We introduce QuESTlink, pronounced ‘quest link’, an open-source Mathematica package which efficiently emulates quantum computers. By integrating with the Quantum Exact Simulation Toolkit (QuEST), QuESTlink offers a high-level, expressive and usable interface to a high-performance, hardware-accelerated emulator. Requiring no installation, QuESTlink streamlines the powerful analysis capabilities of Mathematica into the study of quantum systems, even utilising remote multi-core and GPU hardware. We demonstrate the use of QuESTlink to concisely and efficiently simulate several quantum algorithms, and present some comparative benchmarking against core QuEST.

CONTENTS

I Introduction 1
A. QuESTlink facilities 2

II Technical summary 2
A. Architecture 2
B. Deployment 2
C. Remote computation 3

III User guide 3
A. Mathematica review 3
B. QuESTlink overview 3

IV Demonstrations 5
A. Decoherence 5
B. Variational imaginary-time 6
C. Noisy Trotterisation 7
D. Further examples 9

V Benchmarking 9

VI Future work 10

VII Conclusion 10

VIII Acknowledgements 10

References 10

I. INTRODUCTION

Classical emulation is crucial in the design of quantum computers and algorithms. Despite the recent demonstration of quantum supremacy [1], today’s quantum computers are of insufficient quality to run and test many interesting algorithms. Even precise quantum computers of tomorrow may provide limited help in writing new algorithms, since unlike emulators, they offer limited information about the evolving quantum state. Furthermore, some algorithms, particularly those for noisy intermediate-scale quantum (NISQ) devices [2] like the variational class of algorithms [3], admit limited analytic treatment. Hence, the value of classical emulation is undeniable. The research community needs high-level usable tools that are easy to deploy, offer rapid numerical study, and integrate with other established software. However, the exponentially-growing cost of classically simulating a quantum device makes emulation of even NISQ computers very resource intensive. Emulators must therefore make good use of classical high-performance computing techniques, like multithreading and GPU parallelisation, and be written in low-level performant languages like C. This requirement is at odds with the need for usable tools, which can be used by non-expert programmers and the wider quantum community.

Within this context we have developed QuESTlink: a high-performance Mathematica package for numerically emulating quantum computers, by off-loading expensive computation to remote accelerated hardware, running QuEST [4]. Mathematica is both a language and computational tool, prevalent among physicists, which offers a convenient interactive interface (through notebooks), and an extremely comprehensive and powerful set of utilities. Although the most widely used tool for calculations in the physical sciences [5, 6], Mathematica does not have an intrinsic toolset specifically dedicated to quantum computing emulation. In contrast, QuESTlink offers a usable Mathematica interface, without compromising the excellent performance and simulation capacity of QuEST.

From a laptop environment, a user can symbolically specify a circuit in an intuitive high-level operator format akin to how they appear in the literature. Behind a platform-agnostic Mathematica interface, QuESTlink sends the circuit to a C backend, either locally or on remote high-performance hardware, where it is efficiently emulated. QuESTlink offers multithreaded and GPU simulation of state vectors and density matrices, multi-qubit multi-controlled general unitaries, general noise processes, circuit drawing, and a wide range of stan-
standard gates. Like QuEST, QuESTlink is free, open-source and can be used stand-alone; furthermore, without setup, compilation or installation of any kind.

A. QuESTlink facilities

While the forthcoming sections give a thorough overview of QuESTlink’s facilities, here we provide a short summary to acquaint the reader with the essentials. QuESTlink is, offers, features or enables:

- Multithreaded and GPU-accelerated emulation of quantum computers.
- State-vector and density matrix simulation.
- Off-loading of simulation to remote hardware, through a backend-agnostic interface.
- Seamless integration with Mathematica’s powerful and comprehensive toolset, and interactive notebook programming style.
- Rapid development with Mathematica’s concise, functional language.
- A concise but expressive circuit language, akin to the symbolic description of circuits in the literature.
- Stand-alone with no required external downloading, compiling, or installation of any kind whatsoever.
- Through Mathematica, is compatible with all major operating systems.
- Free and open-source, hosted on Github [7].
- Rendering of circuit diagrams, which can be exported to any file format through Mathematica.
- A suite of functions for higher-level calculations, like computing Hamiltonian expectation values, and derivatives of unitary circuits.
- A comprehensive suite of unitary gates and decoherence channels, including multi-controlled multi-qubit general unitaries, and multi-qubit Kraus maps.

II. TECHNICAL SUMMARY

This section provides an overview of the inner workings of QuESTlink, and the technologies used to build it. Readers intending to use QuESTlink right away may wish to skip to Section III B.

A. Architecture

QuESTlink consists of both a Mathematica package (QuESTlink.m) and an underlying C program (quest_link.c), which interface using C/Link; an implementation of the Wolfram Symbolic Transfer Protocol (WSTP) [8] (formerly called MathLink [9]). QuESTlink emulates quantum computers using the Quantum Exact Simulation Toolkit (QuEST) [4], which is an open-source, high-performance emulator written in C. QuEST is multi-threaded, GPU-accelerated and distributed, and the first two of these facilities are made available to QuESTlink. C/Link facilitates conversion of Mathematica types, utilised in the user’s notebook, into C types, and maps Mathematica-callable functions to C functions. The executable quest_link (quest_link.c compiled with WSTP) sits below as a “backend” process, performing intermediate processing and invoking QuEST’s API. This software stack is visualised in Figure 1.

Through this stack, QuESTlink offers a high-level Mathematica interface to quantum emulation which is numerically performed in C (a significantly faster language than Mathematica), using memory persistent in the C process, and potentially using accelerated hardware. The expensive numerical representations of the quantum states reside only in the C backend (and through QuEST, may also be persistent also in GPU memory), and each are identified by a unique ID. Quantum circuits are represented symbolically in their entirety in Mathematica, and sent to the C backend (at a small runtime overhead) only at emulation-time, compactly encoded into arrays of real numbers. The circuit language, and details of the QuESTlink interface, are presented in Section III B.

B. Deployment

QuESTlink can be obtained and deployed entirely within a Mathematica notebook, without any installation or configuration. This is done by online hosting (currently at qtechtheory.org) of the Mathematica package.
code, and dependent `quest_link` executable. Figure 2 presents the process of serving the package to a user’s Mathematica kernel.

C. Remote computation

QuESTLink even enables quantum emulation using remote computing resources. Through WSTP, the `quest_link` process can run on a remote machine (e.g. a supercomputer) and connect with the user’s local Mathematica kernel via TCP/IP. The remote machine can employ more powerful hardware than available on the user’s machine, and potentially simulate larger quantum states than can fit in the user’s local memory. The protocol of off-loading a circuit simulation to remote hardware is outlined in Figure 3. The tools and documentation for setting up a remote QuESTLink server are provided between the Github repo [7], and `questlink.qtechtheory.org`.

In this remote configuration, Mathematica calls to QuESTLink functions will involve network communication with the remote environment, and hence incur overheads; this is worthwhile if the remote hardware sufficiently accelerates a large quantum simulation which otherwise dominates runtime.

Despite QuESTLink’s effort to minimise the communication cost, this network overhead will be prohibitively costly for some applications. For example, when large simulated quantum states undergo processing by the Mathematica kernel, and hence need to be copied back and forth between the local kernel and the remote environment. To mitigate this slowdown whilst still running QuESTLink remotely, one can launch their Mathematica kernel on the remote machine, using Mathematica’s remote kernel facilities [10]. QuESTLink can also be used inside other Mathematica packages, and so be launched remotely and non-interactively, without a local notebook.

III. USER GUIDE

A. Mathematica review

Before continuing, we offer a quick review of the Mathematica syntax used in this manuscript.

Evaluation of function `f` with input `a` is denoted by `f[a]`, or equivalently `f @ a`. The expression `g[f[a]]` can be formed using prefix notation as `g[f] @ a`, or postfix notation as `a // f // g`. Matrix multiplication between (possibly complex) matrices `a` and `b` is denoted by `a . b`, and `a†` denotes the conjugate transpose of `a`. Expressions with a trailing ; suppress their otherwise displayed result. While `f[] = a` denotes immediate evaluation of `a` and assignment to `f[]`, the syntax `f[] := a` denotes delayed assignment, whereby later invoking `f[]` will evaluate `a` (which may have changed) each time. Elements of a list `x = {a, b, c}` are accessed as `x[[i]]` where index `i ≥ 1`, and `x[[n;m;]]` returns the sublist spanning indices `m` to `n`. The shortcut `ex /. a -> b` replaces sub-expressions of `ex` which match pattern `a`, with `b`.

Finally, for those copying code into Mathematica, subscripts can be quickly entered into a notebook with keyboard shortcut `Ctrl` & `\[Subscript]`.

In the code snippets featured in this manuscript, context should make clear what is `input` to the Mathematica notebook, and what is a rendered `output`.

B. QuESTLink overview

The QuESTLink package requires no installation, and can be downloaded directly from within Mathematica:
The user then has a choice of several “QuEST environments” in which to perform quantum emulation, all of which provide an identical user experience.

The first, CreateDownloadedQuESTEnv[os], enables simulation on the user’s machine by directly downloading a serial QuESTlink executable from qtechtheory.org, compatible with the Operating System os (e.g. "MacOs"). This requires no apriori setup whatsoever.

CreateLocalQuESTEnv[fn] will attempt to launch an existing local QuESTlink executable, located at fn. This allows local simulation using serial, multi-core or GPU resources, depending on how the executable was compiled. Users can compile QuESTlink for their platform using the tools on the Github repo [7].

CreateRemoteQuESTEnv[ip, port1, port2] connects to a remote QuESTlink environment at the given ip address and ports. The remote machine can use serial, multithreading or GPU-acceleration to emulate quantum systems. The facilities to setup a remote QuESTlink server are provided in the Github repo [7].

Once connected to a QuEST environment, a full list of the supported QuESTlink facilities and gate symbols can be obtained by evaluating

```
? QuEST
```

and the documentation of a particular function or operator obtained similarly using `?`.

```
M is a destructive measurement gate which measures the indicated qubits in the Z basis.
```

```
CalchilbertSchmidtDistance

CalchilbertSchmidtDistance[qureg1, qureg2] returns the Hilbert–Schmidt distance (Frobenius norm of the difference) between the given density matrices.
```

Emulation begins by creating quantum registers (a “Qureg”), each represented by a state-vector or density-matrix.

```
numQb = 6;
ψ = CreateQureg[numQb];
ρ = CreateDensityQureg[numQb];
```

These functions return a unique ID for each Qureg, the memory for which is stored in the QuEST environment. Once created, QuESTlink provides a few functions to initialise Quregs:

```
InitZeroState[ψ] (* ψ → 0 ⊗ *)
InitPlusState[ψ] (* ψ → + ⊗ *)
InitClassicalState[ψ, 3] (* ψ → 0 ⊗ 11 *)
InitPureState[ρ, ψ] (* ρ → ψ ⊗ ψ *)
```

```
x = Table[RandomComplex[], 2numQb , 2numQb ];
m = x . x \[\text{Tr}\] [x . x ];
SetQuregMatrix[ρ, m];
```

A quantum circuit u can be applied to a Qureg, agnostic of whether it is a state vector (effecting u |ψ⟩) or a density matrix (effecting u ρ u†), using ApplyCircuit[u, qureg]. QuESTlink features a concise and expressive language for specifying gates and decoherence processes, where target qubits are denoted with subscript integers to gate symbols, and control gates “wrap” the base gate. The below example makes use of the Circuit function, which disables commutation and allows a concise product representation of the circuit. In combination, this enables a syntax akin to how circuits are denoted in the quantum computing literature.

```
m1[\[\theta\]] := \[
\begin{array}{cc}
0 & 1 \\
\exp[\i \theta] & 0 \\
\end{array}
\];
m2[] := RandomVariate @
CircularUnitaryMatrixDistribution[4];
u[\[\theta\]] := Circuit[
X_0 Y_1 Z_2 T_3 S_4 R_x[\[\theta\]] R_y[\[\theta\]] R_z[\[\theta\]] R_3,4[\[\theta\]] ×
C_0 \{ X_1 \} C_4 \{ R_y[\[\theta\]] \} M_8 \{ R[\[\theta\]], X_0 Y_0 X_1 \} P_4,5[\[\theta, \theta\]] ×
C_2 \{ U_{1,3}[m_2[]] \} U_6 \{ m_1[] \} \text{SWAP}_{4,5} C_{2,3}[\text{SWAP}_{0,1}] ×
\text{Damp}_{3}[\[\theta / 10^2\]] \text{Depol}_{5}[\[\theta / 10^3\]] \text{Depol}_{1,2}[\[\theta / 10^2\]] ×
\text{Kraus}_{0}[\{ m_1[] \} \} \text{Kraus}_{1,4}[\{ m_2[] \}]
]
ApplyCircuit[\[\theta, u[\pi / 3]\]]
```

The result of Circuit[] is just a list of operators, allowing easy circuit manipulation. Mathematica features many tools for such lists allowing the user to easily extend, alter, join, compare, etc, their quantum circuit. For example:
To demonstrate the concision [11] offered by QuESTLink, we provide several examples of somewhat sophisticated computations written in only several lines, and efficiently simulated. For users wishing to run these demonstrations directly, they are compiled into a single notebook at questlink.qtechtheory.org/paper_demos.nb.

### A. Decoherence

To begin, we very compactly demonstrate the effect of two-qubit depolarising noise on the expected measurement of a simple Hamiltonian $h$. Starting in a random pure state $\psi$, a depolarising channel with probability 0.1 of any Pauli error occurring is repeatedly applied, in total 100 times.

```mathematica
{ψ, φ} = CreateQuregs[5, 2];
{ρ, σ} = CreateDensityQuregs[5, 2];
SetQuregMatrix[ψ, Normalize@Table[RandomComplex[], 2^5]];
InitPureState[ρ, ψ];

h = .3 + .1 X0 Y1 Z2 - .2 Z0;

data = Table[
   MixTwoQubitDepolarising[ρ, 0, 1, .1];
   {CalcFidelity[ρ, ψ], CalcExpecPauliSum[ψ, h, φ] -
   CalcExpecPauliSum[ρ, h, σ]},
   100];

ListLinePlot[Transpose[data], opts]
```

Note that undisclosed variable `opts` contained additional code for customising the plot.

With these facilities, QuESTLink offers a seamless integration with Mathematica’s comprehensive range of computational and graphical tools, as illustrated in the following demonstrations.
B. Variational imaginary-time

In this demonstration, we emulate the quantum variational imaginary-time simulation routine [12] to approximate the ground-state of a molecular Hamiltonian.

We first download a reduced 6-qubit representation of the electronic structure Hamiltonian of Lithium Hydride (LiH).

```
nQb = 6;
h = GetPauliSumFromCoeffs[ "https://qtechtheory.org/hamil_6qbLiH.txt"]
h[[-2 ;; 3]]
0.0591748 Z[2] Z[2] Z[5] + 0.147366 Z[3] Z[4] Z[5]
```

We create a simple 6-qubit ansatz circuit featuring 39 parameters, denoted with variables \( \vec{\theta} = \{\theta_1, \ldots, \theta_{39}\} \).

```
entangle[qbs_] :=
Table[R[\[Theta], \[Sigma]q \[Sigma]q+1], \{\[Sigma], \{X, Y, Z\}\}, \{q, qbs\}]
gates = Flatten @ Join[
Table[op \[Theta], \{op, \{Rz, Ry, Rx, Rz\}\}, \{q, nQb\}],
entangle[\{1, 3\}]];
ansezatz = MapIndexed[#1 /. \[Theta] \[Rule] \[Theta] #2〚1〛 &, gates];
n\[Theta] = Length[ansatz];
```

This ansatz circuit consists of one and two qubit rotation gates; \( \exp(-i\vec{\theta}/2) \), for \( \vec{\theta} \in \{X, Y, Z, X \otimes X, Y \otimes Y, Z \otimes Z\} \), which in QuESTLink are denoted by \( \text{Rx}\_q[\theta] \) (etc.) and \( \text{R}[\theta, X, X, X] \). In the diagram below, each such paired two-qubit rotation is marked by a vertical link.

```
DrawCircuit[ansatz, nQb]
```

We now create several Quregs with which to emulate the quantum algorithm.

```
{\[Psi], \[Psi], \[Phi]} = CreateQuregs[nQb, 3];
d\[Psi] = CreateQuregs[nQb, n\[Theta]];
```

\( \psi \) will be maintained as the output state of the ansatz circuit, and \( \hbar \psi \) will store the result of applying the Hamiltonian \( \hbar \) to \( \psi \). \( \phi \) will merely provide intermediate work-space for calculations, and \( d\psi \) will store a Qureg for each parameter in the ansatz (a total of \( n\theta \)).

Next, we choose a random initial assignment of the ansatz parameters, and measure the energy of the resulting quantum state.

```
cur\[Theta] = Table[\[Theta] \[Rule] RandomReal[], \{t, n\[Theta]\}];
ApplyCircuit[ansatz / cur\[Theta], \[Psi]]; CalcExpecPauliSum[\[Psi], h, \[Phi]]
-5.87043
```

The variational imaginary-time algorithm [12] involves repeatedly measuring a matrix and vector of quantities,

\[
A_{ij} = \text{Re} \left\{ \frac{\partial \psi(\vec{\theta})}{\partial \theta_i} \right\} \frac{\partial \psi(\vec{\theta})}{\partial \theta_j} \right\},
\]

and iteratively updating the parameters under

\[
\vec{\theta} \rightarrow \vec{\theta} + \Delta t A^{-1} \vec{C},
\]

which we now concisely emulate for \( nt \) iterations.

```
\Delta t = .1;
nt = 100;
Do[
InitZeroState[\[Psi]]; CalcQuregDerivs[ansatz, \[Psi], \[Psi], d\[Psi]]; matrA = CalcInnerProducts[\[Psi]] // Re;
ApplyCircuit[ansatz / cur\[Theta], \[Psi]]; ApplyPauliSum[\[Psi], h, \[Psi]]; vecC = -CalcInnerProducts[\[Psi], d\[Psi]] // Re;
\[Delta]\[Theta] = \[Delta] t LinearSolve[matrA, vecC]; cur\[Theta][All, 2] += \[Delta]\[Theta],
nt
]
```

The energy of the output quantum state, as produced by the reached assignment of the parameters, is close to the true groundstate found through matrix diagonalisation.

```
CalcExpecPauliSum[\[Psi], h, \[Phi]]
-7.87037
Min @ Eigenvalues @ CalcPauliSumMatrix @ h
-7.88074
```

Interestingly, a significantly slower but direct minimisation in Mathematica reveals the energy reached by
imaginary-time was not quite the best possible of our chosen ansatz.

\[
\text{energy}[\theta,\{\text{vars}\}] := \text{Module}[(\theta\rightarrow\{\text{vars}\}),\{\theta,\{\text{t},\{0\}\}\}; \text{NMaximize}\{\text{energy}\@\theta,\theta\}][1]
\]

We invite the interested reader to compare the Mathematica code above to a native C implementation of imaginary time evolution, as hosted on Github [13].

C. Noisy Trotterisation

In this demonstration, we emulate Trotterisation of a spin-ring Hamiltonian, with and without noise, and compare it to direct numerical solving of the Schrödinger equation. Our Hamiltonian is the one-dimensional nearest-neighbour (periodic boundary conditions) nQb-spin Heisenberg model with a random magnetic field in the z direction,

\[
\sum_{j} \sigma_{j}\sigma_{j+1} + r_{j}\sigma_{j}^{z}, \quad r_{j} \sim U[-1,1]. \tag{4}
\]

Real-time simulation of this model to time \( t = n\text{Qb} \) is nominated by Childs et al. [14] as an early practical application of a quantum computer. In this example, we’ll study a five-spin chain Hamiltonian \( h \).

\[
n\text{Qb} = 5;
\]

\[
h = \text{Flatten}\@\text{Join}[\text{Table}[1.0\&\text{Mod}[q,\text{Qb}],\{q,\{X, Y, Z\}\},\{q,\text{Qb}\}],\text{Table}[\text{RandomReal}[-1,1],\{q,\text{Qb}\}]]
\]

To simulate evolution on a quantum computer, we will emulate circuits formed by the Suzuki-Trotter decompositions [15] of the unitary evolution operator of varying order \( n \). Below, \( r \) is the number of repetitions of the order-\( n \) circuit to perform, to ultimately reach time \( t \).

\[
\text{symmetrize}[\lambda, 1] := \lambda h
\]

\[
\text{symmetrize}[\lambda, 2] := \text{With}[(s1 = \text{symmetrize}[h, \lambda/2, 1]), \text{Join}[s1, \text{Reverse}[s1]]]
\]

\[
\text{symmetrize}[\lambda, n] \text{EvenQ} := \text{Block}[\{y, \gamma = 1/(4 - 4^{(n-1)}), \text{With}[(s = \text{symmetrize}[h, \gamma, n-2]), \text{With}[(r = s \&/\&, \gamma \rightarrow \lambda)p], \text{Join}[r, r, r, \gamma \rightarrow (1 - 4p)\lambda, r, r]]]
\]

\[
\text{gateify}[(\text{Verbatim}[\text{Times}])[\text{\_}, \text{\_}]] := \text{R}[2\theta, \text{Times}[\sigma]]
\]

\[
\text{trotterize}[h, n, r, t] := \text{With}[(s = \text{symmetrize}[h, t/r, n]), \text{gateify}@\text{Flatten}@\text{ConstantArray}[s, r]]
\]

Note the original ordering of the Hamiltonian terms is arbitrary, and should ideally be optimised into commuting groups, or at least randomly shuffled. In particular, Childs et al.’s uniform randomisation scheme [16], whereby each repetition sees a random Hamiltonian ordering, is trivially implemented as:

\[
\text{childsify}[h, n, r, t] := \text{Flatten}@\text{Table}[\text{symmetrize}[\text{RandomSample}@h, t/r, n], r]
\]

Even Campbell’s qDRIFT routine [17] can be given a compact implementation.

\[
\text{campbellize}[h, n, r, t] := \text{With}[(c = h[\text{All}, 1], \sigma = h[\text{All}, 2];), \text{N = Length}[h] \star r, \text{With}((\lambda = \text{Total}[c], p = \text{Abs} \&\text{Normalize}[c, \text{Total}]), t\star\lambda/N \text{RandomChoice}[p \rightarrow \sigma, N])]}
\]

However, we opt instead to utilise only deterministic Trotterisation for clarity. For example, the first-order (\( n=1 \)) single-repetition (\( r=1 \)) Trotter circuit (arbitrarily to time \( t=1 \)) has the form:

\[
\text{DrawCircuit}@\text{trotterize}[h, 1, 1, n\text{Qb}]
\]

We compare the states produced from Trotter circuits with the “true” (to numerical precision) time evolution, found by numerically solving the Schrödinger equation using Mathematica’s in-built NDSolveValue routine.
In the proceeding solutions, the Qureg $\psi_0$ will store the initial state (a random pure state), $\psi$ will store the true state, and $\psi$ will store outputs of the Trotter circuits.

We can even imitate an noisy quantum device by inserting decoherence operators into our circuit. In this demonstration, we will follow each Trotter-prescribed gate with one or two qubit depolarising operators, which effect the channel

\[
\text{Depol}[\rho] = (1-p)\rho + \frac{p}{3} \sum_\sigma \sigma \rho \sigma, \quad (5)
\]

(note $\sigma = \sigma^\dagger$ for Paulis) and similarly for the two-qubit analogue. We choose an error rate of $p = 10^{-4}$, comfortably below the rates recently demonstrated in Google’s Sycamore machine [1]. Below is a visualisation of the first five unitary gates of the resulting noisy circuit.

Our simulation of the noisy circuit is near-identical to our simulation of pure states, excepting that we now operate upon density matrices.
\[
\{\rho, \rho_0\} = \text{CreateDensityQuregs}[\text{nQb}, 2];
\text{InitPureState}[\rho_0, \varnothing];
\]
\[
\text{nfids} = \text{Table}[
\quad \text{circ} = \text{noisify}[10^{-4}] \otimes \text{trotterize}[h, \text{order}, \text{reps}, \text{nQb}];
\quad \text{ApplyCircuit}[	ext{circ}, \rho_0, \rho];
\quad \{\text{Length}[	ext{circ}], \text{CalcFidelity}[ho, \varnothing]\},
\quad \{\text{order}, \{1, 2, 4, 6, 8\}\},
\quad \{\text{reps}, 1, 50\}];
\]

As one might expect, the accuracy of the Trotter circuits have waned, and eventually decrease with increased circuit depth, due to the opportunity for additional errors to accrue.

\[
\text{ListLogLinearPlot}[\text{nfids}, \text{opts}]
\]

D. Further examples

Complete notebooks demonstrating some extended computations in QuESTlink are available at questlink.qtechtheory.org.

V. Benchmarking

We now benchmark QuESTlink's local emulation of some arbitrary quantum circuits, and compare it to benchmarks of direct simulation in C, using QuEST. This helps elucidate the runtime overheads incurred by Mathematica integration, and whether QuESTlink is the right emulation tool for the user's target simulation regime.

We nominate a simple circuit consisting of \(X\), \(Y\), and \(Z\) axis rotations of random angles on each qubit, followed by tessellated controlled rotations on every pair of neighbouring qubits. This pattern is repeated in the circuit, and is illustrated in Figure 4. These benchmarks will emulate 15-qubit circuits, with between 1 to 50 repetitions (an upper bound of 4350 gates), and each will be simulated 10 times with re-randomised angles.

Core QuEST is profiled directly in C, through precise timing of each full circuit execution. In contrast, QuESTlink is profiled through a notebook using Mathematica’s \(\text{AbsoluteTiming}\) function; its runtime will include Mathematica evaluation, \text{QuESTlink.m} preprocessing and circuit encoding, the C/Link and WSTP overheads, \text{questlink.c}’s decoding of the circuit, and ultimately QuEST’s simulation.

Benchmarking is performed on a 12-core Xeon W-2133 3.6GHz CPU. At most 8 threads will be employed in multithreaded mode, so as not to interfere with threads used by the Mathematica kernel. GPU-accelerated testing will employ a 24 GB NVIDIA Quadro P6000 in the same machine.

The results of benchmarking are presented in Figure 5, and are for the most part, as expected. The Mathematica overhead of invoking serial QuEST through QuESTlink is small (a factor of \(\approx 1.1\)). Multithreading introduces additional variation in QuESTlink’s runtime, most likely due to dynamic reallocation of threads between the Math-
ematica kernel and the backend QuEST process, during execution. Otherwise, the multithreading overhead is similarly small (a factor $\approx 1.08$). At first glance, GPU QuESTlink appears anomalously slow; on average 7.2 times slower than core GPU QuEST. However, in the largest serial simulation, QuESTlink was on average $5 \times 10^{-2}$ seconds slower than QuEST. Treating this as an overhead unaffected by GPU-acceleration, GPU QuEST’s mean edge of $4 \times 10^{-2}$s over QuESTlink is less surprising. That is, while likely that multithreading benefited the Mathematica kernel in the pre-processing stage (e.g. in circuit encoding), GPU acceleration exemplified the expense of this overhead by contrast, rather than by enhancing it.

While not measured presently, use of a remote QuEST environment is expected to add a constant overhead to QuESTlink’s performance, due to network latency. Though in principle the data cost of communicating a circuit from Mathematica to a remote backend scales linearly with the emulated circuit depth, this cost should be overshadowed by the exponentially growing cost of quantum emulation, and other network overheads. The size of an encoded circuit is upper bounded by $8(\#\text{gates} + \#\text{ctrls} + \#\text{targs} + \#\text{params})$ bytes, where $\#\text{gates}$ is the total number of present gates, $\#\text{ctrls}$ and $\#\text{targs}$ are the total number of control and target qubits (respectively) aggregate over all gates, and $\#\text{params}$ is the total number of present parameters (e.g. angles of rotations). A circuit would need to contain approximately 30 million single-qubit gates to saturate a 1 GB bandwidth network.

VI. FUTURE WORK

QuESTlink is currently under active development, with a growing list of planned work and new features. This list includes:

- Support for emulation on remote, distributed hardware.
- Routines for symbolically evaluating quantum circuits, in lieu of numerical emulation, in order to study them analytically.
- A QASM [18] parser and generator, to and from QuESTlink’s circuit specification language.

We caution that QuESTlink is still in an early form and likely to change, both in interface and architecture.

VII. CONCLUSION

This manuscript introduced QuESTlink, a Mathematica package for emulating quantum circuits, state-vectors and density matrices. QuESTlink offers both high-level symbolic manipulation of quantum circuits, and rapid simulation using possibly remote hardware, such as multicore and GPU-accelerated supercomputers. We presented a broad technical overview of QuESTlink, including its protocol for stand-alone installation-free deployment. We then demonstrated the concision and flexibility possible of QuESTlink, by stepping through several examples of otherwise sophisticated simulations. These examples should enable an interested reader to begin using QuESTlink immediately. Lastly, we performed some simple benchmarks of QuESTlink to estimate the overhead over core QuEST, across its parallelisation modes. QuESTlink is open-source, and accessible at questlink.qtechtheory.org, or on Github [7].

VIII. ACKNOWLEDGEMENTS

The authors sincerely thank Suguru Endo and Balint Koczor for helping shape the functionality and direction of QuESTlink, bug-spotting, and help porting QuESTlink to Windows. Thanks also goes to the early adopters of QuESTlink in the QTechTheory group, for their part in moulding a perfectly cromulent emulator. TJ additionally thanks Quantum Motion Technologies Ltd for financial support in extending QuESTlink, and Sinan Shi for his expertise in rubber duck debugging. SCB acknowledges EPSRC grant EP/M013243/1. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing facility (ARC, please see http://dx.doi.org/10.5281/zenodo.22558) in enabling the related QuEST work.

[1] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. Nature, 574(7779):505–510, 2019.

[2] John Preskill. Quantum Computing in the NISQ era and beyond. Quantum, 2:79, August 2018.

[3] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. New Journal of Physics, 18(2):023023, 2016.
[4] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. QuEST and high performance simulation of quantum computers. *Scientific reports*, 9(1):1–11, 2019.
[5] Andrey Grozin. *Introduction to Mathematica® for Physicists*. Springer, 2014.
[6] Patrick T Tam. *A physicist’s guide to Mathematica*. Academic Press, 2011.
[7] Tyson Jones. QuESTLink (public Github repo). https://github.com/QTechTheory/QuESTLink, 2019.
[8] WSTP (Wolfram Symbolic Transfer Protocol). https://www.wolfram.com/wstp/. Accessed: 2019-12-05.
[9] Todd Gayley. A MathLink tutorial. *Wolfram Research*, 2002.
[10] How to — connect to a remote kernel, wolfram language & system, documentation center. https://reference.wolfram.com/language/howto/ConnectToARemoteKernel.html. Accessed: 2019-12-05.
[11] Brad Leithauser. In praise of concision. *The New Yorker*, Feb 2013.
[12] Sam McArdle, Tyson Jones, Suguru Endo, Ying Li, Simon C Benjamin, and Xiao Yuan. Variational ansatz-based quantum simulation of imaginary time evolution. *npj Quantum Information*, 5(1):1–6, 2019.
[13] Tyson Jones. Tiny Imaginary Time (public Github repo). https://github.com/TysonRayJones/TinyImagTime, 2019.
[14] Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences*, 115(38):9456–9461, 2018.
[15] N Hatano and M Suzuki. Finding exponential product formulas of higher orders, 2005.
[16] Andrew M. Childs, Aaron Ostrander, and Yuan Su. Faster quantum simulation by randomization. *Quantum*, 3:182, September 2019.
[17] Earl Campbell. Random compiler for fast hamiltonian simulation. *Physical review letters*, 123(7):070503, 2019.
[18] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017.