Computer Physics Communications 307 (2025) 109403

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computer Programs in Physics

VAN-DAMME: GPU-accelerated and symmetry-assisted quantum optimal control of multi-qubit systems ☆

José M. Rodríguez-Borbón [a,1], Xian Wang [a,1], Adrián P. Diéguez [b], Khaled Z. Ibrahim [b], Bryan M. Wong [a,*]

[a] Department of Chemistry, Department of Physics & Astronomy, and Materials Science & Engineering Program, University of California-Riverside, 900 University Avenue, Riverside, 92521, CA, USA

[b] Applied Mathematics & Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, 94720, CA, USA

ARTICLE INFO

Keywords:
Quantum optimal control
GPUs
Time-dependent Schrödinger equation
Parallelization
Gradient ascent optimization

ABSTRACT

We present an open-source software package, VAN-DAMME (Versatile Approaches to Numerically Design, Accelerate, and Manipulate Magnetic Excitations), for massively-parallelized quantum optimal control (QOC) calculations of multi-qubit systems. To enable large QOC calculations, the VAN-DAMME software package utilizes symmetry-based techniques with custom GPU-enhanced algorithms. This combined approach allows for the simultaneous computation of hundreds of matrix exponential propagators that efficiently leverage the intra-GPU parallelism found in high-performance GPUs. In addition, to maximize the computational efficiency of the VAN-DAMME code, we carried out several extensive tests on data layout, computational complexity, memory requirements, and performance. These extensive analyses allowed us to develop computationally efficient approaches for evaluating complex-valued matrix exponential propagators based on Padé approximants. To assess the computational performance of our GPU-accelerated VAN-DAMME code, we carried out QOC calculations of systems containing 10 - 15 qubits, which showed that our GPU implementation is 18.4× faster than the corresponding CPU implementation. Our GPU-accelerated enhancements allow efficient calculations of multi-qubit systems, which can be used for the efficient implementation of QOC applications across multiple domains.

Program summary
Program Title: VAN-DAMME
CPC Library link to program files:: https://doi.org/10.17632/zcgw2n5bjf.1
Licensing provisions: GNU General Public License 3
Programming language: C++ and CUDA
Nature of problem: The VAN-DAMME software package utilizes GPU-accelerated routines and new algorithmic improvements to compute optimized time-dependent magnetic fields that can drive a system from a known initial qubit configuration to a specified target state with a large (≈ 1) transition probability.
Solution method: Quantum control, GPU acceleration, analytic gradients, matrix exponential, and gradient ascent optimization.

1. Introduction

The past two decades have witnessed impressive progress toward realizable quantum computing, with several competing prototypes using various platforms such as transmons [1,2], fluxoniums [3,4], trapped ions [5,6], and Rydberg atoms [7]. One key challenge among all of these approaches is the difficult task of efficiently and accurately constructing time-dependent external control pulses that can drive these multi-qubit systems toward desired quantum states to enable massive calculations. The ultimate goal of quantum optimal control (QOC) approaches is to

☆ The review of this paper was arranged by Prof. W. Jong.

* Corresponding author.
E-mail address: bryan.wong@ucr.edu (B.M. Wong).
URL: http://www.bmwong-group.com (B.M. Wong).

[1] J. M. R.-B. and X. W. contributed equally to this work.

https://doi.org/10.1016/j.cpc.2024.109403
Received 6 August 2024; Received in revised form 11 October 2024; Accepted 14 October 2024
Available online 17 October 2024

construct these time-dependent external fields, and several frameworks, such as GRAPE [8], CRAB [9], and Krotov [10], have been previously used to calculate optimal control fields to enable desired gate operations or state transitions [11–13]. In all of these methods, the size of the $2^{n_q} \times 2^{n_q}$ Hamiltonian increases exponentially with the number of qubits, n_q, which prohibits the QOC calculations of large multi-qubit systems. For example, recent benchmarks have shown that 128 classical CPUs are required for QOC simulations of 10 qubits [14], whereas 12 qubits is the current limit for the GRAPE-based algorithm on a quantum-based processor [15]. Because the memory requirements of QOC problems increase exponentially with the number of qubits, recent methods have been proposed to address this difficulty [16,17]. For instance, by taking advantage of checkpoints and reversibility [17], the memory requirements of the standard GRAPE routine have been reduced by a factor of $2^{n_q} \times 2^{n_q} \times C$, with C proportional to the number of time steps. To address this computationally-intensive problem and further decrease these memory requirements, we present a new open-source software package, VAN-DAMME (Versatile Approaches to Numerically Design, Accelerate, and Manipulate Magnetic Excitations), which uses custom, massively-parallelized GPU acceleration schemes and symmetry-based techniques [14] to allow efficient QOC calculations of multi-qubit systems containing up to 15 qubits and beyond.

To maximize the computational efficiency of the VAN-DAMME code, we present several extensive tests on data layout, computational complexity, memory requirements, and performance. These extensive analyses allowed us to develop computationally efficient approaches for evaluating the discretized propagator (i.e., the exponential of a complex-valued matrix), since this is the most time-consuming operation in the QOC algorithm. To accelerate these calculations, we developed customized GPU routines and utilized existing routines in the CUDA BLAS library [18,19] to evaluate hundreds of matrix exponentials in parallel with highly accurate Padé approximants [20–22] and numerical scaling techniques (to ensure numeric robustness) [23]. For additional computational performance, we also parallelized all of the other computationally intensive matrix operations, such as matrix initialization, matrix·matrix multiplications, and matrix·vector multiplications, on GPUs. To assess the computational performance of our GPU-accelerated VAN-DAMME code, we carried out QOC calculations of systems containing 10 - 15 qubits using state-of-the-art A100 GPUs on the *Perlmutter* supercomputer at the National Energy Research Scientific Computing Center (NERSC) [24]. Our work concludes with a detailed discussion of performance analyses and timings of the VAN-DAMME code, which show that our GPU-accelerated approaches are 18.4× faster than the corresponding CPU implementation.

2. Theory and computational methodology

The time-dependent dynamics of a multi-qubit system is governed by the time-dependent Schrödinger equation:

$$i\frac{\partial}{\partial t}|\psi(t)\rangle = \left(H_0 + H_c(t)\right)|\psi(t)\rangle, \tag{1}$$

where H_0 is the static Hamiltonian and $H_c(t)$ is the time-dependent control Hamiltonian. The static Hamiltonian of an n_q-qubit system is defined as

$$H_0 = B_z \cdot \frac{1}{2}\sum_{i=1}^{n_q}\sigma_z^{(i)} + c_{\text{cpl}}^{(1)} \cdot \frac{1}{4}\sum_{i=1}^{n_q}\sigma_z^{(i)}\sigma_z^{(i+1)} + c_{\text{cpl}}^{(2)} \cdot \frac{1}{4}\sum_{i=1}^{n_q}\sigma_z^{(i)}\sigma_z^{(i+2)} + \dots + c_{\text{cpl}}^{(\lfloor \frac{n_q}{2} \rfloor)} \cdot \frac{1}{4}\sum_{i=1}^{n_q}\sigma_z^{(i)}\sigma_z^{(i+\lfloor \frac{n_q}{2} \rfloor)}, \tag{2}$$

where the first term reflects the interaction between the qubits and a static magnetic field, B_z, along the z-direction. The other terms represent the nearest-neighbor coupling, next-nearest-neighbor coupling, etc., and $c_{\text{cpl}}^{(1)}, c_{\text{cpl}}^{(2)}, \dots, c_{\text{cpl}}^{(\lfloor \frac{n_q}{2} \rfloor)}$ are the coupling coefficients. The control Hamiltonian,

$$H_c(t) = B_x(t) \cdot \frac{1}{2}\sum_{i=1}^{n_q}\sigma_x^{(i)} + B_y(t) \cdot \frac{1}{2}\sum_{i=1}^{n_q}\sigma_y^{(i)}, \tag{3}$$

manipulates all the qubits simultaneously with time-dependent control pulses, $B_x(t)$ and $B_y(t)$, along the x- and y-axes, respectively, throughout the control duration $[0, T]$.

To enable a numerical solution of Eq. (3), we discretize the control duration into N timesteps of duration τ, where $\tau = \frac{T}{N}$. The amplitude of the control pulse along the x- and y-axis at $t = (j + \frac{1}{2})\tau$ is denoted by $B_x[(j + \frac{1}{2})\tau]$ and $B_y[(j + \frac{1}{2})\tau]$, respectively. Given the state, $|\psi_j\rangle$, at $t = j\tau$ and the control pulses, the state $|\psi_{j+1}\rangle$ at $t = (j + 1)\tau$ can be obtained with the exponential propagator, i.e.,

$$|\psi_{j+1}\rangle = \exp\left(-i\tau\left(H_0 + H_c[(j + \tfrac{1}{2})\tau]\right)\right)|\psi_j\rangle, \tag{4}$$

where

$$H_c[(j + \tfrac{1}{2})\tau] = B_x[(j + \tfrac{1}{2})\tau] \cdot \frac{1}{2}\sum_{i=1}^{n_q}\sigma_x^{(i)} + B_y[(j + \tfrac{1}{2})\tau] \cdot \frac{1}{2}\sum_{i=1}^{n_q}\sigma_y^{(i)} \tag{5}$$

is the control Hamiltonian at $t = (j + \frac{1}{2})\tau$. The goal of the VAN-DAMME software package is to solve for the time-dependent optimal control fields, $B_x(t)$ and $B_y(t)$, that evolve the multi-qubit system toward a desired target state, $|\psi_{\text{target}}\rangle$. Accordingly, we define the loss function as the following transition probability:

$$P(|\psi_N\rangle) = |\langle\psi_{\text{target}}|\psi_N\rangle|^2, \tag{6}$$

where $|\psi_N\rangle$ is the final state at the Nth timestep.

The VAN-DAMME software package evaluates the analytic gradients of $P(|\psi_N\rangle)$ with respect to B_x and B_y at each timestep, i.e., $\frac{\mathrm{d}P(|\psi_N\rangle)}{\mathrm{d}B_x}\Big|_{(j+\frac{1}{2})\tau}$ and $\frac{\mathrm{d}P(|\psi_N\rangle)}{\mathrm{d}B_y}\Big|_{(j+\frac{1}{2})\tau}$. The gradients at the last timestep, i.e., $\frac{\mathrm{d}P(|\psi_N\rangle)}{\mathrm{d}B_x}\Big|_{(N-\frac{1}{2})\tau}$ and $\frac{\mathrm{d}P(|\psi_N\rangle)}{\mathrm{d}B_y}\Big|_{(N-\frac{1}{2})\tau}$, are evaluated first. Next, similar to the backpropagation method in neural networks, the VAN-DAMME software package recursively evaluates the gradients at the previous timestep using the gradients that have already been evaluated [8,11]. The control pulses are updated iteratively with the expressions

$$\begin{aligned} B_x^{(l+1)}[(j + \tfrac{1}{2})\tau] &= B_x^{(l)}[(j + \tfrac{1}{2})\tau] + \gamma\frac{\mathrm{d}P(|\psi_N\rangle)}{\mathrm{d}B_x^{(l)}}\Big|_{(j+\frac{1}{2})\tau}, \\ B_y^{(l+1)}[(j + \tfrac{1}{2})\tau] &= B_y^{(l)}[(j + \tfrac{1}{2})\tau] + \gamma\frac{\mathrm{d}P(|\psi_N\rangle)}{\mathrm{d}B_y^{(l)}}\Big|_{(j+\frac{1}{2})\tau}, \end{aligned} \tag{7}$$

where l is the iteration, and γ is the update rate evaluated with a golden section search algorithm.

Algorithm 1 summarizes our gradient-based approach. In each iteration of this routine, the propagation of the multi-qubit state in line 7 requires the computation of several thousand complex-valued matrices (i.e., $\mathbf{A}[j] = -i\tau(H_0 + H_c[(j + \frac{1}{2})\tau])$ for $j = 0, \dots, N - 1$), the computation of several thousand matrix exponentials (i.e., $e^{\mathbf{A}[j]}$ for $j = 0, \dots, N - 1$), and the multiplication of these matrix exponentials by a vector as shown in Eq. (4). In Eq. (4), because the matrices H_0 and H_c can be computed in advance (i.e., for each iteration, the B_x and B_y vectors, as well as the H_x and H_y matrices are given), hundreds of matrix exponentials can be calculated simultaneously. The execution of line 8 involves the computation of a complex dot product and the computation of a norm. The implementation of line 10 requires the computation of a large number of complex-valued matrix exponentials, matrix·matrix multiplications, and matrix·vector multiplications. The implementation of line 11 is similar to the implementation of line 7. Finally, the implementation of line 12 requires the addition of vectors. Note that billions of arithmetic operations are needed for each iteration in this routine. In addition, multiple iterations are required to achieve convergence.

The complex-valued matrix operations in Algorithm 1 are time-consuming. For example, as shown on line 7, a different matrix exponential is computed in each time step. Matrix exponentials are also evaluated on line 12. Since these matrix exponentials share the same problem size and memory requirements, there is a compelling case for offloading these computations to GPUs and executing them in batches. Consequently, the VAN-DAMME code leverages the high-performance numerical capabilities of GPUs to handle the majority of these calculations [25]. This approach encompasses all matrix-related operations, including matrix initializations, matrix exponentials, matrix·matrix and matrix·vector multiplications, as well as various linear algebra tasks, resulting in enhanced efficiency and reduced computational burden.

Algorithm 1: Quantum control algorithm in the VAN-DAMME software package.

Input: Control duration T, time step τ, initial magnetic control pulses $B_x[(j+\frac{1}{2})\tau]$ and $B_y[(j+\frac{1}{2})\tau]$ for $j=0,\dots,N-1$, static Hamiltonian H_0, control Hamiltonians $H_x=\frac{1}{2}\sum_{i=1}^{n_q}\sigma_x^{(i)}$ and $H_y=\frac{1}{2}\sum_{i=1}^{n_q}\sigma_y^{(i)}$, initial state $|\psi_{\text{init}}\rangle$, desired target state $|\psi_{\text{target}}\rangle$, threshold probability δ, and maximum number of iterations Max.

Output: Final propagated wavefunction, $|\psi_N\rangle$, and optimized magnetic pulses $B_x[(j+\frac{1}{2})\tau]$ and $B_y[(j+\frac{1}{2})\tau]$ for $j=0,\dots,N-1$.

1 /* **Working with zero-based indexing** */
2 Transform the Hamiltonians H_0, H_x, and H_y with the symmetry-assisted approach as described in Sec. 3
3 Initialize $B_x[(j+\frac{1}{2})\tau]$ and $B_y[(j+\frac{1}{2})\tau]$ for $j=0,\dots,N-1$
4 $P=0$; $l=0$; $|\psi_0\rangle=|\psi_{\text{init}}\rangle$
5 **while** $l<Max$ ***and*** $P<\delta$ **do**
6 **for** $j=0$ ***to*** $N-1$ **do**
7 Calculate $|\psi_{j+1}\rangle$ using Eq. (4)
  end
8 Update P using Eq. (6)
9 **for** $j=N-1$ ***to*** 0 **do**
10 Calculate $\frac{dP(|\psi_N\rangle)}{dB_x}\Big|_{(j+\frac{1}{2})\tau}$ and $\frac{dP(|\psi_N\rangle)}{dB_y}\Big|_{(j+\frac{1}{2})\tau}$ using backpropagation
  end
11 Calculate update rate, γ, using golden section search
12 Update vectors $B_x[(j+\frac{1}{2})\tau]$ and $B_y[(j+\frac{1}{2})\tau]$ using Eq. (7)
13 $l=l+1$
  end
14 **return** $|\psi_N\rangle$, $B_x[(j+\frac{1}{2})\tau]$, and $B_y[(j+\frac{1}{2})\tau]$ for $j=0,\dots,N-1$

3. Symmetry-assisted Hamiltonian reduction

While the VAN-DAMME software package can execute QOC calculations of arbitrary qubit configurations, a significant computational speedup for QOC calculations of multi-qubit systems with permutation (S) or dihedral (D) group symmetry [14] can be obtained, which we focus on in this section. In these situations, the Hilbert space $\mathcal{H}(\mathbb{C}^{2^{n_q}})$ is decomposed into orthogonal subspaces. The evolution of the quantum state is restricted in each subspace as long as the symmetry of the multi-qubit system is preserved. The VAN-DAMME code can take advantage of the symmetry of the multi-qubit system to block diagonalize the Hamiltonians such that the evolution of the multi-qubit system can be calculated with each block separately. In particular, when the initial and target states lie in only one subspace, quantum control calculations are needed for only one block instead of the entire Hamiltonian.

As representative examples to demonstrate the capabilities of the VAN-DAMME software package, we consider two typical entangled states, the W state [26] and the GHZ (Greenberger–Horne–Zeilinger) state [27] given by

$$|W\rangle=\frac{1}{\sqrt{n_q}}\left(|100\dots0\rangle+|010\dots0\rangle+\cdots+|00\dots10\rangle+|00\dots01\rangle\right) \tag{8}$$

and

$$|\text{GHZ}\rangle=\frac{1}{\sqrt{2}}\left(|0\rangle^{\otimes n_q}+|1\rangle^{\otimes n_q}\right) \tag{9}$$

in configurations containing up to 15 qubits. These two states are representatives of the two non-biseparable classes of three-qubit states and cannot be transformed into each other by local operations. These specific states were proposed to prove Bell's theorem and were later introduced in various applications, including robust quantum memory storage [28], quantum key distribution [29], and achieving the Heisenberg limit to estimate error reduction in quantum metrology [30–32]. We define the initial state as "all spin-up," i.e., $|0\rangle^{\otimes n_q}$. Both the initial and target states lie in the first subspace under either S or D symmetry; therefore, we only use the first block of the transformed Hamiltonian. Table 1 shows a comparison of the dimensions between the complete Hilbert space $\mathcal{H}(\mathbb{C}^{2^{n_q}})$ and the first subspace $\mathcal{H}_1^D$ under D symmetry, which equals the size of the complete Hamiltonian and the first block, respectively. Thus, working with one block greatly reduces the computational complexity of our algorithm. For example, in the case of 15 qubits, the VAN-DAMME code executes quantum control calculations on reduced matrices of size $1{,}224\times1{,}224$ rather than the full $32{,}768\times32{,}768$ matrices; i.e., the complexity is $\mathcal{O}(\frac{2^{n_q}}{n_q})$ instead of $\mathcal{O}(2^{n_q})$.

It is worth noting that a desired transition may not be allowed even if the initial and target states lie in the same subspace since the selection rules of allowed transitions in each subspace are determined by the system's symmetry and coupling features. When there is no coupling, i.e., $c_{\text{cpl}}^{(1)},c_{\text{cpl}}^{(2)},\dots,c_{\text{cpl}}^{(\lfloor\frac{n_q}{2}\rfloor)}=0$ in Eq. (2), the multi-qubit system has S symmetry. The resonance frequency that excites all allowed transitions is degenerate under S symmetry, which forbids transitions to the W or GHZ state. The degeneracy of these resonance frequencies can be fully broken by enabling all the coupling terms in Eq. (2). As such, the symmetry of the multi-qubit system is reduced to D, and each transition allowed by the selection rule can be excited by pulses with a different resonance frequency. Our previous studies have shown that breaking the degeneracy of the resonance frequencies allows the multi-qubit system to be more easily controllable [14]. In principle, when full coupling is enabled, any transition can be realized as long as the initial and target states are in the same symmetry-protected subspace.

In addition to preparing the W and GHZ states in fully coupled multi-qubit systems, we benchmark the controllability with arbitrary transitions confined in the same subspace. Our tests on multiple transitions between randomly designated states indicate that the algorithms in the VAN-DAMME code can always converge if the initial and the target states are in the same subspace. This convergence arises because the full coupling between qubits brings enhanced controllability to the system, as discussed above. We have built the API of the VAN-DAMME application so that users can define their own initial and target states.

To enable faster convergence, the VAN-DAMME code uses the amplified gradient modification scheme originally proposed from the TRAVOLTA package [13]. Within this approach, the amplification coefficient is defined as

$$\begin{aligned}\beta&=\tilde{\beta}\cdot\frac{|\langle\psi_\perp|\psi_{\text{target}}\rangle|}{|\langle\psi_\perp|\psi_N\rangle|} && \text{if } |\langle\psi_\perp|\psi_N\rangle|<0.01\cdot|\langle\psi_\perp|\psi_{\text{target}}\rangle|\,,\\ &=1 && \text{if } |\langle\psi_\perp|\psi_N\rangle|\geq0.01\cdot|\langle\psi_\perp|\psi_{\text{target}}\rangle|\,,\end{aligned} \tag{10}$$

where $|\psi_\perp\rangle$ is the normalized difference of the target state and its projection on the initial state:

$$|\psi_\perp\rangle=\frac{|\psi_{\text{target}}\rangle-|\psi_0\rangle\langle\psi_0|\psi_{\text{target}}\rangle}{\sqrt{1-|\langle\psi_0|\psi_{\text{target}}\rangle|^2}}, \tag{11}$$

and $\tilde{\beta}$ is an empirical coefficient that is assigned the following values for different transition tasks.

Table 1
Comparison of dimensions of spaces for $n_q = 10 - 15$. The dimension of $\mathcal{H}(\mathbb{C}^{2^{n_q}})$ and $\mathcal{H}_1^D$ is 2^{n_q} and $\mathcal{O}(\frac{2^{n_q}}{n_q})$, respectively.

Number of qubits, n_q	Complete Hilbert space $\mathcal{H}(\mathbb{C}^{2^{n_q}})$	First subspace $\mathcal{H}_1^D$ under D symmetry
10	1,024	78
11	2,048	126
12	4,096	224
13	8,192	380
14	16,384	687
15	32,768	1,224

$$\begin{aligned}\tilde{\beta} &= \sqrt{10^{-3}} && \text{if } \psi_{\text{target}} = |W\rangle\,,\\ &= \sqrt{10^{-9}} && \text{if } \psi_{\text{target}} = |\text{GHZ}\rangle\,,\\ &= \sqrt{10^{-3}\cdot 1000^{n_q-3}} && \text{if } \psi_{\text{target}} = |1\rangle^{\otimes n_q}\,.\end{aligned} \tag{12}$$

When P is small, the gradients in Eq. (7) are multiplied by the amplification coefficient to prevent an extremely large update rate, γ, in a golden section search algorithm. Our previous study has shown that this heuristic effectively reduces the cost of the search method [13], and, as a result, the number of iterations for achieving convergence.

When the initial and target states are $|0\rangle^{\otimes n_q}$ and $|\text{GHZ}\rangle$, respectively, these two states are not orthogonal to each other. As a result, $P(|0\rangle^{\otimes n_q})$ is not at a local minimum of the loss function. In this situation, the optimization problem becomes non-convex, which cannot be solved with the gradient ascent approach. To resolve this difficulty, the VAN-DAMME code uses the optimal control pulses for preparing $|1\rangle^{\otimes n_q}$ as an initial guess for preparing the GHZ state. Regardless of whether the target state is $|\text{GHZ}\rangle$ or $|1\rangle^{\otimes n_q}$, the expression, $|\psi_\perp\rangle = |1\rangle^{\otimes n_q}$, always holds, which implies that the optimal pulses for transitions to these two states must have the same resonance frequencies. As such, when the control pulses are initialized with the optimal pulses that prepare $|1\rangle^{\otimes n_q}$, only the amplitudes need to be updated to evolve the multi-qubit system toward the GHZ state. Our extensive tests with the VAN-DAMME code show that these approaches enable facile convergence.

4. The computation of matrix exponentials

As shown in Algorithm 1, the computation of the exponential of a matrix,

$$e^{\mathbf{A}t} = \sum_{k=0}^{\infty} \frac{(\mathbf{A}t)^k}{k!}, \tag{13}$$

is paramount in the acceleration of the quantum control of multi-qubit systems. Multiple methods [20–22] have been proposed for implementing Eq. (13), while few of these routines lead to accurate results. The Scale and Square method based on Padé approximants is accurate when properly implemented, which we briefly review below.

4.1. Padé approximants

The Padé approximate [22] to matrix $e^{\mathbf{A}}$ is defined as

$$R_{pq}(\mathbf{A}) = [D_{pq}(\mathbf{A})]^{-1} N_{pq}(\mathbf{A}), \tag{14}$$

where

$$N_{pq}(\mathbf{A}) = \sum_{k=0}^{p} \frac{(p+q-k)!p!}{(p+q)!k!(p-k)!}\mathbf{A}^k, \tag{15}$$

and

$$D_{pq}(\mathbf{A}) = \sum_{k=0}^{q} \frac{(p+q-k)!q!}{(p+q)!k!(q-k)!}(-\mathbf{A})^k. \tag{16}$$

In Eq. (14), the non-singularity of $D_{pq}(\mathbf{A})$ is guaranteed if p and q are large or if the eigenvalues of $\mathbf{A}$ are negative. Moreover, from Eqs. (15) and (16), it should be noted that $N_{qq}(\mathbf{A}) = D_{qq}(-\mathbf{A})$. Moler et al. [20,21] have shown that Padé approximants are accurate when the norm of $\mathbf{A}$ is small. In addition, they have shown that diagonal approximants $p = q$ are preferred over off-diagonal approximants $p \neq q$.

4.2. Scaling and squaring

Because Padé approximants are very accurate when the norm of $\mathbf{A}$ is small, scaling and squaring methods have been proposed [22]. In this approach, the elements of the $\mathbf{A}$ matrix are first scaled down by a factor m so that $R_{pq}(\mathbf{A}/m)$ is a good approximation to $e^{\mathbf{A}/m}$. Then, the $R_{pq}(\mathbf{A}/m)$ matrix is raised to the mth power such that $e^{\mathbf{A}} = (R_{pq}(\mathbf{A}/m))^m$. The scaling and power-raising methods exploit the property of the exponential function:

$$e^{\mathbf{A}} = (e^{\mathbf{A}/m})^m. \tag{17}$$

Furthermore, to speed up the computations in Eq. (17), m is usually set to $m = 2^j$ with $j \geq 0$. If the $e^{\mathbf{A}/2^j}$ matrix is computed via $R_{qq}(\mathbf{A}/2^j)$ as described in Eq. (14), the parameters q and j have to be determined. The work of Moler et al. [20,21] shows that if $||\mathbf{A}|| \leq 2^{j-1}$,

$$[R_{qq}(\mathbf{A}/2^j)]^{2^j} = e^{\mathbf{A}+\mathbf{E}}, \tag{18}$$

where

$$\frac{||\mathbf{E}||}{||\mathbf{A}||} \leq \epsilon(q,j). \tag{19}$$

Eq. (19) can be used to determine the values of q and j. That is, given a tolerance error, ϵ, and the norm, $||\mathbf{A}||$, multiple values of q and j can be tabulated using an error function as described in Ref. [22]. Moreover, it is sensible to pick the resulting pair (q, j) that minimizes the amount of work in the calculation of $(R_{qq}(\mathbf{A}/m))^m$.

Furthermore, Higham [23] showed that it is advantageous to set $q = 13$ and choose j such that $||\mathbf{A}||_{\max}/2^j \leq \theta$ where θ is a pre-calculated constant ($||\mathbf{A}||_{\max}$ is the element of $\mathbf{A}$ with the maximum magnitude). By doing so, the relative error introduced by the scaling and power-raising method is bounded by 1.1×10^{-16}, a factor proportional to the round-off error in IEEE double-precision arithmetic. Setting lower values for q (or relaxing the constraints for θ) decreases the computational complexity of the calculations at the expense of lower accuracy. Since accuracy is paramount in our work, we set $q = 13$ and compute θ as mentioned previously, at the expense of this additional computational cost.

4.3. CPU routine

The first step in the computation of $e^{\mathbf{A}}$ is the computation of the $N_{qq}(\mathbf{A})$ and $D_{qq}(\mathbf{A})$ matrices. Efficient ways to compute these matrices have been proposed [22,23], which we briefly describe below. The matrix, N_{pq}, can be computed with the expression

$$N_{pq}(\mathbf{A}) = b_0\mathbf{I} + b_1\mathbf{A} + b_2\mathbf{A}^2 + b_3\mathbf{A}^3 + b_4\mathbf{A}^4 + \cdots + b_{12}\mathbf{A}^{12} + b_{13}\mathbf{A}^{13}. \tag{20}$$

Defining

$$\begin{aligned}\mathbf{V} &= b_0\mathbf{I} + b_2\mathbf{A}^2 + b_4\mathbf{A}^4 + b_6\mathbf{A}^6 + b_8\mathbf{A}^8 + b_{10}\mathbf{A}^{10} + b_{12}\mathbf{A}^{12}\\ &= b_0\mathbf{I} + b_2\mathbf{A}^2 + b_4\mathbf{A}^4 + b_6\mathbf{A}^6 + \mathbf{A}^6(b_8\mathbf{A}^2 + b_{10}\mathbf{A}^4 + b_{12}\mathbf{A}^6)\end{aligned} \tag{21}$$

and

$$\begin{aligned}\mathbf{U} &= b_1\mathbf{A} + b_3\mathbf{A}^3 + b_5\mathbf{A}^5 + b_7\mathbf{A}^7 + b_9\mathbf{A}^9 + b_{11}\mathbf{A}^{11} + b_{13}\mathbf{A}^{13} \\ &= \mathbf{A}(b_1\mathbf{I} + b_3\mathbf{A}^2 + b_5\mathbf{A}^4 + b_7\mathbf{A}^6) + \mathbf{A}^7(b_9\mathbf{A}^2 + b_{11}\mathbf{A}^4 + b_{13}\mathbf{A}^6) \\ &= \mathbf{A}[(b_1\mathbf{I} + b_3\mathbf{A}^2 + b_5\mathbf{A}^4 + b_7\mathbf{A}^6) + \mathbf{A}^6(b_9\mathbf{A}^2 + b_{11}\mathbf{A}^4 + b_{13}\mathbf{A}^6)]\end{aligned} \tag{22}$$

gives

$$N_{qq}(\mathbf{A}) = \mathbf{V} + \mathbf{U}. \tag{23}$$

Since $N_{qq}(\mathbf{A}) = D_{qq}(-\mathbf{A})$, it follows that $D_{qq}(\mathbf{A}) = \mathbf{V} - \mathbf{U}$. The elements b_i, $i = 0, \ldots, 13$, are the coefficients shown in Eq. (15). Finally, once $N_{qq}(\mathbf{A})$ and $D_{qq}(\mathbf{A})$ are calculated, Eq. (14) has to be solved. Algorithm 2 shows the steps for the computation of $e^{\mathbf{A}}$ using Padé Approximants with scaling of the input matrix.

Algorithm 2: Computer implementation of the matrix exponential $e^{\mathbf{A}}$ using Padé Approximants.

Input: Parameter θ and $n \times n$ matrix $\mathbf{A}$.
Output: $n \times n$ matrix $e^{\mathbf{A}}$.
1 Determine s (a minimal integer number) such that $||\mathbf{A}||_{\max}/2^s \leq \theta$
2 Compute $\mathbf{A} = \mathbf{A}/2^s$
3 Compute $\mathbf{A}^2 = \mathbf{A}\,\mathbf{A}$
4 Compute $\mathbf{A}^4 = \mathbf{A}^2\,\mathbf{A}^2$
5 Compute $\mathbf{A}^6 = \mathbf{A}^4\,\mathbf{A}^2$
6 Compute $\mathbf{V} = b_0\mathbf{I} + b_2\mathbf{A}^2 + b_4\mathbf{A}^4 + b_6\mathbf{A}^6 + \mathbf{A}^6(b_8\mathbf{A}^2 + b_{10}\mathbf{A}^4 + b_{12}\mathbf{A}^6)$ (see Eq. (21))
7 Compute $\mathbf{U} = \mathbf{A}[b_1\mathbf{I} + b_3\mathbf{A}^2 + b_5\mathbf{A}^4 + b_7\mathbf{A}^6 + \mathbf{A}^6(b_9\mathbf{A}^2 + b_{11}\mathbf{A}^4 + b_{13}\mathbf{A}^6)]$ (see Eq. (22))
8 Compute $N_{qq}(\mathbf{A}) = \mathbf{V} + \mathbf{U}$
9 Compute $D_{qq}(\mathbf{A}) = \mathbf{V} - \mathbf{U}$
10 Solve the linear system of equations $D_{qq}(\mathbf{A})R_{qq}(\mathbf{A}) = N_{qq}(\mathbf{A})$ for $R_{qq}(\mathbf{A})$ (see Eq. (14))
11 **return** $e^{\mathbf{A}} = (R_{qq}(\mathbf{A}))^{2^s}$

In line 1 of this algorithm, $||\mathbf{A}||_{\max} = \max_{i,j} |a[i,j]|$ for $i = 0, \ldots, n-1$ and $j = 0, \ldots, n-1$. Otherwise, this algorithm is a straightforward implementation of the equations just described. The algorithm is fast and accurate for calculating only one matrix exponential, $e^{\mathbf{A}}$. In our work, we require u exponential matrices (i.e., $e^{\mathbf{A}[i]}$ for $i = 0, \ldots, u-1$) to be computed simultaneously. While a parallel version of Algorithm 2 can be implemented on CPUs without major effort, the VAN-DAMME software package uses custom algorithms that harness the computational power of GPUs [25,33], which we describe below.

4.4. GPU routine

As described in Section 2, our approach computes multiple matrix exponentials in parallel. Because the $\mathbf{A}[i]$ matrices for $i = 0, \ldots, u-1$, are small (i.e., 1,224 × 1,224 or smaller), it is advantageous to compute multiple exponentials $e^{\mathbf{A}[i]}$ simultaneously since computing a single matrix exponential does not generate enough work for one GPU or a multi-core CPU. The computation of multiple small matrix exponentials in parallel provides enough work to saturate the GPU computational resources. The computational field where hundreds or thousands of small problems are solved efficiently in parallel is known as Batched Computations [34–36,13]. In this field, the number of problems solved in parallel is called the batch size. Examples of batched computations include the simultaneous multiplication of hundreds of small matrices, the simultaneous factorization of hundreds of small matrices via the QR decomposition, and the simultaneous factorization of hundreds of small banded matrices via the LU decomposition. For multi-core platforms, multiple approaches have been used to solve small problems simultaneously. For instance, if there are n cores and m problems, each core solves an individual problem and continues until all tasks have been solved (the one-core-one-problem approach). In other methods, multiple cores collaborate to solve an individual problem (the multiple-core-one-problem approach).

The algorithm used in the VAN-DAMME software package for computing the exponentials of multiple matrices simultaneously on GPUs or multi-core CPUs is presented in Algorithm 3. In line 2, the $\mathbf{A}[i]$ matrices for $i = 0, \cdots, u-1$ are computed. As shown in Algorithm 1, for each iteration, the $\mathbf{H}_0$ and $\mathbf{H}_c$ matrices are computed first, and next, the $\mathbf{A}[i]$ matrices are obtained via matrix additions. In lines 3 and 4, if the magnitude of the maximum element of matrix $\mathbf{A}[i]$ is larger than θ, the elements of this matrix are scaled by a factor 2^{s_i} such that $||\mathbf{A}[i]||_{\max}/2^{s_i} \leq \theta$. In lines 5, 6, and 7, the $\mathbf{A}^2[i]$, $\mathbf{A}^4[i]$, and $\mathbf{A}^6[i]$ matrices are computed. The $\mathbf{V}[i]$ matrices are computed in multiple steps. First, in line 8, the $\mathbf{T}[i]$ matrices are computed in two steps: the matrix addition step, $b_8\mathbf{A}^2[i] + b_{10}\mathbf{A}^4[i] + b_{12}\mathbf{A}^6[i]$, and next, the matrix product step. Second, in line 9, the $\mathbf{V}[i]$ matrices are computed via matrix additions. Similar steps are used in the computation of the $\mathbf{U}[i]$ matrices as shown in lines 10 and 11. The linear systems described in line 14 are solved in multiple steps. In the first step, the $D_{qq}(\mathbf{A}[i])$ matrix is decomposed via the LU decomposition [37,22]. Specifically, the $\mathbf{P}[i]$, $\mathbf{L}[i]$, and $\mathbf{U}^*[i]$ matrices satisfying $\mathbf{P}[i]D_{qq}(\mathbf{A}[i]) = \mathbf{L}[i]\mathbf{U}^*[i]$ are computed, where $\mathbf{P}[i]$, $\mathbf{L}[i]$, and $\mathbf{U}^*[i]$ are the permutation, lower triangular, and upper-triangular matrices, respectively. In the next step, the inverse of the $D_{qq}(\mathbf{A}[i])$ matrices is calculated, i.e., $(D_{qq}(\mathbf{A}[I]))^{-1} = (\mathbf{P}[i]^{-1}\mathbf{L}[i]\mathbf{U}^*[i])^{-1} = \mathbf{U}^*[i]^{-1}\mathbf{L}[i]^{-1}\mathbf{P}[i]$. Finally, given $(D_{qq}(\mathbf{A}[I]))^{-1}$, finding $N_{qq}(\mathbf{A}[i])$ is immediate. In line 15, the $\mathbf{R}_{qq}(\mathbf{A}[i])$ matrices are scaled up as described in Eq. (17).

Algorithm 3: Computation of matrix exponentials, $e^{\mathbf{A}[i]}$, for $i = 0, \ldots, u-1$, using Padé Approximants.

Input: Batch size u, parameter θ, time step τ, and $n \times n$ matrices $\mathbf{H}_0$ and $\mathbf{H}_c$.
Output: $n \times n$ matrices $e^{\mathbf{A}[i]}$, for $i = 0, \ldots, u-1$.
1 /* **All of the steps below are executed in batches** */
2 Compute $\mathbf{A}[i]$ for $i = 0, \ldots, u-1$ (see Eq. (4))
3 Compute s_i such that $||\mathbf{A}[i]||_{\max}/2^{s_i} \leq \theta$ for $i = 0, \ldots, u-1$
4 Compute $\mathbf{A}[i] = \mathbf{A}[i]/2^{s_i}$ for $i = 0, \ldots, u-1$
5 Compute $\mathbf{A}^2[i] = \mathbf{A}[i]\mathbf{A}[i]$ for $i = 0, \ldots, u-1$
6 Compute $\mathbf{A}^4[i] = \mathbf{A}^2[i]\mathbf{A}^2[i]$ for $i = 0, \ldots, u-1$
7 Compute $\mathbf{A}^6[i] = \mathbf{A}^4[i]\mathbf{A}^2[i]$ for $i = 0, \ldots, u-1$
8 Compute $\mathbf{T}[i] = \mathbf{A}^6[i]\left(b_8\mathbf{A}^2[i] + b_{10}\mathbf{A}^4[i] + b_{12}\mathbf{A}^6[i]\right)$ for $i = 0, \ldots, u-1$
9 Compute $\mathbf{V}[i] = b_0\mathbf{I} + b_2\mathbf{A}^2[i] + b_4\mathbf{A}^4[i] + b_6\mathbf{A}^6[i] + \mathbf{T}[i]$ for $i = 0, \ldots, u-1$
10 Compute $\mathbf{T}[i] = \mathbf{A}^6[i]\left(b_9\mathbf{A}^2[i] + b_{11}\mathbf{A}^4[i] + b_{13}\mathbf{A}^6[i]\right)$ for $i = 0, \ldots, u-1$
11 Compute $\mathbf{U}[i] = \mathbf{A}[i]\left(b_1\mathbf{I} + b_3\mathbf{A}^2[i] + b_5\mathbf{A}^4[i] + b_7\mathbf{A}^6[i] + \mathbf{T}[i]\right)$ for $i = 0, \ldots, u-1$
12 Compute $N_{qq}(\mathbf{A}[i]) = \mathbf{V}[i] + \mathbf{U}[i]$ for $i = 0, \ldots, u-1$
13 Compute $D_{qq}(\mathbf{A}[i]) = \mathbf{V}[i] - \mathbf{U}[i]$ for $i = 0, \ldots, u-1$
14 Solve $D_{qq}(\mathbf{A}[i])R_{qq}(\mathbf{A}[i]) = N_{qq}(\mathbf{A}[i])$ for $R_{qq}(\mathbf{A}[i])$ for $i = 0, \ldots, u-1$
15 **return** $e^{\mathbf{A}[i]} = \left[R_{qq}(\mathbf{A}[i])\right]^{2^{s_i}}$ for $i = 0, \ldots, u-1$

4.5. Implementation of Algorithm 3

In the following sections, we describe the technical implementation of Algorithm 3.

4.5.1. Data layout

There are many possibilities for choosing the data layout. For example, the multiple $\mathbf{A}[i]$ matrices for $i = 0, \ldots, u-1$, can be stored in contiguous or non-contiguous pools of GPU main memory. Because our routine has to allocate memory for multiple matrices, with each matrix having multiple instances (i.e., the $\mathbf{A}$ matrix needs memory for u instances: see Algorithm 3), all the $\mathbf{A}[i]$ matrices for $i = 0, \ldots, u-1$ are stored in contiguous memory in the VAN-DAMME software package. When the matrices are stored in contiguous pools of memory, only one

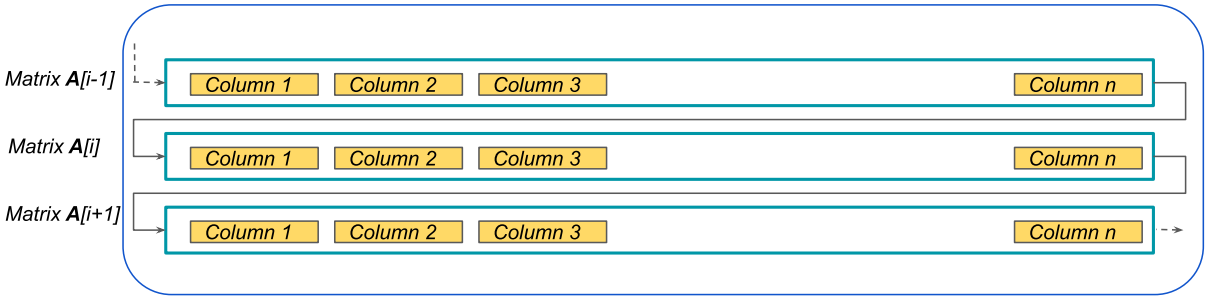

Fig. 1. Schematic of pool of memory of size $u \times n^2 \times$ sizeof(datatype) used in the VAN-DAMME software package.

memory allocation call is required. This approach saves time and minimizes the fragmentation of the memory. Next, we store the $\mathbf{A}[i]$ matrices for $i = 0, \ldots, u-1$ back to back. That is, given a pool of memory with size $u \times n^2 \times$ sizeof(datatype), matrix $\mathbf{A}[0]$ is stored first, $\mathbf{A}[1]$ next, and so on, with the $\mathbf{A}[u-1]$ matrix stored last. Moreover, to make use of the CUDA API, each $\mathbf{A}[i]$ matrix, as well as the other matrices in Algorithm 3, is represented in column-major order. Fig. 1 depicts our approach.

4.5.2. Implementation details

We next analyze the technical implementation of our quantum control algorithms in the VAN-DAMME software package. Fortunately, the latest CUDA BLAS library [19] supports many of the batch computations required in the implementation of this routine. These computations include (a) batched multiplication of matrices (i.e., $\mathbf{C}[i] = \alpha\mathbf{A}[i]\mathbf{B}[i] + \beta\mathbf{C}[i]$ for $i = 0, \ldots, u-1$), (b) batched LU decomposition of matrices (i.e., $\mathbf{L}[i]\mathbf{U}^*[i] = \mathbf{P}[i]\mathbf{A}[i]$ for $i = 0, \ldots, u-1$), and (c) batched inversion of matrices in the previous LU decomposition (i.e., $\mathbf{A}^{-1}[i] = [\mathbf{P}[i]^{-1}\mathbf{L}[i]\mathbf{U}^*[i]]^{-1}$ for $i = 0, \ldots, u-1$). Once computations (a), (b), and (c) are executed, the implementation of line 14 in Algorithm 3 is immediate: the LU decomposition of $D_{qq}(\mathbf{A}[i])$ is executed, and $(D_{qq}(\mathbf{A}[i]))^{-1}$ is computed followed by $R_{qq}(\mathbf{A}[i]) = (D_{qq}(\mathbf{A}[i]))^{-1} N_{qq}(\mathbf{A}[i])$.

Interestingly, CUDA BLAS neither implements the batched scaling operation nor the batched addition of matrices unless the batched data is stored contiguously in memory. Although these operations can be implemented on the CPU, we did a custom implementation of them on the GPU to minimize the data movement between the CPU and GPU (and vice-versa), which takes advantage of the thousands of cores on the target GPU. The implementation of line 3 requires finding the largest element of each $\mathbf{A}[i]$, and, next, the computation of s_i. Given s_i, the elements of $\mathbf{A}[i]$ must be scaled down as shown in line 4.

Fig. 2 shows the steps involved in the scaling of matrix $\mathbf{A}[i]$ (the rows of each matrix are depicted vertically). In part (a), ceil$(n/32)$ blocks are created with each GPU block having 32 threads. In this design, the first thread of the first block is responsible for finding the maximum element of the first row of $\mathbf{A}[i]$. The second thread of the first block finds the largest element of the second row of $\mathbf{A}[i]$, and so on. In short, the threads in the first GPU block find the largest elements in each of the first 32 rows of $\mathbf{A}[i]$. The threads in the second block compute the largest elements in each row of the next 32 rows. The other blocks compute the largest elements for the remaining rows of $\mathbf{A}[i]$. At the end, the largest elements in each row are written in the GPU main memory array $\mathbf{r}[i]$.

In part $(b.1)$, a GPU block with 32 threads is created. The threads in this block read the first 32 elements of array $\mathbf{r}[i]$ and write these elements into the shared memory array $\mathbf{m}[i]$ of size 32. Next, this block reads another 32 elements into registers. If the value in register one is larger than the first element in array $\mathbf{m}[i]$, the register is written into the first element of $\mathbf{m}[i]$ and likewise for the remaining 31 registers. This process of reading and comparing continues until all the elements of $\mathbf{r}[i]$ have been processed. At the end of this step, the array $\mathbf{m}[i]$ contains the largest element of $\mathbf{A}[i]$.

In part $(b.2)$, the largest element (i.e., $||\mathbf{A}||_{\max}$) of $\mathbf{m}[i]$ is found. This computation is done via a parallel reduction in shared memory. First, a GPU block with 32 threads is created. Next, the first thread of this block compares the first and second elements, and the maximum of those elements is written into the first position of $\mathbf{m}[i]$. The second thread compares the third and fourth elements of $\mathbf{m}[i]$, and the largest element is written into the third position of $\mathbf{m}[i]$. Likewise, the third thread compares the fifth and sixth elements of $\mathbf{m}[i]$, and the largest element is written into the fifth position of $\mathbf{m}[i]$. A similar process is used for the remaining 13 threads in the block (the remaining 16 GPU threads do not execute any work). At the end of this process, the threads in the block synchronize their work. The amount of work to find the largest element has been divided by two. Next, the step above repeats, but this time, only considers the first, third, fifth, seventh, ..., and thirty-first elements of $\mathbf{m}[i]$. This time, thread one compares elements one and three, thread two compares elements five and seven, and thread two compares elements nine and eleventh, and so on for the first eight threads in the block. The comparisons and synchronization continue until the largest element (denoted as s_i) of $\mathbf{m}[i]$ appears in the first position of $\mathbf{m}[i]$. In the scientific computing literature, this type of computation is known as Parallel Reduction via interleaved addressing [38]. Finally, given $||\mathbf{A}||_{\max}$, the scaling factor s_i is computed.

Finally, in part (c), the division of the elements of $\mathbf{A}[i]$ by 2^{s_i} is executed by another routine, which divides the elements of matrix $\mathbf{A}[i]$ into square blocks, reads the elements in the block, divides them by 2^{s_i}, and finally, writes the normalized elements back to the GPU main memory. The element s_i is saved for later calculations. For multiple matrices (i.e., $\mathbf{A}[i]$ with $i = 0, \ldots, u-1$), u routines are executed in parallel via a GPU kernel that uses the (x, y) dimensions to address the elements of the $\mathbf{A}[i]$ matrix and the z dimension to address individual matrices $z = 0, \ldots, u-1$.

As mentioned above, the cuBLAS library does not implement batched matrix additions. As a result, we implemented the operation $\mathbf{C}[i] = \alpha\mathbf{A}[i] + \beta\mathbf{C}[i]$ for $i = 0, 1, \ldots, u-1$. Our implementation is straightforward: the $\mathbf{C}[i]$ and $\mathbf{A}[i]$ matrices are split into 32×32 blocks (observing boundaries), and a GPU routine is called that adds these blocks. This routine reads two sub-blocks (one sub-block in $\mathbf{A}[i]$ and another in $\mathbf{C}[i]$), scales the blocks using the α and β parameters, respectively, and then, adds the scaled sub-blocks. Finally, the routine writes the resulting sub-blocks into the GPU main memory. As before, this GPU kernel uses the (x, y) dimensions to address the elements of the $\mathbf{A}[i]$ matrix and the z dimension to address individual matrices. To increase performance, the batched matrix addition, $\mathbf{C}[i] = \alpha_1\mathbf{A}[i] + \alpha_2\mathbf{B}[i] + \beta\mathbf{C}[i]$ for $i = 0, 1, \ldots, u-1$, is implemented in the same fashion.

4.5.3. Computational complexity

We now analyze the computational complexity of Algorithm 3. First, all operations are complex-valued, so our analysis is in the context of complex operations. For instance, multiplying two complex numbers requires four multiplications and two additions for a total of six arithmetic operations. To make our analysis simple, we count this multiplication as one complex operation. Table 2 summarizes our findings.

The implementation of line 2 requires $u \times \mathcal{O}(2n^2)$ complex multiplications and $u \times \mathcal{O}(n^2)$ complex additions. The implementation of line 3

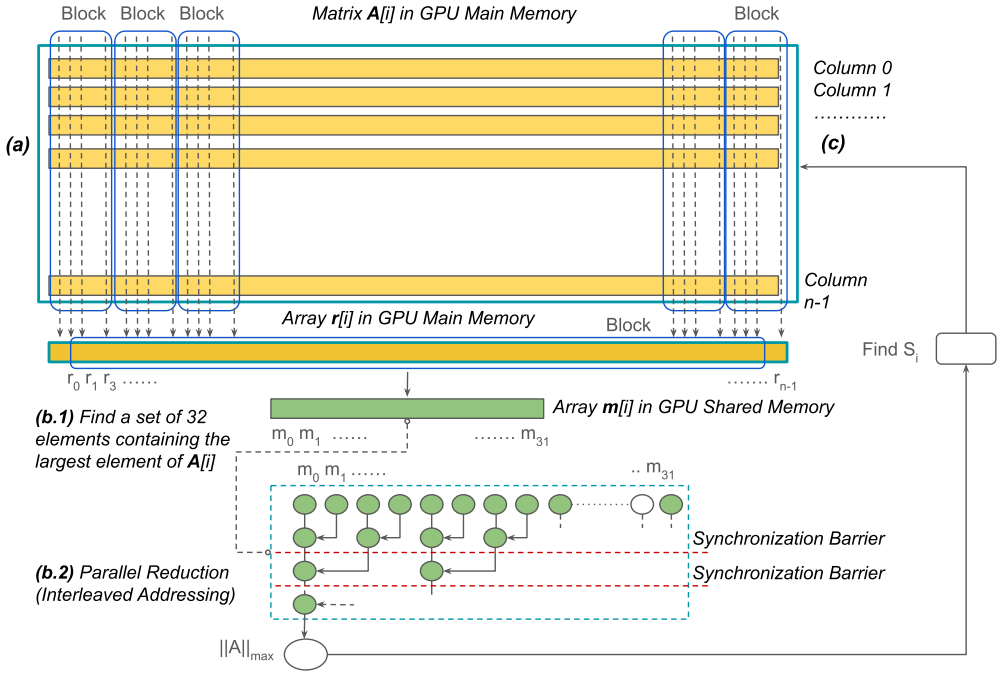

Fig. 2. GPU kernel for the normalization of the input $n \times n$ $\mathbf{A}[i]$ matrix in the VAN-DAMME software package. In part (a), the maximum value for each row is found, and this value is written into the GPU main memory array, $\mathbf{r}[i]$. In parts $(b.1)$ and $(b.2)$, a reduction in GPU shared memory finds the largest element $||\mathbf{A}||_{\max}$ in $\mathbf{r}[i]$. In part (c), the value s_i is used to compute $||\mathbf{A}[i]||/2^{s_i}$.

Table 2
Computational complexity (with double-precision complex numbers).

Line(s)	Complexity	Description
2	$u \times \mathcal{O}(3n^2)$	Matrix additions
3	$u \times \mathcal{O}(n^2) + uC$	Comparisons
4	$u \times \mathcal{O}(n^2)$	Divisions
5,6,7	$u \times \mathcal{O}(2n^3)$	Matrix multiplications
8	$u \times \mathcal{O}(2n^3 + 3n^2 + 2n^2)$	Matrix additions and multiplications
9	$u \times \mathcal{O}(5n^2 + 4n^2)$	Matrix additions
10, 11	Similar to lines 8 and 9	Matrix additions and multiplications
12,13	$u \times \mathcal{O}(n^2)$	Matrix additions
14	$u \times \mathcal{O}(2n^2/3 + 2n^2 + 2n^3 + 2n^3)$	Solving a linear system via LU decomposition
15	$(s_0 + s_1 + \cdots + s_{u-1})\mathcal{O}(2n^3)$	Matrix multiplications

requires about $u \times \mathcal{O}(n^2)$ operations and a constant number of operations. Specifically, for each $\mathbf{A}[i]$ matrix, the elements with the maximum norm have to be found first for each row, requiring $n(n-1)$ complex comparisons; next, s_i has to be computed. Finding the array $\mathbf{m}[i]$ requires $n-32$ comparisons, and the parallel reduction via interleaved addressing requires 31 comparisons. The implementation of line 4 requires $u \times \mathcal{O}(n^2)$ complex divisions in the worst case scenario. The number of arithmetic operations required to execute lines 5, 6, and 7 is proportional to that of matrix multiplication: $u \times \mathcal{O}(2n^3)$. The computation of line 8 requires one matrix multiplication and two matrix additions. The execution of the additions takes $3n^2$ complex multiplications and $2n^2$ complex additions. A similar analysis applies for lines 9, 10, and 11. Line 14 is very expensive to implement. First, executing the LU decomposition of a square matrix requires $2n^2/3$ complex operations [37]. Next, finding the inverses of $\mathbf{L}$ via forward substitution, or $\mathbf{U}^*$ via backward substitution, requires n^2 complex operations. In addition, given the inverses of $\mathbf{L}$ and $\mathbf{U}$, finding the inverse of $D_{qq}(\mathbf{A}[i])$ requires $2n^3$ complex operations as the post-multiplication by the matrix $\mathbf{P}[i]$ can be implemented via column exchanges. Solving for $N_{qq}(\mathbf{A}[i])$ requires $2n^3$ additional operations. Finally, the implementation of line 15 requires multiple matrix multiplications. Thus, the computation of the matrices $e^{\mathbf{A}[i]}$ is expensive, which can be accelerated with GPUs, as demonstrated in our results in Section 5.2.

4.5.4. Memory requirements

In this section, we analyze the memory requirements of the quantum control algorithms in the VAN-DAMME software package. For a batch size of u and each $n \times n$ $\mathbf{A}[i]$ matrix (for $i = 0, \ldots, u-1$) used in the quantum control algorithm, representing the set of $\mathbf{A}[i]$ matrices requires $u \times n^2 \times$ sizeof(datatype) bytes. In our case, the matrices $\mathbf{A}[i]$ are complex, and as a result, 16 bytes per element are required (i.e., complex double-precision floating point numbers). Table 3 summarizes the memory requirements of Algorithm 3 as a function of the number of qubits when the size of the batch is set to $u = 128$.

Table 3 does not include the $D_{qq}(\mathbf{A}[i])$, $N_{qq}(\mathbf{A}[i])$, and $R_{qq}(\mathbf{A}[i])$ matrices because once the $\mathbf{U}[i]$ and $\mathbf{V}[i]$ matrices are computed, the $\mathbf{A}^2[i]$, $\mathbf{A}^4[i]$, $\mathbf{A}^6[i]$ and $\mathbf{T}[i]$ matrices are deallocated, and as a result, additional memory is available. As can be seen in Table 3, both the size of the input matrices and the batch size cannot be increased simultaneously since the physical memory available on the platform is limited. In addition, Algorithm 1 requires extra memory due to the implementation of the backpropagation routine. For each iteration, the backpropagation method requires memory proportional to $N \times n \times$ sizeof(datatype) bytes. As previously mentioned, this implementation requires 16 bytes per element. Thus, the combined memory requirements of the matrix exponentials and the backpropagation routine account for over 95% of

Table 3
Memory utilization for 10 - 15 qubits with a Batch size $u = 128$.

Number of qubits	n	$\mathbf{A}[u]$, $\mathbf{A}^2[u]$, $\mathbf{A}^4[u]$, and $\mathbf{A}^6[u]$	$\mathbf{T}[u]$	$\mathbf{U}[u]$	$\mathbf{V}[u]$	Total memory
10	78	47.6 MB	11.9 MB	11.9 MB	11.9 MB	83.3 MB
11	126	124.0 MB	31.0 MB	31.0 MB	31.0 MB	217.0 MB
12	224	392.0 MB	98.0 MB	98.0 MB	98.0 MB	686.0 MB
13	380	1128.0 MB	282.0 MB	282.0 MB	282.0 MB	1974.0 MB
14	687	3591.2 MB	897.8 MB	897.8 MB	897.8 MB	6284.6 MB
15	1224	11704.4 MB	2926.1 MB	2926.1 MB	2926.1 MB	20482.7 MB

the overall memory needs of Algorithm 1, enabling the simulation of QOC problems involving 15 qubits or more.

5. Computational results and performance

5.1. Optimal controls for preparing the W and GHZ states

To demonstrate the capabilities of the VAN-DAMME software package, we calculate optimal control pulses that evolve a multi-qubit system with full coupling from $|0\rangle^{\otimes n_q}$ to the W or GHZ state. Figs. 3(a) and (b) compare the optimal control pulses between a 6- and 15-qubit system for preparing the W state. Note that the initial state $|0\rangle^{\otimes n_q}$ is next to the W state in the transition cascade of a fully coupled multi-qubit system [14]; i.e., the direct transition from $|0\rangle^{\otimes n_q}$ to the W state is allowed by the selection rule. As a result, this transition can be realized by control pulses of only one resonance frequency. The shapes of B_x and B_y are nearly identical and differ only by a $\frac{\pi}{2}$ phase shift; i.e., the transition is enabled by a circularly polarized magnetic control pulse. Figs. 3(c) and (d) compare the power spectra, i.e., the Fourier transform of the optimal control pulses, between the 6- and 15-qubit system. It is worth noting that the resonance frequency for the 15-qubit system is smaller than that for 6-qubits. In general, a redshift in the resonance frequency emerges as the number of qubits, n_q, increases due to the introduction of more coupling terms. Since the optimal control pulses are simple sinusoids, the probability P converges to ≈ 1.0 in a few iterations for both cases, as shown in Figs. 3(e) and (f).

Compared with the optimal control pulses that enable transitions from $|0\rangle^{\otimes n_q}$ to the W state, the optimal pulses for preparing the GHZ state are significantly more complicated, as shown in Figs. 4(a) and (b). In the power spectrum shown in Fig. 4(c), 14 resonance frequencies emerge for the 6-qubit system. In contrast, Fig. 4(d) indicates several hundreds of resonance frequencies are required to evolve a 13-qubit system toward the GHZ state since the initial $|0\rangle^{\otimes n_q}$ state and $|\psi_\perp\rangle = |1\rangle^{\otimes n_q}$ are at the two ends of the transition cascade [14]. As such, any transition allowed by the selection rule may be required to realize the transition from $|0\rangle^{\otimes n_q}$ to the GHZ state, and these transitions are enabled by hundreds of different resonance frequencies. Such complex optimal pulses pose difficulties for convergence, as shown in Figs. 4(e) and (f). Interestingly, if the pulses are initialized with white noise, the VAN-DAMME code is unable to converge to prepare the GHZ state. Therefore, we initialized the pulses with the optimal pulses used to prepare $|1\rangle^{\otimes n_q}$. This initial guess contains all the required frequency components and initializes $P = 0.5$ in the first iteration rather than zero. The VAN-DAMME code subsequently updates the amplitude for each resonance frequency and achieves convergence in 8 and 150 iterations for the 6 and 13-qubit systems, respectively.

In addition, we also calculated the optimal control pulses that prepare the W and GHZ states for fully coupled multi-qubit systems containing up to 15 and 13 qubits, respectively. The quantum control calculations for the 13-qubit GHZ state preparation was carried out on only the GPU since it was computationally prohibitive to calculate on the CPU (< 3 days on the GPU vs. ≈ 80 days on the CPU). The optimal pulses, power spectra, amplitude of the control pulses/gradients in each iteration, and the convergence of P vs. iteration are the same for the CPU and the CPU+GPU approaches, which demonstrate that our approach was accurately implemented in the VAN-DAMME software package. We compare the computational performance between the CPU and CPU+GPU approaches in Section 5.2.

5.2. Computational performance on GPUs

To demonstrate the computational performance of our GPU parallelization scheme in the VAN-DAMME code, we report the execution times of Algorithm 1 on one compute node of the *Perlmutter* supercomputer [24] at NERSC, which is equipped with one EPYC-7763 processor (a 64-core CPU) and 256 GB of RAM. In addition, each node houses four NVIDIA A100 GPUs, each having 40 GB of RAM. In our calculations, we set the number of CPU threads to eight (one thread per core) and use one GPU. Our testbed CPU [39] has a peak performance of 3.58 Teraflops per second and a maximum bandwidth of 204.8 GB/sec while the testbed GPU [40] has a peak performance of 9.7 Teraflops per second (FLOPS) and a maximum bandwidth of 2.0 terabytes per second (TB/s). To assess the computational performance of Algorithm 1 on different hardware architectures, we compare the execution times for two implementations: (1) a CPU baseline implementation that utilizes threaded numerical routines in the Cray BLAS LibSci library [41], and (2) our hybrid CPU+GPU implementation that utilizes the kernels as described in Section 4.4. The most time-consuming operation in Algorithm 1 is the calculation of matrix exponentials, which we further analyze below.

Fig. 5 shows the execution times for a single iteration of Algorithm 1 when applied to 10 - 15 qubits. To obtain the execution times of our baseline implementation (i.e., using CPUs only), we use one node on the *Perlmutter* supercomputer at NERSC. Because the complexity of the routine is dominated by the computation of matrix exponentials, we see proportional changes in the execution times: larger arguments in the matrix exponentials result in longer simulations. In our baseline CPU code, we observe that as the number of qubits increases, the execution time per iteration increases. On average, the increase in execution times is 3.6 for each additional qubit that is added. The minimum increase in execution times occurs when the number of qubits increases from 10 to 11 (a 2.5× increase). The maximum increase occurs when the number of qubits increases from 14 to 15 (a 4.9× increase). This is in agreement with our expectations: these time increases correspond to changes in the size of the matrix exponentials, as shown in Table 3, and as a result, rises in the computational complexity, as shown in Table 2.

To obtain the execution times of our hybrid implementation (i.e., 8 cores + 1 GPU), we used the same hybrid node on the *Perlmutter* supercomputer. In our implementation, all matrix operations are executed on the GPU. These operations include the computation of matrix exponentials, as shown in Algorithm 3, as well as other matrix operations required in Algorithm 1. As shown in Fig. 5, the observed results are in agreement with the expected behavior. On average, the increase in execution times is 3.9. The minimum increase in execution times occurs when the number of qubits increases from 10 to 11 (a 1.3× increase); the maximum increase occurs when the number of qubits increases from 14 to 15 (a 7.3× increase). As shown previously, these increases correspond to changes in the size of the matrices and, as a result, changes in the computational complexity.

To further understand the execution times of our CPU+GPU hybrid algorithm, Table 4 shows the percentage of execution times per kernel

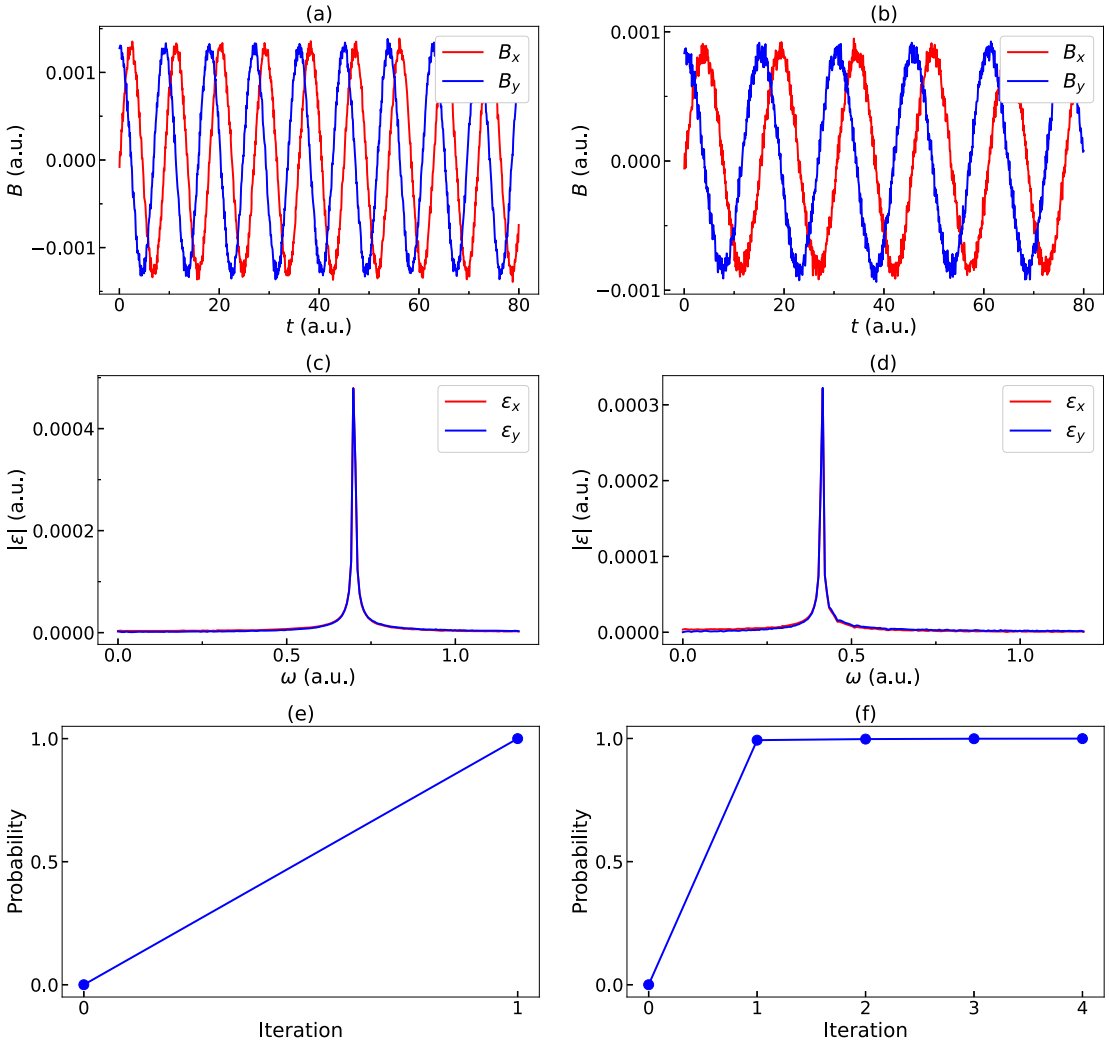

Fig. 3. Comparison of quantum optimal control calculations executed with the VAN-DAMME code for (a, c, e) 6-qubit and (b, d, f) 15-qubit systems with full coupling for preparing the W state. (a, b) Optimal control pulses; (c, d) Power spectra; (e, f) Convergence of transition probability vs. iteration.

for 10, 12, and 14 qubits for one iteration in Algorithm 1. As shown in this table, the LU factorization takes more than 50% of the execution time for all the cases. Moreover, two steps in the computation of the matrix exponentials, the LU factorization and the inversion of matrices (previous LU decomposition), take more than 65% of the overall execution time for all the cases. The overall computation of matrix exponentials requires additional matrix additions and matrix multiplications (see Table 3), and as a result, the computation of exponentials takes 80% or more of the overall execution time for all the cases. In short, the LU decomposition of matrices is the most time-consuming operation. Because the two fundamental operations in the LU decomposition are the swapping and subtraction of rows, and because all the rows of the target matrix cannot be stored in shared memory simultaneously, the performance of this kernel is limited by the GPU memory bandwidth, i.e., the LU decomposition is a memory-bound kernel. The second-most time-consuming kernel is the multiplication of matrices, whose performance is limited by the number of floating point operations the GPU executes per clock cycle, i.e., the multiplication of matrices is a compute-bound kernel.

A comparison of execution times indicates that our hybrid CPU+GPU implementation in the VAN-DAMME software package is 18.4 times faster (based on a geometric average) than the baseline CPU routines. The minimum gains are for the case of 10 qubits, which is 10.0 times faster. The maximum gains are for the simulation of 13 qubits, which is 31.2 times faster. The observed gains in performance are due to multiple

Table 4
Percentage of total execution time for each Kernel in the VAN-DAMME code.

Kernel	10 qubits	12 qubits	14 qubits
cuBLAS LU Factorization	54.9	51.0	55.4
cuBLAS Matrix Multiplication	22.7	22.8	23.5
cuBLAS LU Inversion	16.9	16.3	18.2
Inhouse Matrix Addition	3.8	6.0	2.2
Inhouse Matrix $\mathbf{A}[i]$ Calculation	0.7	1.2	0.4
Inhouse Matrix Max Operation	0.2	0.4	0.1
Other Kernels	0.8	2.3	0.2

factors. First, in Algorithm 1, the computation of the matrix exponentials is the most time-consuming operation, and to achieve the best performance, our routine uses Padé approximants, which are known to be fast and accurate. To make the computation robust, we numerically scaled down/up the input and output of the matrix exponentials. Moreover, to exploit the parallelism present in our algorithm, the VAN-DAMME code computed hundreds of matrix exponentials simultaneously, as shown in Routine 3. In addition, each line of code in Routine 3 has been parallelized, and high-performance kernels have been implemented or borrowed from existing libraries. In our algorithm, shown in Table 4, the LU decomposition of matrices and the multiplication of matrices are the most critical arithmetic operations. To achieve competitive performance, the VAN-DAMME code uses the batched routines in the cuBLAS library. From the CUDA documentation, the batched LU decomposi-

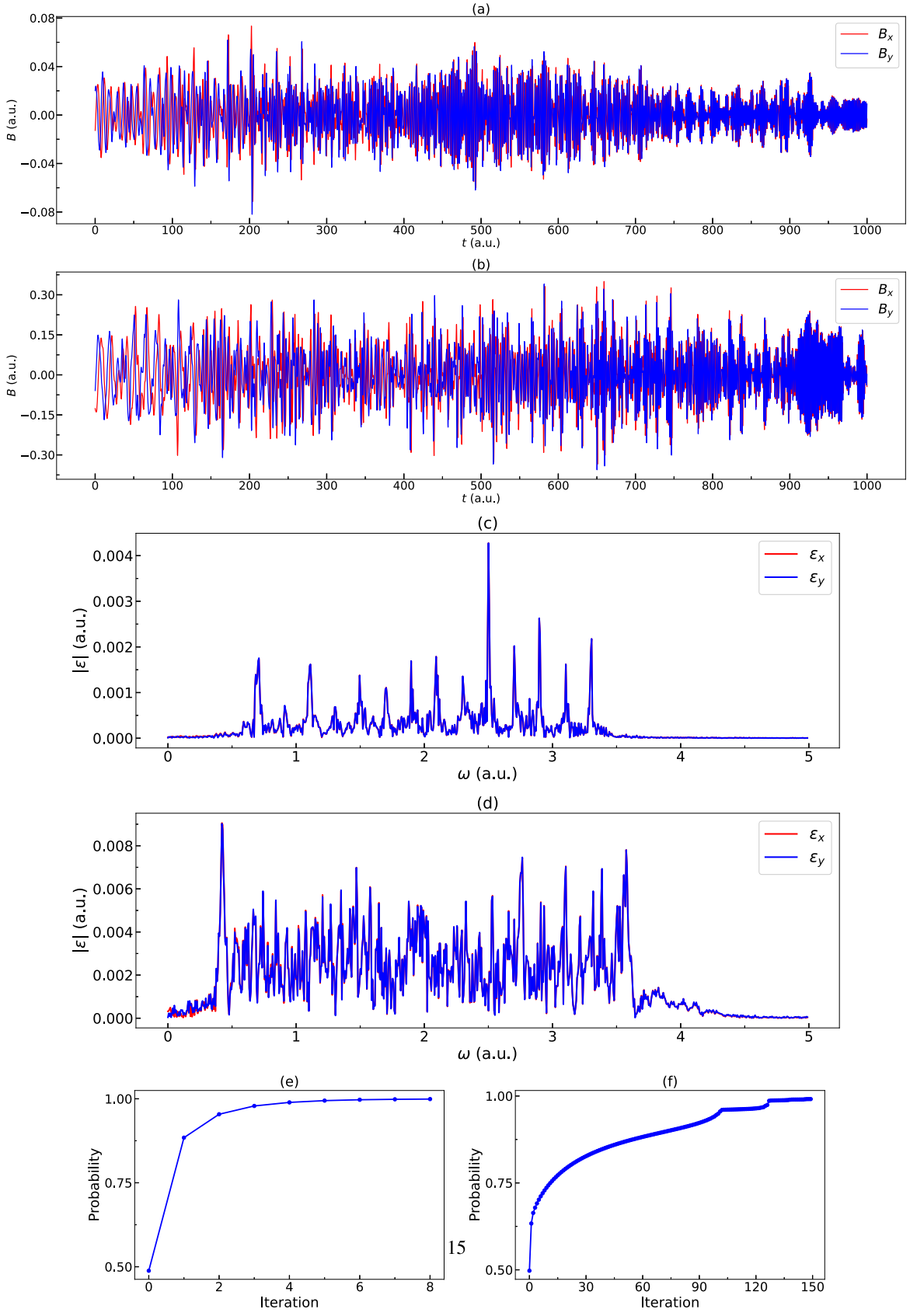

Fig. 4. Comparison of quantum optimal control calculations executed with the VAN-DAMME code for (a, c, e) 6-qubit and (b, d, f) 13-qubit systems with full coupling for preparing the GHZ state. (a, b) Optimal control pulses; (c, d) Power spectra; (e, f) Convergence of transition probability vs. iteration.

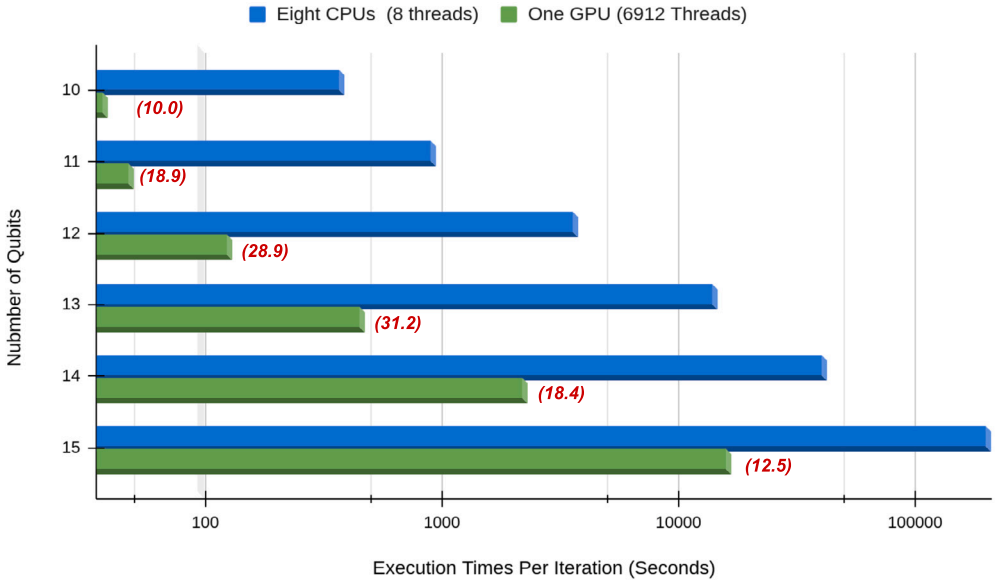

Fig. 5. Execution times (log scale) of the VAN-DAMME code on CPUs and GPUs for quantum control calculations on 10 - 15 qubits. The number in parenthesis is the ratio of the execution time on CPUs to that on GPUs.

tion of matrices has been tailored to decompose small-sized matrices and take advantage of both the GPU memory bandwidth and the GPU floating-point throughput. Other batched operations in the cuBLAS library, including the batched matrix inversions (previous LU decomposition) and batched matrix multiplications, have been tailored to work on small-sized matrices, which are used in the VAN-DAMME software package. Also, our kernels, including those for matrix initializations and additions, have been designed to work in batches as shown in Fig. 2. As shown in Table 4, our kernels only use about 8.0% of the overall execution time. In contrast, the CPU library routines are not optimized for batched operations and, therefore, do not perform well for the quantum control calculations in this work.

Fig. 5 shows that as the size of the matrices increases, the comparative gains in performance of our hybrid CPU+GPU approach decrease. Table 4 shows that the most time-consuming task for all qubit configurations is the LU decomposition. As described above, the LU decomposition is a memory-bound kernel. For large matrices, the cost of moving data between the GPU main memory and the GPU shared memory, or the GPU registers, affects the overall performance. A similar reasoning applies to LU inversions, which is the third most time-consuming task. Despite this behavior, our projections indicate that our hybrid CPU+GPU implementation in the VAN-DAMME software package should be about 8.0 times faster for 16 qubits.

6. Conclusions

In conclusion, we have developed and provided the open-source VAN-DAMME software package for accelerating QOC calculations of multi-qubit systems with advanced GPU parallelization approaches. To enable additional computation performance, the VAN-DAMME code also leverages symmetry-based techniques that can decompose the multi-qubit Hilbert space to block diagonalize the Hamiltonians used in the QOC calculations. This reduction uses the first block of the transformed Hamiltonians in QOC calculations and limits the state transition to a symmetry-protected subspace to suppress quantum errors. To understand the computational bottlenecks in the VAN-DAMME code, we carried out several extensive tests on data layout, computational complexity, memory requirements, and performance. These extensive analyses allowed us to develop computationally efficient approaches to compute matrix exponentials on GPUs since this is the most time-consuming operation in the QOC algorithm. To enable these performance gains, the VAN-DAMME software package uses a custom GPU routine that computes the exponentials of hundreds of small matrices simultaneously using Padé approximants. To ensure the numerical robustness of our calculations, we leveraged various properties of the matrix exponential, including scaling down the argument of the matrix exponential and subsequent scaling up of the resulting matrices after the matrix exponential is computed.

All of the operations in the VAN-DAMME code are executed in batches to maximize computational efficiency, which include scaling of the arguments in the matrix exponentials, computation of Padé factors, computation of matrix·matrix and matrix·vector products, addition of matrices, and other matrix and vector operations. To validate the accuracy of our implementation, we applied the GPU-accelerated VAN-DAMME code to a variety of multi-qubit QOC calculations on conventional CPUs and benchmarked its performance on state-of-the-art A100 GPUs. These computational timing tests demonstrated that the GPU-accelerated VAN-DAMME code generates the same results (i.e., optimal pulses, power spectra, gradients, and convergence properties) as the benchmark calculations on CPUs but with a speedup that is 18.4× faster. Our GPU-accelerated approach allows efficient calculations of multi-qubit systems containing up to 15 qubits and beyond, and the custom parallelization techniques in the VAN-DAMME code can be used for the efficient implementation of QOC applications across multiple domains.

CRediT authorship contribution statement

José M. Rodríguez-Borbón: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Xian Wang:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Adrián P. Diéguez:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Data curation. **Khaled Z. Ibrahim:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation. **Bryan M. Wong:** Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through the Advanced Computing (SciDAC) program under Award Number DE-SC0022209. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award BES-ERCAP0023692.

Data availability

The VAN-DAMME program is available via the Computer Physics Communications library link (https://doi.org/10.17632/zcgw2n5bjf.1).

References

[1] J.M. Gambetta, J.M. Chow, M. Steffen, Building logical qubits in a superconducting quantum computing system, npj Quantum Inf. 3 (1) (2017) 2.
[2] F. Arute, K. Arya, R. Babbush, D. Bacon, J.C. Bardin, R. Barends, R. Biswas, S. Boixo, F.G. Brandao, D.A. Buell, et al., Quantum supremacy using a programmable superconducting processor, Nature 574 (7779) (2019) 505–510.
[3] V.E. Manucharyan, J. Koch, L.I. Glazman, M.H. Devoret, Fluxonium: single Cooper-pair circuit free of charge offsets, Science 326 (5949) (2009) 113–116.
[4] L.B. Nguyen, Y.-H. Lin, A. Somoroff, R. Mencia, N. Grabon, V.E. Manucharyan, High-coherence fluxonium qubit, Phys. Rev. X 9 (4) (2019) 041041.
[5] J.I. Cirac, P. Zoller, Quantum computations with cold trapped ions, Phys. Rev. Lett. 74 (20) (1995) 4091.
[6] T. Monz, P. Schindler, J.T. Barreiro, M. Chwalla, D. Nigg, W.A. Coish, M. Harlander, W. Hänsel, M. Hennrich, R. Blatt, 14-qubit entanglement: creation and coherence, Phys. Rev. Lett. 106 (13) (2011) 130506.
[7] S. Ebadi, T.T. Wang, H. Levine, A. Keesling, G. Semeghini, A. Omran, D. Bluvstein, R. Samajdar, H. Pichler, W.W. Ho, et al., Quantum phases of matter on a 256-atom programmable quantum simulator, Nature 595 (7866) (2021) 227–232.
[8] N. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbrüggen, S.J. Glaser, Optimal control of coupled spin dynamics: design of NMR pulse sequences by gradient ascent algorithms, J. Magn. Res. 172 (2) (2005) 296–305.
[9] T. Caneva, T. Calarco, S. Montangero, Chopped random-basis quantum optimization, Phys. Rev. A 84 (2) (2011) 022326.
[10] V.F. Krotov, I. Feldman, An iterative method for solving optimal-control problems, Eng. Cybern. 21 (2) (1983) 123–130.
[11] A. Raza, C. Hong, X. Wang, A. Kumar, C.R. Shelton, B.M. Wong, NIC-CAGE: an open-source software package for predicting optimal control fields in photo-excited chemical systems, Comput. Phys. Commun. 258 (2021) 107541.
[12] X. Wang, P. Kairys, S.H.K. Narayanan, J. Hückelheim, P. Hovland, Memory-efficient differentiable programming for quantum optimal control of discrete lattices, in: 2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS), IEEE, 2022, pp. 94–99.
[13] J.M. Rodríguez-Borbón, X. Wang, A.P. Diéguez, K.Z. Ibrahim, B.M. Wong, TRA-VOLTA: GPU acceleration and algorithmic improvements for constructing quantum optimal control fields in photo-excited systems, Comput. Phys. Commun. 296 (2024) 109017.
[14] X. Wang, M.S. Okyay, A. Kumar, B.M. Wong, Accelerating quantum optimal control of multi-qubit systems with symmetry-based Hamiltonian transformations, AVS Quantum Sci. 5 (4) (2023).
[15] D. Lu, K. Li, J. Li, H. Katiyar, A.J. Park, G. Feng, T. Xin, H. Li, G. Long, A. Brodutch, J. Baugh, B. Zeng, R. Laflamme, Enhancing quantum control by bootstrapping a quantum processor of 12 qubits, npj Quantum Inf. 3 (1) (2017) 45.
[16] N. Leung, M. Abdelhafez, J. Koch, D. Schuster, Speedup for quantum optimal control from automatic differentiation based on graphics processing units, Phys. Rev. A 95 (4) (2017) 042318.
[17] S.H.K. Narayanan, T. Propson, M. Bongarti, J. Hückelheim, P. Hovland, Reducing memory requirements of quantum optimal control, in: International Conference on Computational Science, Springer International Publishing, Cham, 2022, pp. 129–142.
[18] D. Kirk, NVIDIA CUDA software and GPU parallel computing architecture, in: Proceedings of the 6th International Symposium on Memory Management, ISMM '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 103–104.
[19] NVIDIA Incorporated, CUDA toolkit documentation, https://developer.nvidia.com/cuda-toolkit, 2023. (Accessed 19 October 2024).
[20] C. Moler, C. Van Loan, Nineteen dubious ways to compute the exponential of a matrix, SIAM Rev. 20 (4) (1978) 801–836.
[21] C. Moler, C. Van Loan, Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later, SIAM Rev. 45 (1) (2003) 3–49.
[22] G.H. Golub, C.F. Van Loan, Matrix Computations, 4th edition, The Johns Hopkins University Press, 2715 North Charles Street, Baltimore, MD, 21218, USA, 2013.
[23] N.J. Higham, The scaling and squaring method for the matrix exponential revisited, SIAM J. Matrix Anal. Appl. 26 (4) (2005) 1179–1193.
[24] N.E.R.S.C. NERSC, NERSC technical documentation, https://docs.nersc.gov, 2022. (Accessed 19 October 2024).
[25] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, J.S. Vetter, The scalable heterogeneous computing (SHOC) benchmark suite, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 63–74.
[26] A. Cabello, Bell's theorem with and without inequalities for the three-qubit Greenberger-Horne-Zeilinger and W states, Phys. Rev. A 65 (3) (2002) 032108.
[27] D.M. Greenberger, M.A. Horne, A. Zeilinger, Going beyond Bell's theorem, in: Bell's Theorem, Quantum Theory and Conceptions of the Universe, Springer, 1989, pp. 69–72.
[28] M. Fleischhauer, M.D. Lukin, Quantum memory for photons: dark-state polaritons, Phys. Rev. A 65 (2) (2002) 022314.
[29] M. Hillery, V. Bužek, A. Berthiaume, Quantum secret sharing, Phys. Rev. A 59 (3) (1999) 1829.
[30] J.J. Bollinger, W.M. Itano, D.J. Wineland, D.J. Heinzen, Optimal frequency measurements with maximally correlated states, Phys. Rev. A 54 (6) (1996) R4649.
[31] V. Giovannetti, S. Lloyd, L. Maccone, Quantum-enhanced measurements: beating the standard quantum limit, Science 306 (5700) (2004) 1330–1336.
[32] V. Giovannetti, S. Lloyd, L. Maccone, Quantum metrology, Phys. Rev. Lett. 96 (1) (2006) 010401.
[33] V. Volkov, J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE, 2008, pp. 1–11.
[34] A. Haidar, T. Dong, P. Luszczek, S. Tomov, J. Dongarra, Optimization for performance and energy for batched matrix computations on GPUs, in: Proceedings of the 8th Workshop on General Purpose Processing Using GPUs, 2015, pp. 59–69.
[35] A. Haidar, T. Dong, P. Luszczek, S. Tomov, J. Dongarra, Batched matrix computations on hardware accelerators based on GPUs, Int. J. High Perform. Comput. Appl. 29 (2) (2015) 193–208.
[36] J. Dongarra, S. Hammarling, N.J. Higham, S.D. Relton, P. Valero-Lara, M. Zounon, The design and performance of batched blas on modern high-performance computing systems, Proc. Comput. Sci. 108 (2017) 495–504.
[37] L.N. Trefethen, D. Bau III, Numerical Linear Algebra, 1st edition, 1997, Siam, 3600 Market Street, 6th Floor, Philadelphia, PA, 19104, USA.
[38] M. Harris, et al., Optimizing parallel reductions in CUDA, Nvidia Dev. Technol. 2 (4) (2007) 70.
[39] AMD Incorporated, AMD EPYC-7763 processor, https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7763.html, 2020. (Accessed 19 October 2024).
[40] NVIDIA Incorporated, NVIDIA a100 tensor core GPU, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020. (Accessed 19 October 2024).
[41] N.E.R.S.C. NERSC, The Cray BLAS libraries, https://docs.nersc.gov/development/libraries/libsci, 2022. (Accessed 19 October 2024).