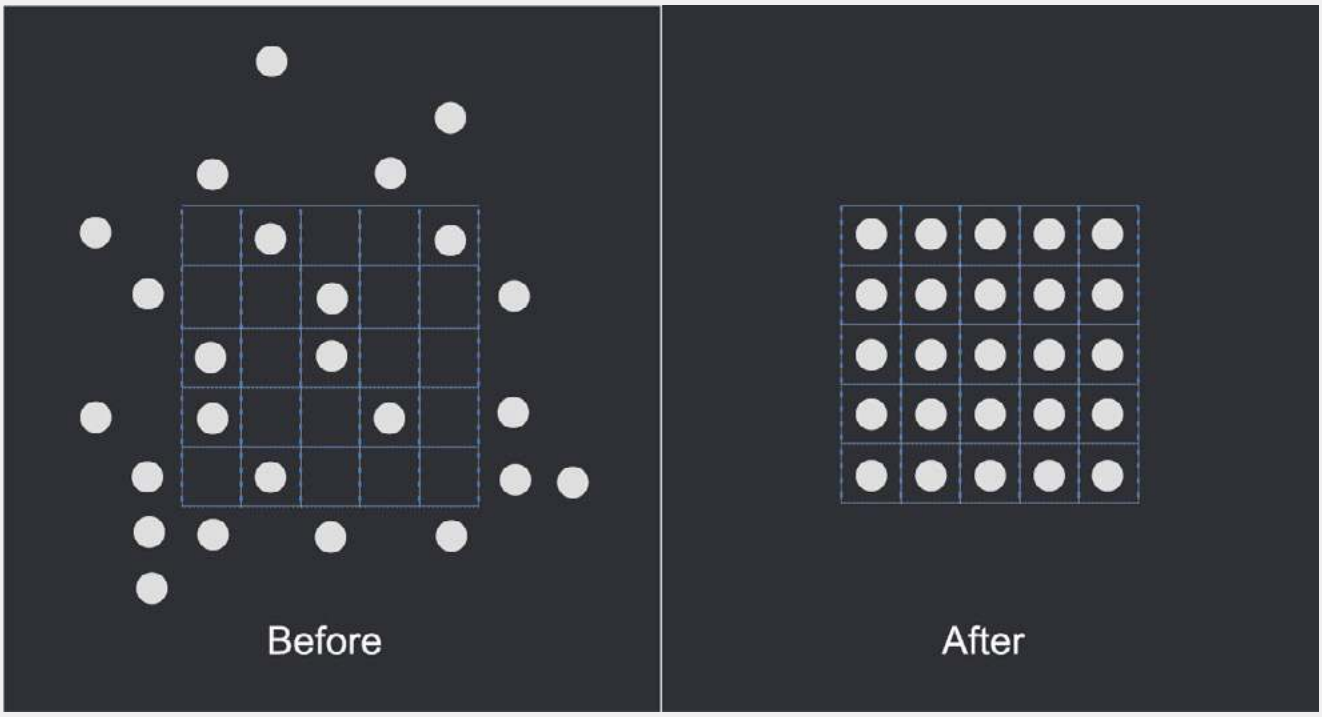


Fig. 1 *Before/after arrangement schematics of a 5×5 array of single atoms in a 2D tightly packed configuration.*

In this example, we will demonstrate the fundamental building block of 2D atom arrays, the arrangement of a single line of atoms. We will show how to use QUA, our high-level pulse programming language, to program a sequence of pulses that arranges an array of atoms into any desired configuration using 2 DAC channels of the OPX+ device driving 2 acousto-optic deflectors.

SYSTEM SETUP

In order to understand the example at hand, it is first important to understand the system at hand. First, we will discuss the operating principles of AOM (see Figure 2).

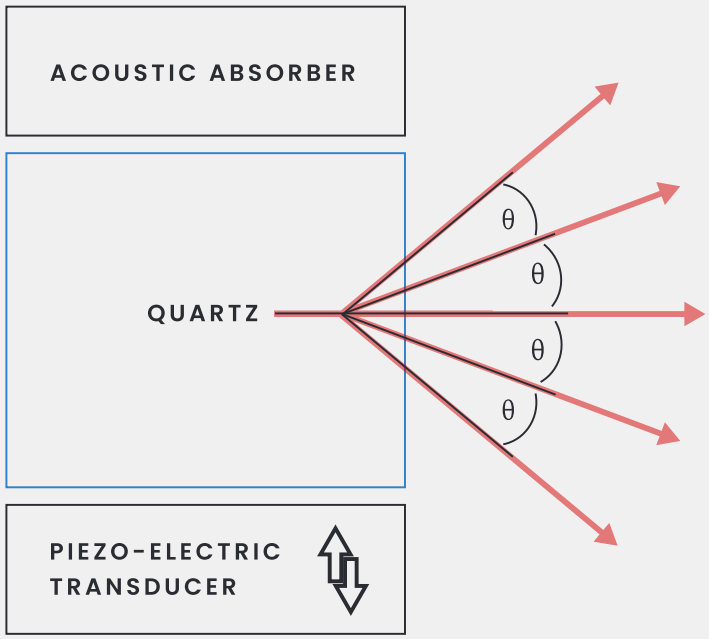


Fig. 2 *The operating principle of an AOM or AOD. An incoming laser beam is diffracted by an RF signal traveling perpendicular to the direction of propagation of the laser beam.*

In general strokes, an acousto-optical modulator or deflector (AOM/AOD) is a 3-port device that accepts a laser beam in one of its ports and an RF signal on the other and outputs a laser beam diffracted by the RF signal. The coupling between light and RF is done via a crystal with piezoelectric properties. The result is a shift of the laser frequency and angle of refraction of the beam which are both proportional to the injected RF frequency.

The dynamic trap is often generated using two crossed acousto-optical modulators, one controlling the vertical position of the beam and the other the horizontal one. By playing a single RF tone to one of the AODs we select a row (or column) on which we want to arrange atoms. The second AOD RF channel can then contain a multitone signal, grabbing several atoms at the same time and moving them to empty sites on the same row (or column). In such a way atoms are arranged row by row (or column by column). Faster arrangement protocols moving multiple lines in parallel are also possible but usually require additional AOD pairs.

We perform the ordering line by line by grabbing multiple atoms at a time and moving them to their target destination, similarly to what was done in [2]. Other strategies are possible, such as moving single atoms along 2D trajectories [3]. Here, we focus on parallelized multi-atom arrangement strategies as we believe they perform and scale better than moving atoms one by one. Trajectories are calculated in real-time based on the atoms' positions while the pulse lengths and chirp rates are dynamically adjusted to be the fastest possible without resorting to heating up the atoms.

We will discuss the basic building blocks of the system and then show a full QUA program example that assembles one line of atoms into the desired configuration. We will also give the raw DAC trace and spectrogram as sampled by a 1GSPS ADC so that you can be convinced that the desired signal was generated with no phase slips, jumps, or gaps. Determining the atoms' position is done by a separate readout system composed of a camera and a PC that performs simple ROI image processing to determine the atom occupation matrix. It is then transmitted directly to the OPX+ pulse processing unit (PPU) registers via a python API. The communication latency for a ~100 traps array is typically <5ms. Faster transfer protocols are also possible (contact us for more on this).

ATOM ARRANGEMENT OPTIMIZATION

Although this example focuses on the arrangement itself, the initial optimization of the waveform profile is also usually taken care of by QUA using calibration protocols that find corrections for AOD and RF amplifier non-linearities. It is also usually done by using off-the-shelf Python optimization libraries together with a cost-function implemented in real-time using QUA. If real-time calibration/locking is required, a fully native QUA code that includes an optimizer running entirely on the PPU can be used. It will implement calibration and locking configurations, from simple PIDs to PDH locking schemes that can reach MHz bandwidths all directly coded from a high-level programming language. If you'd like to access a simulated environment where you can play around and test all this for yourself, shoot us a line.

All the amplitude ramps and frequency chirps required to arrange atoms into a 10 sites array are programmed with ~100 lines of QUA code and compiled to a code running on our pulse processing unit (PPU). The OPX+ then generates on the fly a phase-coherent 1-5ms long pulse at 1GSPS that arranges 9 atoms into their desired target state. The power ramping (AM) and frequency chirping (FM) of the individual tweezer tones don't have to be a simple linear ramp therefore we opted for a Blackman amplitude ramp as it usually provides better results and demonstrates that arbitrary AM profiles are possible. We have left the FM chirp to be a simple linear frequency ramp, however, that can also be changed to an arbitrary frequency modulation trajectory which can be optimized to further speed up the arrangement.

As the code example below will show, QUA code is so intuitive and straightforward that it allows users to quickly build up more elaborate arrangement protocols (such as implementing atom reservoirs and real-time defect correction) which would be nearly impossible to quickly iterate on using low-level FPGA programming.

As the number of qubits increases and arrays become larger, the lifetime of a single-filled configuration will drop down to levels where real-time correction of array defects will be necessary. For a background gas limited lifetime of ~10 seconds, a 256 qubit configuration [4] has a lifetime of only about 40ms which is comparable to the timescale for the camera exposure (usually around 20 ms) and shorter than a typical rearrangement sequence (50-100ms). This results in arrays with defects (missing atoms) after the first rearrangement pass thus requiring a second (shorter) rearrangement sequence to reach near-unity filling ratios. This means that for a given background pressure and laser power what will eventually limit the size of atom arrays is the efficiency of atom rearrangement. Thus, a powerful and flexible platform on which to quickly try out and iterate various arrangement protocols will be indispensable in the quest to make ever-larger qubit systems based on atom arrays.

SEQUENCE DESCRIPTION

A typical procedure for generating ordered atom arrays starts by creating a static array of traps by using a spatial light modulation device (SLM) which holographically generates an arbitrary array of traps (see Figure 2). Other methods to generate static traps are also possible but this choice makes little difference in what follows.

Superimposed on top of this static trap array a second, dynamic, and deeper trap array is adiabatically turned on in order to capture individual atoms and shuffle them to their target destination. After the shuffling has been completed the dynamic array is adiabatically switched off and the atoms remain trapped in the static trap ready to perform quantum operations (see Figure 3).

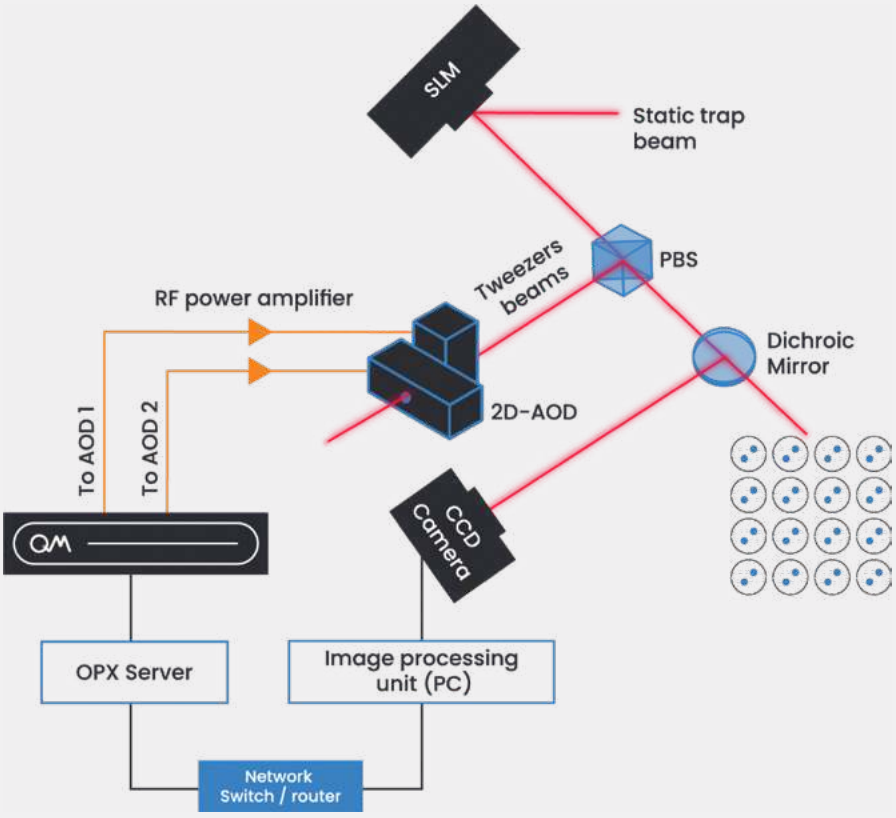


Fig. 3 *Typical Atom Array Setup. The static trap beam is reflected from an SLM (Spatial Light Modulator) which imprints the static trap array on the beam. The static array beam then goes through a polarizing beam splitter (PBS) where it is combined with the dynamical tweezers beams used to perform the atom arrangement.*

The dynamical tweezers beams are created using a pair of RF signals originating from the OPX+. Each one of those signals (one for the horizontal and one for the vertical axes) goes through a power RF amplifier before going into an acousto optic deflector (AOD) where they both diffract a single laser beam to generate the multitude of tweezers used in the arrangement sequence.

The static trap and tweezer beams then go through a dichroic mirror and beam shaping optics before entering the vacuum chamber to interact with the atoms. Light scattered from the atoms is diverted to an EMCCD camera via the dichroic mirror which takes images of the atom array, sends the images to a processor which does simple image processing

to determine the position of the atoms, and sends the atom locations via ethernet to registers on the OPX+ pulse processing unit (PPU). These registers are then accessible from QUA and the information about the position of the atoms is used to calculate the arrangement sequence in real-time.

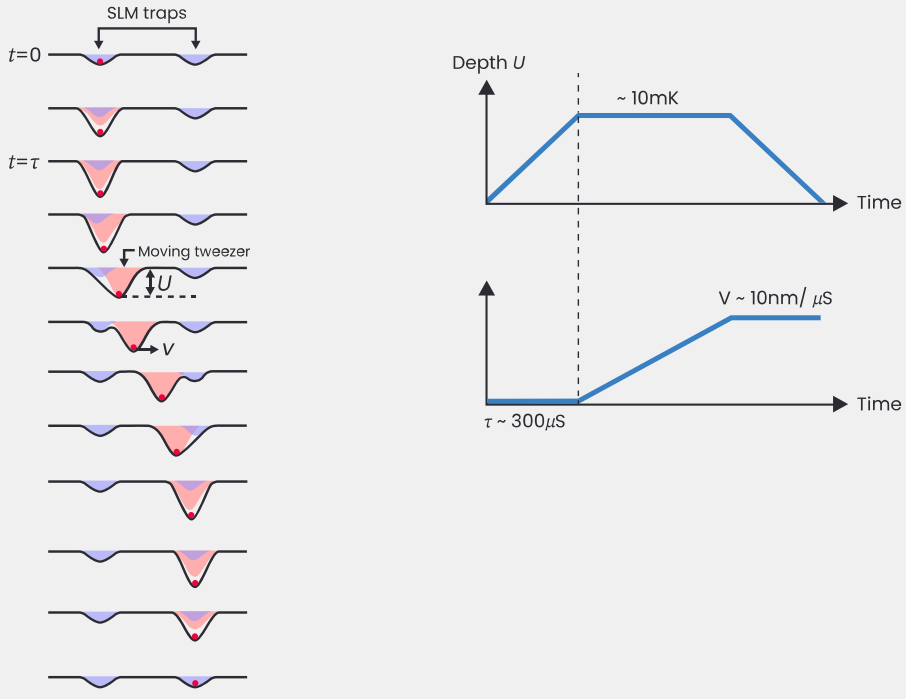


Fig. 4 Time Sequence Of A Single Tweezer Picking Up A Single Atom. *At T=0 An Atom Is Trapped In The Static Trap (The One Made Using The SLM In This Example).*

Figure 4 shows a single tweezer time sequence for a single atom. A tweezer beam is then superimposed on the static trap by setting the appropriate RF tone frequency and the tweezer power is ramped over ~300 us so as to generate a trap potential that is deeper than the one generated by the static SLM trap.

The tweezer RF tone is then linearly chirped so as to sweep the beam position, the atom is trapped in the light field generated by the tweezer and is thus moved by the tweezer beam. The chirp rate must be lower than a certain threshold so as to avoid heating the atom during transport. Typical maximum transport velocities are in the order of 10 nm/us . After the atom reaches its new desired trap position the chirping stops and the power is ramped back down (again in about 300us) so as to place the atom back into the static trapping potential generated by the SLM.

Multiple tweezer beams are superimposed thereby picking up and placing several atoms at the same time. In the simple example we show below we pick up and transport 9 atoms at a time but in general, more atoms can be moved in parallel and further optimization of the amplitude and frequency ramps can be performed to shorten the arrangement time.

At the end of the arrangement cycle, the dynamic tweezers' power is ramped down and a second image of the atoms is taken to verify the arrangement and detect holes in the array which will be corrected in a second rearrangement cycle.

PROTOCOL IMPLEMENTATION

A common sequence for neutral atom-based quantum computers using the OPX+ (see Figure 3) is thus composed of the following steps:

1. Loading of atoms from a magneto-optical trap or optical molasses into the static trap. This loading is probabilistic and so only about 50% of the static trap sites are filled with atoms.
2. Taking an image of the loaded atom ensemble in the static trap array and determining the position of atoms using a straightforward region of interest image processing procedure and transfer the atom occupation matrix to the OPX+ pulse processing unit (PPU).
3. Determining the trajectories atoms need to perform in order to reach the target configuration (this is done live on the PPU).
4. Adiabatically turning on the dynamic tweezer trap array coming from 2 acousto-optic deflectors so as to capture only the atoms that need to be shuffled to new positions on the static trap.
5. Chirping the frequency of each tweezer beam in the dynamic tweezer trap independently so that all atoms reach their target position.
6. Ramping down the arrangement tweezer 1D array.
7. Changing the horizontal selector AOD frequency and repeat steps 3-6 until all rows have been arranged.
8. Taking another image to verify the target configuration is obtained and detect defects in the array.
9. Repeating the arrangement procedure to improve initialization efficiency.
10. Performing quantum operations.
11. Readout = Taking another image of the atoms and deduce the quantum state.

The QUA code attached below performs steps 3-6 which compose the heart of the arrangement protocol. The idea is to calculate in real-time the required tweezer detuning for each atom, find the maximum required detuning, and divide it by the maximal chirp rate (determined experimentally as the fastest chirp rate that does not heat the atoms) to get the chirp pulse length.

Each atom's tweezer chirp rate is then calculated so that the atom moves to the desired destination site during the calculated chirp pulse length. So all atoms move together with the same pulse length, but each tweezer/atom has a different chirp rate resulting in different trajectories for the various atoms. The whole calculation period, prior to sending the pulses to the AODs takes approximately 11μs in this example. This is a negligible amount of time compared to other time scales in the process, thus leaving plenty of time to perform even more complicated calculations.

```
from qm.qua import *
from qm import SimulationConfig
import numpy as np
from time import sleep
from qm.QuantumMachinesManager import QuantumMachinesManager
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
from Array_sorting_config import * #this config file contains all the pulse
parameters, the definition of the Blackman pulses as well as the initial
frequency/phase values for the individual array sites

qmm = QuantumMachinesManager()
qm = qmm.open_qm(config)


Row_size = 10
Max_Number_of_Tweezers = 9
Row_frequencies_list = [Row_selector_IF+Row_Spacing*x for x in range(Row_size)]


# The size of the atom location vector is usually larger than the target one
since you need to collect atoms from an array that is larger as the occupation
has a probability of ~50%
# We thus usually take the initial array size to be > 2x the number of atoms
required in the target state so as to be sure we have enough atoms to completely
fill in the target array
Atom_location_list = [1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0,
0, 0, 1, 0, 1, 0, 0, 1, 0, 1]
Atom_Target_List=[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
# The next commented line is an alternative atom target list containing atoms in
every other site, we will test both target configurations on the same code
#Atom_Target_List=[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0]

Atom_Target_List_bool = [bool(x) for x in Atom_Target_List]
Target_frequencies_python = [int(Column_IF[i]) for i in range(len(Atom_Target_
List)) if Atom_Target_List[i]]

with program() as order_atoms:


    # Declaring some variables we will use later
    Current_Row = declare(int, value=0) # the row we are currently arranging,
        when arranging a 2D array this variable will be scanned in a for loop but
        is kept constant at 0 for this single row example
    i = declare(int)
    j = declare(int)
    N_atoms_in_row = declare(int)
    N_atoms_in_row_target = declare(int)
    Row_Frequency = declare(int)
    Max_Detuning = declare(int)
    Min_Detuning = declare(int)
    Tweezer_Moves_Length = declare(int)
    OneOver_Tweezer_Moves_Length = declare(int)
    Number_of_Tweezers = declare(int)


    # Declaring some vectors we will use later
```

```
        Atom_location = declare(int, value=Atom_location_list) # a QUA vector
        containing the location of the atoms, the python vector 'Arom_location_list'
        is received from the image acquisition system
        Atom_Target_List_QUA = declare(int, value=Atom_Target_List)# a QUA vector
        containing the target destination of the atoms
        Row_Frequencies = declare(int, value=[int(x) for x in Row_frequencies_list])
        # a QUA vector containing the frequencies corresponding to each row
        Phase_list = declare(fixed, value = Phases_List) # a QUA vector containing
        the phases of each site on the row
        Tweezers_phases = declare(fixed, value = Phases_List) # a QUA vector
        containing the phases of each tweezer, initialized to the original phase list
        but will be changed during runtime
        Column_frequencies = declare(int, value=Column_IF) # a QUA vector containing
        the frequencies of each column

        Tweezers_frequencies = declare(int, value=[int(x) for x in np.zeros(Max_
        Number_of_Tweezers)]) # a QUA vector containing the frequency/location of each
        tweezer, is initialized to be outside the AOM band and updated during runtime
        Amplitude_list = declare(fixed, value=[0.0] * Max_Number_of_Tweezers) # a QUA
        vector containing the amplitudes of each tone, will be either 0 or 1 but can
        be updated to reflect AOM/RF amplifier non linearities
        Tweezers_Detunings = declare(int, value = [int(x) for x in np.zeros(Max_
        Number_of_Tweezers)]) # a QUA vector containing the detuning of each atom to
        his target, will be updated during runtime
        Chirp_Rates = declare(int, value = [int(x) for x in np.zeros(Max_Number_of_
        Tweezers)]) # a QUA vector containing the chirp rate for each tweezer, will
        be updated during runtime
        Target_frequencies = declare(int, value = Target_frequencies_python) # a QUA
        vector containing the target frequencies/locations of the atoms
        Max_Number_of_Tweezers_QUA = declare(int, value = Max_Number_of_Tweezers) #
        the maximum allowed number of tweezers (the maximum number of atoms to move)


        assign(N_atoms_in_row, Math.sum(Atom_location)) # assign to N_atoms_in_row
        the total number of atoms to arrange
        assign(N_atoms_in_row_target, Math.sum(Atom_Target_List_QUA)) # The total
        number of atoms needed in the ordered row
        assign(Row_Frequency, Row_Frequencies[Current_Row]) # The current row index


        # generating an integer vector of size=3 and storing three values in it (we
        could have done this with a for loop but since it's only 3 values we did it
        by hand)
        Atoms_or_Targets = declare(int, size=3)
        assign(Atoms_or_Targets[0], N_atoms_in_row) # how many atoms in the current
        row?
        assign(Atoms_or_Targets[1], Max_Number_of_Tweezers) # what's the max number
        of tweezers we are allowed to use?
        assign(Atoms_or_Targets[2], N_atoms_in_row_target) # how many atoms are in
        the target state?


        # how many tweezers do we need?
        # The number of required tweezers for this row arrangement is the minimum of
        the number of atoms in the initial configuration, the number of final atoms
        needed in the ordered row and the number of tweezers available
        assign(Number_of_Tweezers, Math.min(Atoms_or_Targets)) # find the minimum
```

```
117       valur in Atoms_or_Targets and store the answer in Number_of_Tweezers
118
119       # scan the atom locations and update the list of Tweezers frequencies
120       assign(j, 0) # assigning 0 to the j index
121       assign(i, 0) # assigning 0 to the i index
122       with while_(j < Number_of_Tweezers): # pick up only the amount of atoms you
123       need to fill the target, if there aren't enough atoms or tweezers available
124       then pick as many as you can
125
126         with if_(Atom_location[i] == 1): # if an atom is present in site i
127
128           assign(Amplitude_list[j], 1.0) # set the amplitude of the Tweezer j to
129           active (=1), this can also be any other value and can be optimized to
130           compensate for AOM and RF amplifier nonlinearities. We kept things simple
131           here.
132           assign(Tweezers_frequencies[j], Column_frequencies[i]) # update the
133           tweezer j frequency with that correponding to the atom index i
134           assign(Tweezers_phases[j], Phase_list[i]) # update the tweezer phase j
135           with that corresponding to the atom index i
136           assign(j, j+1)
137
138         assign(i, i+1)
139
140       # calculate the detunings vector, In this example, tweezer number 0 will go
141       to target site 0, tweezer 1 to target site 1 etc.
142       j2=declare(int)
143       with for_(j2, 0 , j2 < Number_of_Tweezers, j2+1): assign(Tweezers_
144       Detunings[j2], Target_frequencies[j2] - Tweezers_frequencies[j2]) # The
145       detuning each tweezer needs to undergo (chirp) is the difference between
146       the initial and final frequenciesassign(Min_Detuning, Math.min(Tweezers_
147       Detunings)) # find the minimum detuning assign(Max_Detuning, Math.
148       max(Tweezers_Detunings)) # find the maximum detuning # Take the absolute
149       value of the detuning and store it into Max_Detuning with if_(-1*Min_Detuning
150       > Max_Detuning):
151
152         assign(Max_Detuning, -Min_Detuning)
153
154
155       # Calculate the pulse length:
156       # We calculate the tweezer arrangement pulse length as being the maximum
157       detuning that an atom needs to traverse divided by the maximum allowed chirp
158       rate
159       # In our case Max_Rate is a python constant (=3.2 Hz/ns) which was
160       experimentally found to be sufficiently slow so as not to heat up atoms.
161       # The resulting pulse length comes out to be Tweezer_Moves_Length=3ms for the
162       particular initial and final atom configurations in this example
163       assign(Tweezer_Moves_Length, Cast.to_int(Max_Detuning/(1000*Max_Rate))*1000)
164       # We multiply and divide by 1000 to avoid integer casting issues since Max_
165       Rate can in general be non-integer but is usually specified with up to a
166       single decimal point as this provides sufficient precision for the use
167
168       # Calculate the chirp rates for all Tweezers.
169       # Each tweezer has its own chirp rate depending on the initial and final
170       position of the arranged atom. All tweezers move at the same time with a
171       common chirp pulse length (i.e. they all leave and arrive at the same time).
172       with for_(j, 0, j < Number_of_Tweezers, j+1):#N_active
173
174         assign(Chirp_Rates[j], Cast.to_int((Tweezers_Detunings[j])/(Tweezer_Moves_
```

```
        Length/1000))) # x1000 to go to 'mHz/ns' scaling instead of the default
        'Hz/ns' scaling so as to have a better resolution on the chirp values.
        'uHz/ns' and 'nHz/ns' are also possible


    # set the phases and frequencies of the Tweezers
    update_frequency('Row_selector', Row_Frequency)
    for index in range(Max_Number_of_Tweezers):

        frame_rotation(Tweezers_phases[index], 'Column_{}'.format(index + 1))
        update_frequency('Column_{}'.format(index + 1), Tweezers_
        frequencies[index])


    # Now that the calculations are done, we start playing the pulses.
    # All calculations thus far took 11 µs to do in real time on the PPU
    # Transfer of the atom location matrix takes less than 5ms for a 100 atom
    array

    # align all Tweezers/columns and row selector
    align('Row_selector', 'Column_1', 'Column_2', 'Column_3', 'Column_4',
    'Column_5', 'Column_6', 'Column_7', 'Column_8', 'Column_9')

    # ramp up power of occupied Tweezers and row selector, we use a Blackman
    pulse to ramp up and down the tweezer's amplitudes

        play('Blackman_up_long', 'Row_selector')

    for element_index in range(Max_Number_of_Tweezers):
    play('Blackman_up_long'*amp(Amplitude_list[element_index]), 'Column_{}'.
    format(element_index+1))

    # chirp Tweezers, each with its own chirp rate but all share the same pulse
    length of Tweezer_Moves_Length/4. The division by 4 is there since we need to
    supply the pulse duration in steps of 4ns since
    # this is the units of the PPU period which is 4ns long (250MHz clock).
    for element_index in range(Max_Number_of_Tweezers):
        play('Constant' * amp(Amplitude_list[element_index]), 'Column_{}'.
        format(element_index + 1), duration=Tweezer_Moves_Length/4, chirp=(Chirp_
        Rates[element_index], 'mHz/nsec'))


    # ramp down power of occupied Tweezers and row selector, again with a Blackman
    pulse
    #play('Blackman_down_long', 'Row_selector')
    for element_index in range(Max_Number_of_Tweezers):
        play('Blackman_down_long'*amp(Amplitude_list[element_index]), 'Column_{}'.
        format(element_index+1))

# Executing the program
job = qm.execute(order_atoms)
job.result_handles.wait_for_all_values()
res = job.result_handles
```

QUA Code.

ATOM ARRANGEMENT RESULTS

When running this code and acquiring the DAC output going to the column AOD (the row AOD signal is trivial, it's simply a sinusoidal ramping up and down) with a 1GSPS acquisition card we get the following waveform (see Figure 6):

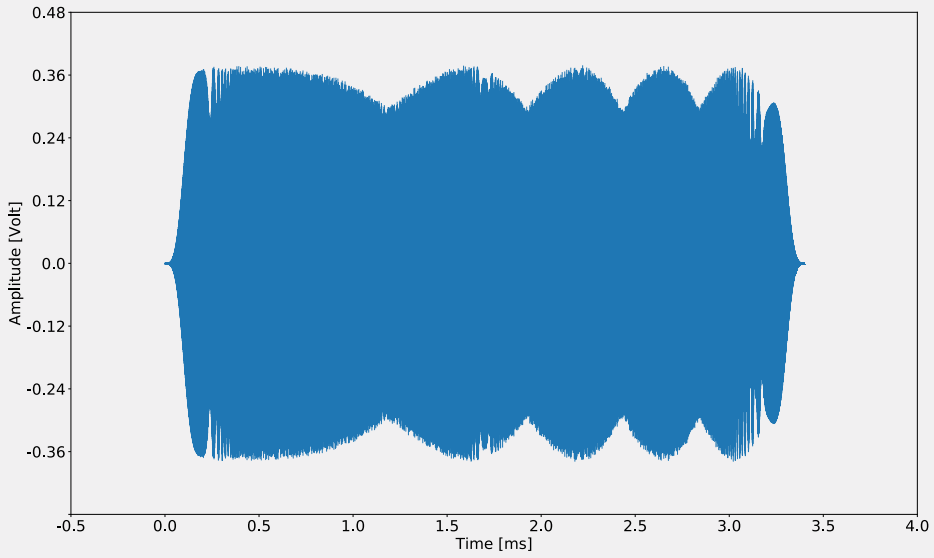


Fig. 6. *Acquired waveform of the atom_arrange program (column DAC channel).*

Here we can see the 200µs-long Blackman ramp increasing the power adiabatically for all the tones, the 3ms long chirp pulse, and the 200µs Blackman ramp decreasing the power adiabatically to release the atoms into the static trap. The total pulse length is thus 3.4ms long.
Zooming in on the pulse we can see that it is phase-coherent and contains no discontinuities, as seen in Figure 7.

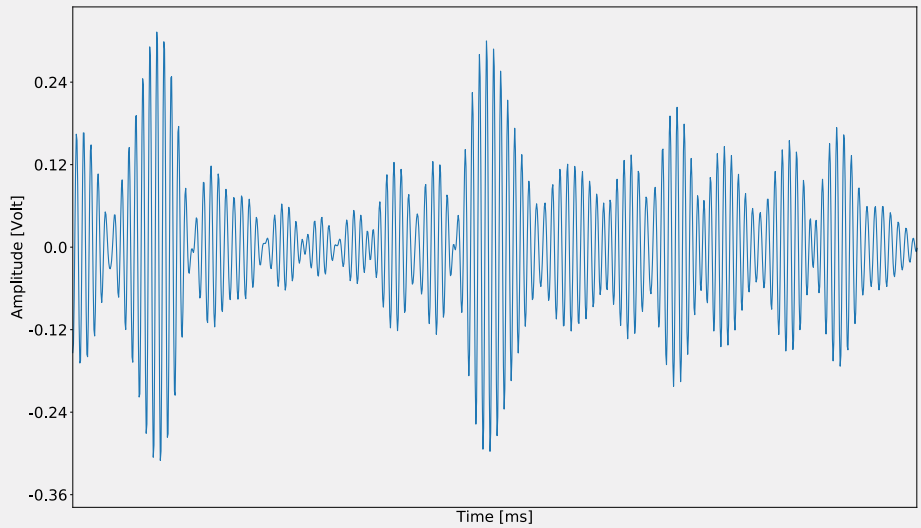


Fig. 7. *Zoom-in of the 3.4ms long arrangement pulse*

By clicking here you can download the raw 1GSPS, 3.4ms long pulse as acquired by a 1GSPS acquisition card (CSV format) and you could then verify for yourself that the pulse was created with no phase jumps or dead time in its entirety. The spectrogram of this pulse is in Figure 8.

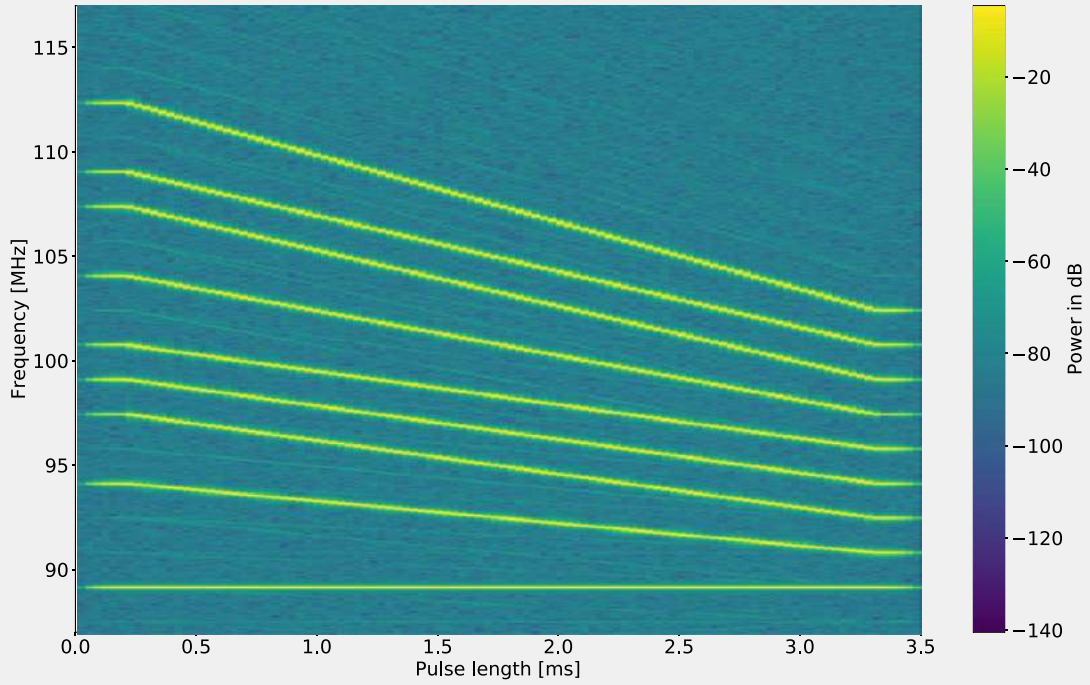


Fig. 8. *Spectrogram of the pulse seen in figure 4*

In order to interpret the results of the spectrogram, we need to understand that the left-hand side represents the initial states of the atoms, while the right-hand side represents their final states. Moving vertically from the bottom to the top, we read the atoms' initial states:

Atom_location_list = [1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1]

Which is then moved to their final states

Atom_Target_List=[1, 1, 1, 1, 1, 1, 1, 1]

If, for example, we would want to obtain a different final state, we can set:
Atom_Target_List=[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0]

A setting where we'd want the atoms to form a lattice with a 50% filling ratio, the resulting pulse will be as seen in Figure 9.

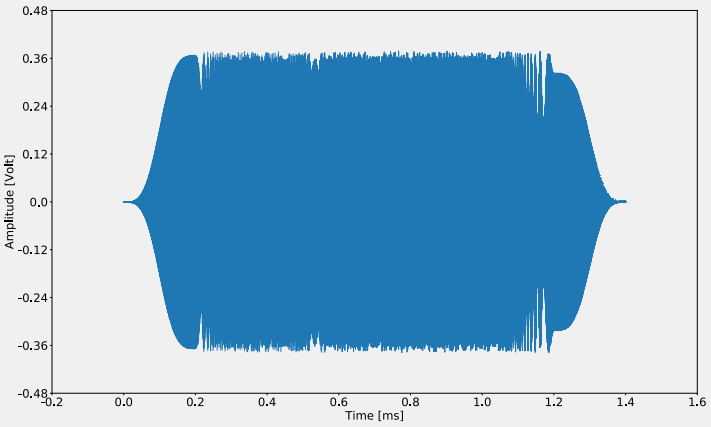


Fig. 9. *Acquired waveform of the 1.4ms long arrangement pulse (column DAC).*

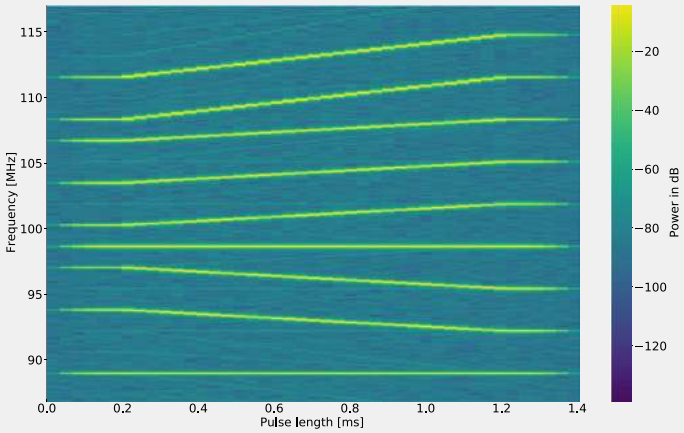


Fig. 10. *Spectrogram of the pulse seen in figure 9.*

In the spectrogram (Figure 10), we see that the real-time QUA code generated a shorter, 1.4ms-long pulse for the arrangement of this row of atoms since all atoms were at most 2 sites away from their target location. Here we see that using QUA and the OPX+ we managed to generate an adaptive atom arrangement protocol that optimizes the pulse length to each particular initial/final atom configuration.

This adaptability can be further exploited when performing measurements comparing the results of running computations on different atom array configurations. In such cases, the effective distance of the initially detected array to each of the possible target states can be evaluated in real-time in the PPU and the closest target configuration can be then chosen on the fly, thus minimizing the preparation time of ensembles of target configurations.

As previously mentioned, more elaborate arrangement protocols can be devised taking advantage of the simplicity and low latency of the QUA language and PPU.
Incorporating atom reservoirs in the vicinity of the target array to quickly fill in defects, cleaning pulses that remove unwanted atoms and anything else you could think of can now be easily programmed using a high-level programming language (QUA) and executed with the low latency and speed expected from high-end FPGA systems.

This example demonstrates that anyone with an OPX+ can now build atom arrangement protocols for hundreds of qubits in a matter of weeks instead of the laborious process of developing a custom FPGA or PC+DDS solution. For more information about how the Quantum Orchestration Platform can be used in your lab to speed your experiments, please don't hesitate to contact us, we'd love to help you out.

References

[1] R. Samajdar, et al., Quantum phases of Rydberg atoms on a kagome lattice. Proceedings of the National Academy of Sciences, 118(4) (2021)

[2] M. Endres, et al., Atom-by-atom assembly of defect-free one-dimensional cold atom arrays. Science 354(6315) (2016)

[3] D. Barredo, et al., An atom-by-atom assembler of defect-free arbitrary two-dimensional atomic arrays. Science, 354(6315) (2016)

[4] S. Ebadi, et al. Quantum Phases of Matter on a 256-Atom Programmable Quantum Simulator. arXiv preprint arXiv:2012.12281 (2020)

The Quantum Orchestration Platform

AN END TO END QUANTUM CONTROL SOLUTION TO DRIVE THE FASTEST TIME TO RESULTS, AT ANY SCALE

OPX+

RUN STATE OF THE ART EXPERIMENTS WITH EASE

An architecture designed from the ground up for quantum control, the OPX+ lets you run the quantum experiments of your dreams right from the installation. With a quantum feature-rich environment, the OPX+ is built for scale and performance. Now, you can **run the most complex quantum algorithms and experiments in a fraction of the development time**.

PULSE PROCESSING UNIT

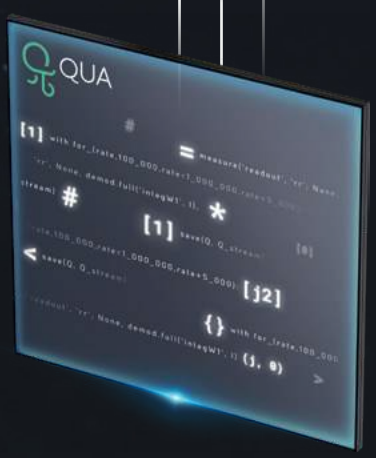
ACHIEVE THE FASTEST TIME TO RESULTS

Within the OPX+ is the Pulse Processing Unit, QM's leading-edge quantum control technology. Progress with incomparable speed and extreme flexibility. Run even the most demanding experiments efficiently, with the fastest runtimes and the lowest latencies in the industry, including quantum protocols that require **real-time waveform generation, real-time waveform acquisition, real-time comprehensive processing, and control flow.**

QUA

CODE QUANTUM PROGRAMS SEAMLESSLY

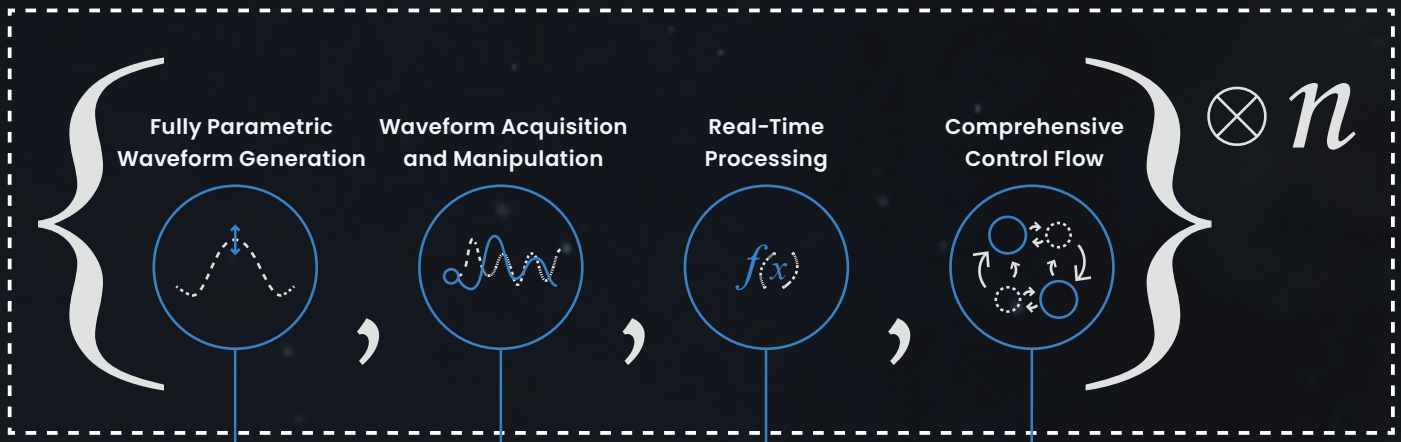
Implement the protocols of your wildest dreams as easily as writing pseudocode. Designed for quantum control, QUA is the first universal quantum pulse-level programming language. Code even the most advanced programs and run them with the best possible performance. Natively describe your most challenging experiments, from complex AI-based multi-qubit calibrations to multi-qubit quantum error correction.



**All of the information above is also valid for the OPX*

QUANTUM MACHINES

YOUR PROTOCOLS LIVE IN THIS PHASE SPACE



Real-Time Waveform Generation

- Fully parametrized: length, frequency, phase (relative & absolute), amplitude, bandwidth, chirp
- Compensations: Crosstalk matrix, FIR and IIR filters
- And many more

Real-Time Measurements

- High fidelity analog to digital conversion
- General integration & demodulation
- Weighted Integrations accumulated integrations, sliced integrations, etc.
- Time Tagging, TTL counting

Real-Time Processing

Turing complete:
- Basic arithmetics
- Evaluation of trigonometric functions
- Vector operations
- Casting of variable types
- And more (Turing complete)

Real-Time Control Flow

- If/else
- For loops
- While loops
- Switch case

Real-Time 'Quantum' Estimations

- State estimations
- Error estimations
- Bayesian estimations
- Correlation functions
- Neural nets based estimations

Real-Time Multi-Qubit Feedback

- Qubit stabilization & tracking
- Quantum error correction
- And many more

THE QUANTUM ORCHESTRATION PLATFORM COVERS THIS SPACE!

- Easily express quantum algorithms and experimental protocols that comprise all of the above.
- Seamlessly sync measurements, real-time calculations, and pulses of different quantum elements.
- Loop over a wide range of parameters in real-time, including intermediate frequencies, amplitudes, phases, delays, integration parameters, measurement axes, etc.
- Use if/else and switch-case statements to condition operations in real time (real time feedback).
- Define procedures (macros) to be reused in the code and access an extensive family of libraries.

QUANTUM MACHINES

If you wish to learn more:

info@quantum-machines.co

About Quantum Machines

Quantum Machines (QM) drives quantum breakthroughs that accelerate the path towards the new age of quantum computing. The company's Quantum Orchestration Platform (QOP) fundamentally redefines the control and operations architecture of quantum processors.

The full-stack hardware and software platform is capable of running even the most complex algorithms right out of the box, including quantum error correction, multi-qubit calibration, and more. Helping achieve the full potential of any quantum processor, the QOP allows for unprecedented advancement and speed-up of quantum technologies as well as the ability to scale into the thousands of qubits. Visit us at: www.quantum-machines.co

*The information contained in this document is confidential and intended solely for its addressees. The information is the property of QM Technologies Inc. ("QM") and may contain legally privileged information. QM may make changes to specifications and product descriptions at any time, and this document does not represent a commitment on the part of QM, but is supplied solely for allowing the intended recipients hereof to consider a general business engagement with QM. This information is subject to change without notice. © QM Technologies Inc.