Computer Physics Communications 296 (2024) 109017

Contents lists available at ScienceDirect

Computer Physics Communications

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

Computer Programs in Physics

TRAVOLTA: GPU acceleration and algorithmic improvements for constructing quantum optimal control fields in photo-excited systems ☆,☆☆

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

[a] Materials Science & Engineering Program, University of California-Riverside, 900 University Avenue, Riverside, 92521, CA, USA
[b] Department of Physics & Astronomy, University of California-Riverside, 900 University Avenue, Riverside, 92521, CA, USA
[c] Applied Mathematics & Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, 94720, CA, USA
[d] Materials Science & Engineering Program, Department of Chemistry, and Department of Physics & Astronomy, University of California-Riverside, 900 University Avenue, Riverside, 92521, CA, USA

ARTICLE INFO

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

ABSTRACT

We present an open-source software package, TRAVOLTA (Terrific Refinements to Accelerate, Validate, and Optimize Large Time-dependent Algorithms), for carrying out massively parallelized quantum optimal control calculations on GPUs. The TRAVOLTA software package is a significant overhaul of our previous NIC-CAGE algorithm and also includes algorithmic improvements to the gradient ascent procedure to enable faster convergence. We examine three different variants of GPU parallelization to assess their performance in constructing optimal control fields in a variety of quantum systems. In addition, we provide several examples with extensive benchmarks of our GPU-enhanced TRAVOLTA code to show that it generates the same results as previous CPU-based algorithms but with a speedup that is more than ten times faster. Our GPU enhancements and algorithmic improvements enable large quantum optimal control calculations that can be efficiently and routinely executed on modern multi-core computational hardware.

Program summary
Program Title: TRAVOLTA
CPC Library link to program files: https://doi.org/10.17632/grwppm37rn.1
Licensing provisions: GNU General Public License 3
Programming language: C++, openBLAS, and CUDA
Supplementary material: Brief review of LU decomposition, raw numerical values used to generate Fig. 6 in the main text, and input examples for the TRAVOLTA software package.
Nature of problem: The TRAVOLTA software package utilizes GPU accelerated routines and new algorithmic improvements to compute optimized electric fields that can drive a system from a known initial vibrational eigenstate to a specified final quantum state with a large (≈ 1) transition probability.
Solution method: Quantum control, GPU acceleration, analytic gradients, Crank-Nicolson propagation, and gradient ascent optimization.

1. Introduction

The implementation and use of quantum optimal control (QOC) approaches continue to attract significant interest due to recent time-resolved advances in photocatalysis [1], photo-excited systems [2,3], and quantum gate operations [4–9]. In short, the ultimate goal of QOC is to construct optimal control pulses that drive a quantum system from an initial to a desired target state. In contrast to initial value problems

☆ The review of this paper was arranged by Prof. Jimena Gorfinkiel.
☆☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

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

https://doi.org/10.1016/j.cpc.2023.109017
Received 9 July 2023; Received in revised form 8 November 2023; Accepted 13 November 2023
Available online 22 November 2023

solved by propagating the time-dependent Schrödinger equation forward in time, QOC focuses on the *inverse* problem to construct control pulses that enable desired transitions [10–15]. As related examples, the GRAPE [16], CRAB [17], and Krotov [18] computational approaches were developed to solve QOC problems in small spin-1/2 systems. The most computationally expensive part in all these QOC algorithms is evaluating the exponential of several large matrices at each time step. To address this problem for chemical/material systems (as opposed to small spin-1/2 systems), we previously developed the NIC-CAGE software package [19], which utilizes more efficient linear propagators and gradients within a Crank-Nicolson scheme.

Despite the efficiency of the linear propagators in the NIC-CAGE code, several other computational bottlenecks in its QOC algorithms could be further improved. For example, in previous studies, we discovered that the gradient ascent algorithm in NIC-CAGE causes the first iteration to have an extremely small gradient, resulting in slow convergence [20,21]. In addition, several intensive mathematical operations, such as inverses of complex-banded matrices, matrix·matrix multiplications, and matrix·vector multiplications (among others) are executed numerous times and can be time-consuming. To address both of these bottlenecks, we present a new open-source software package, TRAVOLTA (Terrific Refinements to Accelerate, Validate, and Optimize Large Time-dependent Algorithms), which uses new algorithmic improvements to the gradient and custom massively-parallelized GPU acceleration schemes to improve computational performance.

The TRAVOLTA code is a completely rewritten code in the high-performance C++ and CUDA parallel programming languages to enable efficient and large QOC calculations on modern multi-core GPUs. In addition to algorithmic improvements to the gradient ascent algorithm to improve convergence, we also developed three customized high-performance kernels to assess their computational efficiency. We executed each of these customized kernels on state-of-the-art A100 GPUs on the *Perlmutter* supercomputer at the National Energy Research Scientific Computing Center (NERSC) to test their accuracy against previous benchmark QOC calculations. We provide computational timings of the TRAVOLTA code as a function of system size (with examples of input/output parameters in the Supplemental Material used to run the code), which show that our GPU-based batch kernel algorithm is more than ten times faster than the corresponding CPU implementation (with computational performance that actually increases with system size). Finally, we conclude with a discussion and perspective look at potential applications of our algorithmic improvements and GPU parallelization techniques for QOC calculations of other quantum systems.

2. Theory and computational methodology

In previous work, we developed the NIC-CAGE software package to successfully construct optimal control fields for a variety of photo-excited chemical systems [19]. To understand the new GPU and algorithmic enhancements in the TRAVOLTA code developed in this work, we briefly summarize the original NIC-CAGE algorithms in this section. The temporal dynamics of nuclei in a molecular system are governed by the time-dependent Schrödinger equation, which, in atomic units is given by

$$i\frac{\partial}{\partial t}\psi(x,t) = \mathcal{H}(x,t)\psi(x,t), \tag{1}$$

where the time-dependent Hamiltonian $\mathcal{H}(x,t)$ is

$$\mathcal{H}(x,t) = -\frac{1}{2m}\frac{\partial^2}{\partial x^2} + V(x) - \mu(x)\epsilon(t). \tag{2}$$

In the expression above, x is the reduced coordinate along a reaction path, m is the effective mass associated with the molecular motion, $V(x)$ is the Born–Oppenheimer electronic energy of the molecule along the reaction path, $\mu(x)$ is the dipole moment function, and $\epsilon(t)$ is the time-dependent external electric field whose temporal form is iteratively optimized using the QOC algorithms in this work.

The Hamiltonian $\mathcal{H}(x,t)$ can be discretized across a grid of L equidistant points with separation Δx, resulting in the following matrix equation

$$\mathbf{H} = -\frac{1}{24m(\Delta x)^2}\left(-\mathbf{I}^{(2-)} + 16\mathbf{I}^{(1-)} - 30\mathbf{I} + 16\mathbf{I}^{(1+)} - \mathbf{I}^{(2+)}\right) + \mathbf{V}, \tag{3}$$

where $\mathbf{I}$ is an $L\times L$ identity matrix with entries of 1 on the main diagonal, $\mathbf{I}^{(1\pm)}$ are $L\times L$ matrices with entries of 1 on the 1st diagonal above (1+) / below (1–) the main diagonal, $\mathbf{I}^{(2\pm)}$ are $L\times L$ matrices with entries of 1 on the 2nd diagonal above (2+) / below (2–) the main diagonal, and $\mathbf{V}$ is an $L\times L$ diagonal matrix with entries $[\mathbf{V}]_{ij} = V(x_i)\delta_{ij}$, where x_i is the value of x at the ith grid point, and δ_{ij} is the Kronecker delta. From these definitions, $\mathbf{H}$ is a pentadiagonal matrix. Using the Crank-Nicolson scheme, the time-evolution of the quantum system is given by

$$\left(\mathbf{I} + \frac{i\tau}{2}\mathbf{H}_{j+1/2}\right)\boldsymbol{\psi}_{j+1} = \left(\mathbf{I} - \frac{i\tau}{2}\mathbf{H}_{j+1/2}\right)\boldsymbol{\psi}_j, \tag{4}$$

where $\boldsymbol{\psi}_j$ is vectorized in space ($\mathbf{x}$) and evaluated at time $t_j = j\tau$, where $j = 0,\ldots,N-1$ and $\tau = \frac{T}{N-1}$ is the time step across a grid of N equidistant points on the interval $[0,T]$. That is, $\boldsymbol{\psi}_j$ is a column vector, and $\mathbf{H}_{j+1/2}$ is evaluated at time $t_{j+1/2} = (j+1/2)\tau$, where $j = 0,\ldots,N-2$. For compactness of notation, we define $\mathbf{U}_{j+1/2} = \left(\mathbf{I} + \frac{i\tau}{2}\mathbf{H}_{j+1/2}\right)$ and $\mathbf{W}_{j+1/2} = \left(\mathbf{I} - \frac{i\tau}{2}\mathbf{H}_{j+1/2}\right)$.

The original NIC-CAGE software package uses an iterative gradient-ascent algorithm that maximizes the transition probability, P given by

$$P\left[\psi_{N-1}(x)\right] = \left|\int_{-\infty}^{\infty}\psi_f^*(x)\psi_{N-1}(x)dx\right|^2, \tag{5}$$

where ψ_f is a known desired target wavefunction (given by the user), and ψ_{N-1} is the propagated wavefunction at the last time step (after applying $N-1$ successive propagation steps of Eq. (4)). To prevent unphysically large values of the electric field, we define the loss function as

$$J\left[\psi_{N-1}(x),\epsilon\right] = P\left[\psi_{N-1}(x)\right] - \alpha\int_0^T \epsilon^2(t)\mathrm{d}t, \tag{6}$$

where α is an empirical penalty factor given by the user. The NIC-CAGE software package calculates analytic gradients of $J\left[\psi_{N-1}(x),\epsilon(t)\right]$ with respect to $\epsilon(t)$ (i.e., $\frac{\mathrm{d}J\left[\psi_{N-1}(x),\epsilon\right]}{\mathrm{d}\epsilon_{j+1/2}}$) at all time steps using the chain rule (see Ref. [19] for further details). The optimized, time-dependent external electric field at the lth iteration step, $\epsilon_{j+1/2}^{(l)}$, is then numerically computed using the expression

$$\epsilon_{j+1/2}^{(l)} = \epsilon_{j+1/2}^{(l-1)} + \gamma\frac{\mathrm{d}J\left[\psi_{N-1}(x),\epsilon^{(l-1)}\right]}{\mathrm{d}\epsilon_{j+1/2}^{(l-1)}}, \tag{7}$$

where γ is the learning rate of the gradient ascent algorithm, which is calculated using a bisection line-search algorithm. This process iterates until the probability, P, exceeds some predetermined threshold. Our sequential NIC-CAGE algorithm is summarized below.

In the NIC-CAGE software package, the propagation of the wavefunction in line 7 requires solving a large number of sequential linear equations of the form $\mathbf{U}_{j+1/2}\boldsymbol{\psi}_{j+1} = \mathbf{W}_{j+1/2}\boldsymbol{\psi}_j$ given by Eq. (4). Specifically, the number of linear equations increases with the number of time steps N, whereas the size of the pentadiagonal matrices $\mathbf{U}_{j+1/2}$ and $\mathbf{W}_{j+1/2}$ increases with the number of points in the spatial grid, L. The execution of line 8 requires operations such as multiplication of vectors by scalars, dot products, and the calculation of vector norms. The calculation of the gradient in line 10 requires the inversion of pentadiagonal matrices, matrix·matrix multiplications, and matrix·vector multiplications, as well as other operations. In this step, the inversion

Algorithm 1: Original NIC-CAGE Algorithm.

Input: Spatial interval $[x_{\min}, x_{\max}]$, grid spacing Δx, time interval $[0, T]$, time step τ, mass m, dipole moment function $\mu(x)$, potential energy function $V(x)$, initial state number i, desired final state number f, threshold probability δ, and maximum number of iterations Max.
Output: Initial wavefunction $\psi_i(x)$, desired final wavefunction $\psi_f(x)$, final propagated wavefunction $\psi_{N-1}(x)$, optimized electric field $\epsilon(t)$, and power spectrum of optimized electric field.

1 /* Working with zero-based indexing */
2 Diagonalize time-independent Schrödinger Equation in Eq. (1) to obtain $\psi_i(x)$ and $\psi_f(x)$.
3 $\epsilon_{j+1/2} = 0$ for $j = 0, ..., N-2$
4 $P = 0$; $Iter = 0$; $\psi_0(x) = \psi_i(x)$
5 **while** $Iter < Max$ ***and*** $P < \delta$ **do**
6 **for** $j = 0$ ***to*** $N-2$ **do**
7 Calculate $\psi_{j+1}(x)$ from Eq. (4)
  end
8 Update J and P using Eqs. (5) and (6)
9 **for** $j = N-2$ ***to*** 0 **do**
10 Calculate $\frac{dJ(\psi_{N-1}(x),\epsilon)}{d\epsilon_{j+1/2}}$ with chain rule
  end
11 Calculate γ using bisection line-search method
12 Update vector $\epsilon_{j+1/2}$ using Eq. (7)
13 $Iter = Iter + 1$
  end
14 return $\epsilon_{j+1/2}$

of the pentadiagonal matrices can be executed in parallel. Once these inverses are found, the remaining operations have to be executed sequentially. The line-search method executed in line 11 calls the function in line 7 several times to search for the optimal update rate, γ.

Because several operations in Algorithm 1 are time-consuming, we offloaded all of the operations in line 10 to the GPU (discussed further in Section 4), which allows for the parallel construction of pentadiagonal matrices as well as the parallel computation of matrix inverses. In addition, due to the high performance of GPUs for computing numerically intensive operations [22,23], we offloaded all sequential operations involving matrices, including matrix initializations, matrix·matrix/matrix·vector multiplications, and other linear algebra operations.

3. Amplified gradient modification

We briefly introduce the line-search method [24,25] for the ground to first-excited state ($v_i = 0 \rightarrow v_f = 1$) transition in a Morse potential (which mimics the photo-induced stretching of an O–H bond [26]) to illustrate our amplified gradient modification. The Morse potential in this work has the functional form

$$V(x) = 0.1994[\exp(-1.189(x - 1.821)) - 1]^2 - 0.1994. \tag{8}$$

The objective of the line-search method is to calculate the update rate, γ, that minimizes the loss function $-J(\gamma)$, given the gradient $\frac{dJ}{d\epsilon(t)}$ in each iteration. Our analysis is based on the assumption that $-J(\gamma)$ has a minimum at some value of γ larger than zero and is convex near its minimum. The line-search procedure is accomplished in two phases. First, the algorithm evaluates $-J(\gamma^{(j)})$ at an increasing sequence of $\gamma^{(j)}, j = 0, 1, 2, \ldots$ (shown as blue dots in Fig. 1a) which starts at $\gamma^{(0)} = 0$ and has the recurrence relation

$$\gamma^{(j+1)} = \left(\gamma^{(j)} + 0.3\right) \times 1.4, \tag{9}$$

where 0.3 and 1.4 are empirical coefficients. The loop of the evaluation breaks when $-J(\gamma^{(n)}) > -J(\gamma^{(n-1)})$ is satisfied for some integer n so that the minimum of $-J(\gamma)$ in the interval $[0, \gamma^{(n)}]$. To avoid an unrealistically large value of $\gamma^{(n)}$, it is common practice to set a threshold value, γ_{thres}, for the upper bound of the interval. If $-J(\gamma^{(n)}) > -J(\gamma^{(n-1)})$ is not achieved within $[0, \gamma_{\text{thres}}]$, the recurrence in Eq. (9) is forced to break. The second step is to search for the approximate value of that minimum in the interval $[0, \gamma^{(n)}]$ with the bisection line-search algorithm. Since the function $-J(\gamma)$ is convex near the minimum, the gradient at any point to the left of the minimum must be smaller than 0, and the gradient at any point to the right is larger than 0. The bisection line-search method evaluates the sign of the gradient $-\frac{dJ(\gamma)}{d\gamma}$ (note that this is different from the gradient $\frac{dJ}{d\epsilon(t)}$) at the midpoint, $\frac{\gamma^{(n)}}{2}$, of the interval $[0, \gamma^{(n)}]$. If $-\frac{dJ(\gamma^{(n)}/2)}{d\gamma} > 0$, the minimum is in the left half of the interval given by $[0, \frac{\gamma^{(n)}}{2}]$; otherwise, it is in the right half (i.e., $[\frac{\gamma^{(n)}}{2}, \gamma^{(n)}]$). We retain the half containing the minimum only and recursively evaluate the gradient at the midpoint of the new interval and halve the interval again until the length of the interval is smaller than a threshold value. The midpoint of the final interval is then taken as the optimal γ, and the bisection line-search procedure is terminated. Fig. 1b shows the points (red dots) at which the gradient $-\frac{dJ(\gamma)}{d\gamma}$ is evaluated. The solid arrow indicates the optimal γ that the bisection line search eventually outputs.

It is worth noting that the optimal update rate γ is extremely large (on the order of 10^{12}), which we further explain below. The transition probability, P (which ranges from 0 to 1 by definition), is typically a smooth functional of the control field $\epsilon(t)$. As such, the gradient $\frac{dP}{d\epsilon(t)}$ is zero when P is at its minimum of 0; in addition, $\frac{dJ}{d\epsilon(t)} \approx \frac{dP}{d\epsilon(t)}$ has a very small norm when $P = 0$ because the penalty factor α is typically set to a small value. We found that these small gradient issues primarily occur in the first iteration since $\epsilon(t)$ is initialized as a zero vector or with small amplitude white noise in the NIC-CAGE algorithm, which makes P nearly 0. As a result, γ needs to be very large to make any substantial update to $\epsilon(t)$. This forces the $\gamma^{(j)}$ defined in Eq. (9) to be a long sequence, and $-J(\gamma^{(j)})$ has to be evaluated by the forward propagation in line 7 of Algorithm 1 many times, which is extremely time-consuming. Another downside is that the scales of the x-axis ($\sim 8.28 \times 10^{12}$) and y-axis (~ 0.87) are extremely not comparable, and the gradient $-\frac{dJ(\gamma)}{d\gamma}$ everywhere is very close to 0. This causes a floating point underflow error in determining the sign of the gradient, which can cause the algorithm to retain the wrong half of the interval for γ. As shown in Fig. 1b, the conventional bisection line-search algorithm eventually outputs an incorrect value of γ ($\sim 8.28 \times 10^{12}$, indicated by the solid arrow) instead of the correct value of $\sim 5.73 \times 10^{12}$ (indicated by the hollow arrow).

To address this small gradient problem, we multiply the gradient $\frac{dJ}{d\epsilon(t)}$ by an empirical coefficient β to amplify its norm. The update rate γ' in this amplified gradient modification satisfies $\gamma'\beta \approx \gamma$, where γ is the update rate in the conventional method. Therefore γ' can be small when the amplified gradient coefficient β is set to a sufficiently large value. One improvement, as we see in Fig. 1c, is that the amplified gradient modification evaluates $-J(\gamma)$ (which requires calling the forward propagation in line 7 of Algorithm 1) at only 11 points, while the conventional method requires 90 evaluations (blue dots in Figs. 1a, c). In addition to accelerating the first phase of the bisection line-search method, the amplified gradient modification also fixes the floating point underflow error in the second phase since the scales of the x-axis (~ 18.36) and y-axis (~ 0.87) are now comparable, as shown in Fig. 1d. In conclusion, the modified algorithm now outputs the correct optimal γ' (indicated by the solid red arrow in Fig. 1d) in significantly less execution time.

As discussed above, extremely large values of γ occur when the probability P is very small due to the small value of the gradient $\frac{dJ}{d\epsilon(t)}$ when P is at its minimum of zero. When $P > 0.1$, the optimal γ is typically less than 0.1, and the amplified gradient modification is no longer necessary. As such, we seek to define the empirical coefficient β so that the gradient $\frac{dJ}{d\epsilon(t)}$ is amplified only when $P < 0.1$, and β should be negatively correlated to P. We found the following definition

$$\begin{aligned} \beta &= \frac{0.1}{\sqrt{P}} \quad \text{if} \quad P < 0.1, \\ &= 1 \qquad \text{if} \quad P \geq 0.1, \end{aligned} \tag{10}$$

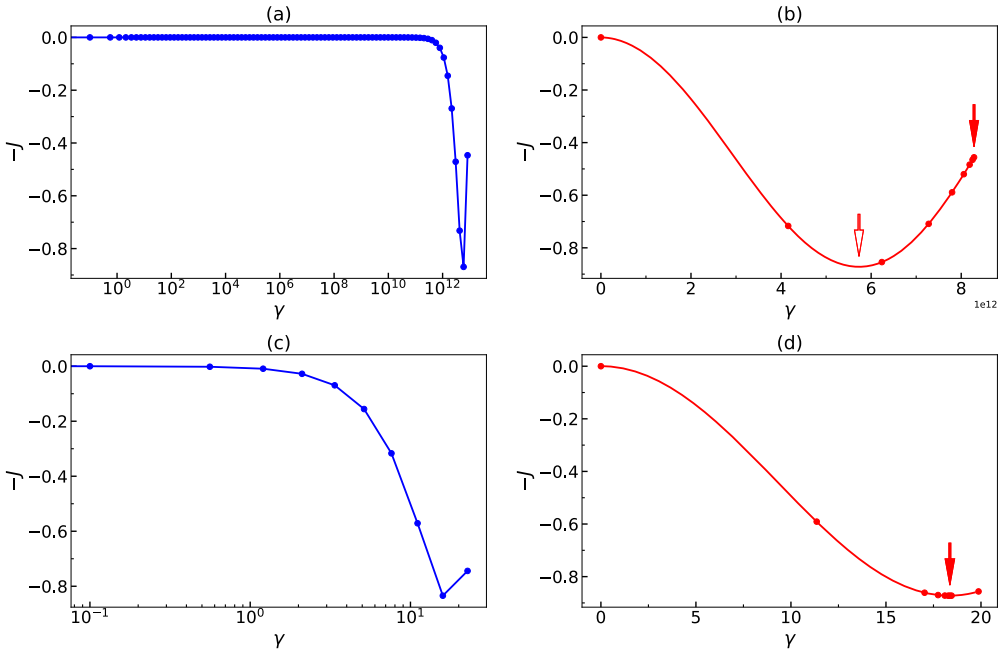

Fig. 1. Comparison of conventional and amplified gradient methods for the bisection line search in the first QOC iteration for the $\nu_i = 0 \rightarrow \nu_f = 1$ transition in the Morse potential. (a)/(c) Phase one: evaluating $-J(\gamma)$ with the (a) conventional and (c) amplified gradient method, respectively. The algorithm evaluates $-J(\gamma)$ at different points defined by Eq. (9) (blue dots) until a minimum occurs within the range of γ. (b)/(d) Phase two: searching for the minimum of the function $-J(\gamma)$ with the (b) conventional and (d) amplified gradient method, respectively. The algorithm evaluates the sign of the gradient $-\frac{dJ(\gamma)}{d\gamma}$ at different points (red dots) determined by the bisection line-search algorithm. The solid arrow indicates the optimal γ that the bisection line-search eventually outputs. Note that the line search for the conventional gradient method outputs a wrong value of γ shown in panel (b), which is not the true minimum (indicated by the hollow arrow).

satisfies these requirements for routine QOC calculations in the TRAVOLTA software package. We compare the results of the original gradient and the amplified gradient modification in Section 5.1.

4. GPU acceleration

In line 10 of Algorithm 1, we need to compute the inverse of millions of small complex banded matrices. In this section, we describe a GPU-based method to compute these inverses efficiently. Our method is based on the LU decomposition [27–29] of a matrix $\mathbf{A}$ where $\mathbf{A} = \mathbf{LU}$. A review of the canonical LU decomposition method and its implementation on CPUs is given in the Supplemental Material. Multiple approaches have been developed to execute LU decomposition including right-looking and left-looking LU factorization methods [27]. In this work, we chose to work with right-looking factorization methods since they are efficient in computing matrix inverses [30] and perform well on GPUs [31]. To increase computational performance, we do not use pivoting approaches since the $\mathbf{U}_{j+1/2}$ matrices are non-singular, and our calculations show that our algorithms are still accurate when Gaussian elimination without pivoting is applied to the $\mathbf{U}_{j+1/2}$ matrices (i.e., no underflow/overflow floating point errors occur).

4.1. Batched LU decomposition

Algorithm 1 allows for the computation of multiple matrix inverses in parallel. To parallelize this, we developed an efficient GPU kernel that takes multiple complex banded matrices as input, executes the LU decomposition, and finally computes the inverses. The general procedure for executing a large number of small problems in parallel with high-performance computing is known as Batched Computations [32,33]. In the context of our GPU-accelerated TRAVOLTA code, we first execute the LU decomposition of a large set of small complex-banded matrices and subsequently use them to efficiently compute matrix inverses. To achieve high performance on GPUs [34], our kernel utilizes the following strategies: (1) utilization of coalesced reads and writes, (2) efficient use of shared memory, (3) minimization of the number of thread divergences and synchronization barriers, (4) targeting high occupancy and/or increasing the instruction level parallelism per thread [35], and (5) minimization of the movement of data within the GPU and between the CPU and the GPU. Because the optimization of the movement of data inside the GPU is paramount (i.e., all reads and writes operations have to be coalesced), we address this problem first.

Fig. 2 presents our data layout for achieving coalesced reads for a set of matrices with band size b. Panel (a) shows banded matrices $\mathbf{A}_i$, for $i = 1, \ldots, m$, and panel (b) packs the rows of the input matrices back-to-back in an array. For the $b = 2$ case, the first row of the resulting array contains the elements

$$[0\ 0\ A_1[1,1]\ A_1[1,2]\ A_1[1,3]\ 0\ 0\ A_2[1,1]\ A_2[1,2]\ A_2[1,3]\ \ldots$$

$$0\ 0\ A_m[1,1]\ A_m[1,2]\ A_m[1,3]].$$

For each row of each $\mathbf{A}_i$ in this array, the TRAVOLTA code uses $2b + 1$ memory locations. When the input banded matrix has no elements to the left of the element (i, i), for $i = 1, 2, \ldots, b - 1$, the empty slots are filled with zeros. The second row of this array contains the elements

$$[0\ A_1[2,1]\ A_1[2,2]\ A_1[2,3]\ A_1[2,4]\ 0\ A_2[2,1]\ A_2[2,2]$$

$$A_2[2,3]\ A_2[2,4]\ \ldots\ 0\ A_m[2,1]\ A_m[2,2]\ A_m[2,3]\ A_m[2,4]].$$

The third row of this array contains the elements

$$[A_1[3,1]\ A_1[3,2]\ A_1[3,3]\ A_1[3,4]\ A_1[3,5]\ A_2[3,1]\ A_2[3,2]\ A_2[3,3]$$

$$A_2[3,4]\ A_2[3,5]\ \ldots\ A_m[3,1]\ A_m[3,2]\ A_m[3,3]\ A_m[3,4]\ A_m[3,5]],$$

and likewise for the remaining rows of this array. When the $n - b + 1, \ldots, n$ rows of the input matrices have no elements to the right of the (i, i) element, these slots have to be filled with zeros. Thus, the last row of the array contains

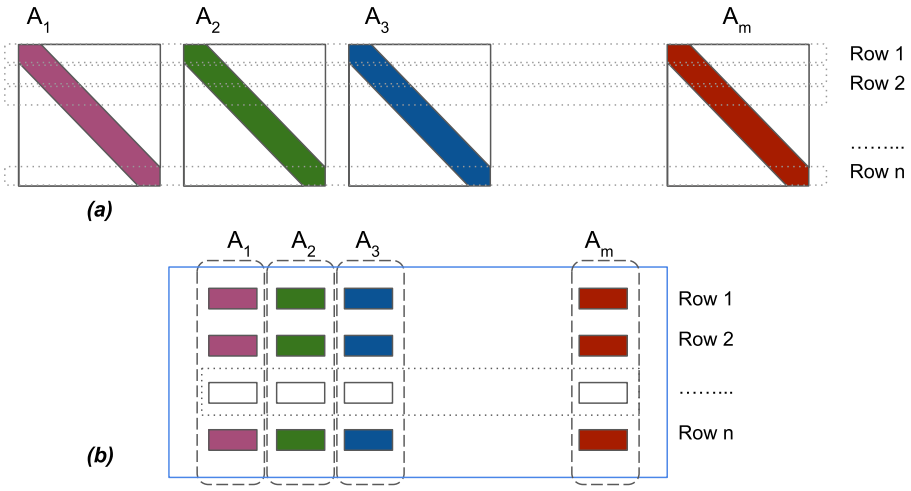

Fig. 2. Data layout of the input banded matrices. (a) Set of banded matrices $\mathbf{A}_1$, $\mathbf{A}_2$, ..., $\mathbf{A}_m$. (b) The rows of the banded matrices are stored back-to-back in one single array.

$[A_1[n,n-2]\ A_1[n,n-1]\ A_1[n,n]\ 0\ 0\ A_2[n,n-2]$

$A_2[n,n-1]\ A_2[n,n]\ 0\ 0\ \ \ldots\ A_m[n,n-2]\ A_m[n,n-1]\ A_m[n,n]\ 0\ 0]$.

To simplify the notation from this point forward, we denote the array of packed matrices as matrix **A**. With these newly-constructed input matrices, we next present our LU decomposition approach.

Algorithm 2: GPU kernel for LU decomposition of a set of banded matrices $\mathbf{A}_1\mathbf{A}_2...\mathbf{A}_m$ packed back-to-back.

Input: Packed matrix **A** with size $n*(2*b+1)*batchSize$, where n = number of rows of **A**, b = size of the band, and $batchSize$ = number of packed matrices in **A**.
Output: Packed matrices **L** and **U** each with size $n*(b+1)*batchSize$.

```
1  /* Working with zero-based indexing */
2  SA[2b][(2b+1)(BLOCKSIZE)] /* Tile in shared memory */
3  threadId = blockDim.x * blockIdx.x + threadIdx.x
4  Center = (2b+1) * threadIdx.x + b /* Tile center */
5  RowtoRead = 0 ; RowtoWrite = 0
6  for RowtoRead = 0 to (2b-1) do
7  |  ReadRow(A, n, RowtoRead, threadId, SA)
   end
8  ThreadBarrier
9  for j = 0 to (n-1) do
10 |  for row = 1 to min(b, n-j) do
11 |  |  SA[row][Center-row] = SA[row][Center-row]/SA[0][Center]
   |  end
12 |  for row = 1 to min(b, n-j) do
13 |  |  t = SA[row][Center-row]
14 |  |  for col = 1 to min(b, n-j) do
15 |  |  |  SA[row][Center-row+col] = SA[row][Center-row+col] - t *
   |  |  |    SA[0][Center+col]
   |  |  end
   |  end
16 |  ThreadBarrier
17 |  WriteRow(L, n, RowtoWrite, threadId, SA[0])
18 |  WriteRow(U, n, RowtoWrite, threadId, SA[0])
19 |  ThreadBarrier
20 |  MoveTileUp(2b, threadIdx.x, SA)
21 |  ThreadBarrier
   |  if RowtoRead ≤ (n-1) then
22 |  |  ReadRow(A, n, RowtoRead, threadId, SA)
   |  end
23 |  ThreadBarrier
24 |  RowtoRead = RowtoRead + 1; RowtoWrite = RowtoWrite + 1
   end
```

Using the approaches described in Algorithm 2 in the Supplemental Material along with the matrix-packing approach described above, we summarize our batched LU decomposition kernel on GPUs in Algorithm 2. In line 2, the kernel declares a shared memory tile, which has $2b$ rows having $(2b+1)BLOCKSIZE$ complex elements, where b is the size of the band (2 in our implementation), and $BLOCKSIZE$ is the number of threads per GPU block (32 in our code). In line 3, the routine declares a variable that uniquely identifies a GPU thread for each matrix $\mathbf{A}_i$ present in **A**. In line 4, the routine declares a variable that points to the element $\mathbf{A}_i[j,k]$ in tile SA for the current thread corresponding to $i = threadId$, row j, and column k. In line 7, the tile is populated so that all the threads in the block cooperate to read the rows of **A**. Specifically, instead of reading $(2b+1)BLOCKSIZE$ complex numbers per row, the threads cooperate to read $2*(2b+1)BLOCKSIZE$ double-precision floating point numbers. In line 8, the threads in the block synchronize their work, and this concludes the initialization part.

The code described in line 9 and below is similar to the banded LU decomposition shown in the Supplemental Material. To take into account the layout of the rows in the tile, a few changes are required. For example, the current row j of each input matrix is always placed in row zero of the tile SA as shown in line 11 (designated as $SA[0][Center]$). Note that as the row index increases, the column index in the tile decreases (designated by the instruction $A[row][Center-row]$). Thus, the tile elements $SA[0][Center]$ and $SA[1][Center-1]$ correspond to the elements $A_i[j,k]$ and $A_i[j+1,k]$ for a matrix $\mathbf{A}_i$, rows j and $j+1$, and column k. Line 15 shows that, for a given matrix $\mathbf{A}_i$, browsing the elements of the rows in the tile require the use of two variables: one that finds the beginning of the data within the row (given by the variable row) and another that points to the column (the variable col). In lines 17 and 18, the rows of the matrices **L** and **U** are written to the GPU main memory. To allow for coalesced writes, this operation is executed in a cooperative fashion similar to the read operation. In line 20, the tile is moved up to clear space for a new row (the first row of the tile is no longer required). In line 22, the new row of **A** is brought into the last row of the tile. The $ThreadBarrier$ instructions ensure the threads execute the computation in an orderly fashion. In our design, the number of GPU threads in execution is proportional to the number of matrices packed in **A**. Moreover, while the number of input matrices is m, our kernel only processes $batchSize$ matrices at the time to account for resources (i.e., main memory) available on the GPU. Having described the LU decomposition kernel, we now describe the kernels responsible for computing the inverses below.

4.2. Batched inverses: forward substitution

The next step is to compute the inverses of the banded matrices $\mathbf{A}_i$ given their banded factors $\mathbf{L}_i$ and $\mathbf{U}_i$, for $i = 1,\ldots,m$. The first task is

to compute $\mathbf{L}_i^{-1}$. Computing this inverse is equivalent to solving the set of linear equations $\mathbf{L}_i \left[\mathbf{y}_1\ \mathbf{y}_2\ \dots\ \mathbf{y}_n\right] = \left[\mathbf{e}_1\ \mathbf{e}_2\ \dots\ \mathbf{e}_n\right]$ where the column vectors $\mathbf{y}_k$ and $\mathbf{e}_k$ are the kth columns of $\mathbf{L}_i^{-1}$ and the identity matrix $\mathbf{I}$, respectively. The forward substitution routine that solves the linear equation $\mathbf{L}_i\mathbf{y}_k = \mathbf{e}_k$ for banded lower triangular matrices is shown in the Supporting Material. Before computing the inverses of $\mathbf{L}_i$, we generate an array of identity matrices $\mathbf{W}$ such that $\mathbf{W} = \left[\mathbf{I}_1\ \mathbf{I}_2\ \dots\ \mathbf{I}_m\right]$. In this array, the rows of matrix $\mathbf{I}_1$ are written first (row 1, 2, and so on), followed by the rows of $\mathbf{I}_2$, until the rows of matrix $\mathbf{I}_m$ are written. With the matrix of packed matrices $\mathbf{L}$ (which contain $\mathbf{L}_1, \mathbf{L}_2, \dots, \mathbf{L}_m$) and the array of identity matrices $\mathbf{W}$ (which contain $\mathbf{I}_1, \mathbf{I}_2, \dots, \mathbf{I}_m$), we can compute the inverses. Algorithm 3 shows our batched forward substitution routine on GPUs.

Algorithm 3: GPU kernel to solve the set of linear equations $\mathbf{L}_i \left[\mathbf{y}_1\ \mathbf{y}_2\ \dots\ \mathbf{y}_n\right] = \left[\mathbf{e}_1\ \mathbf{e}_2\ \dots\ \mathbf{e}_n\right]$ for $i = 1, \dots, m$.

Input: Packed matrix $\mathbf{L}$ with size $n*(b+1)*batchSize$, where n = number of columns, b = size of the band, and $batchSize$ = number of packed matrices; set of matrices $\mathbf{W} = [\mathbf{I}_1\ \mathbf{I}_2\ \dots\ \mathbf{I}_m]$ each with size $n \times n$.
Output: Matrices $\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_m$ each with size $n \times n$

```
1  /* Working with zero-based indexing */
2  matrixId = blockIdx.y /* Matrix index */
3  colId = blockDim.x * blockIdx.x + threadIdx.x /* Column index */
4  SA[(b+1)(BLOCKSIZE)] /* Tile in shared memory */
5  Location = (b+1) * threadIdx.x
6  Register[(b+1)] /* Local array */
7  Register[0] = Complex(0,0)
8  Register[1] = W[n*n*matrixId + colId]
9  Register[2] = Complex(0,0)
10 Y[n*n*matrixID + colId] = Register[1]; /* y[0] = e[0] */
11 RowtoRead = 1
12 ReadRow(L, n, RowtoRead, threadId.x, SA)
13 RowtoRead = RowtoRead + 1
14 ThreadBarrier
15 for j = 1 to (n-1) do
16     z = (j == 1) ? 1 : 0
17     t = Complex(0,0)
18     for k = max(j-b,0) to (j-1) do
19         t = t + SA[0][Location + z] * Register[z]
           z = z + 1
       end
20     Result = (Register[2] - t) /* No division, L_i is unitary */
21     Y[n*n*matrixID + n*j + colId] = Result
22     Register[0] = Register[1]
23     Register[1] = Result
24     ThreadBarrier
       if (RowtoRead ≤ (n-1)) then
25         ReadRow(L, n, RowtoRead, threadId.x, SA)
26         Register[2] = W[n*n*matrixId + n*RowtoRead + colId]
       end
27     ThreadBarrier
28     RowtoRead = RowtoRead + 1
   end
```

In this kernel, each thread solves a linear equation of the form $\mathbf{L}_i\mathbf{y}_k = \mathbf{e}_k$ with $i = matrixId$ and $k = colId$ as shown in lines 2 and 3. In line 4, a shared memory tile is declared, which is used to store the rows of the banded matrices $\mathbf{L}_i$. The variable *Location* is declared in line 5, which points to the first element of the banded matrix $\mathbf{L}_i$ in *SA*. To improve performance, a vector containing the latest set of elements of $\mathbf{e}_k$ and $\mathbf{y}_k$ is maintained in registers, as shown in line 6. In line 8, the element $Register[1]$ is set to $e_k[0]$, which is the first element of the kth column of matrix $\mathbf{I}_i$. Line 10 outputs the first element of vector $\mathbf{y}_k$. In line 12, all the threads in the block cooperatively read the first row of the packed matrix $\mathbf{L}$, and the threads subsequently synchronize their work.

After the initialization, starting from line 15, the remaining elements of the output vector $\mathbf{y}_k$ are calculated. The loop instruction in line 18 computes the dot product required in the forward substitution routine. The solution for $y_k[j]$ is computed in line 20, and this value is written into the main memory in the next line. Because the first entry of the array *Register* is no longer needed, we update the values as shown in lines 22 and 23. Thus, at the end of the first iteration, $Register[0]$ is set to $y_k[0]$, $Register[1]$ is set to $y_k[1]$, and $Register[2]$ is set to $e_k[2]$. Likewise, at the end of the second iteration, $Register[0] = y_k[1]$, $Register[1] = y_k[2]$, and $Register[2] = e_k[3]$. Finally, in line 25, a row of the array $\mathbf{L}$ is read into the tile. Similarly, a new element of the vector $\mathbf{e}_k$ is read. Finally, the ThreadBarrier instructions allow for the synchronization of the work among the threads in the block.

4.3. Batched inverses: backward substitution

Given the matrices $\mathbf{U}_1, \mathbf{U}_2, \dots, \mathbf{U}_m$ (computed in routine 2) and $\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_m$ (computed in routine 3), the next step is to solve the set of linear equations $\mathbf{U}_i\mathbf{X}_i = \mathbf{Y}_i$ for $i = 1, \dots, m$. Solving $\mathbf{U}_i\mathbf{X}_i = \mathbf{Y}_i$ is equivalent to solving n linear equations of the form $\mathbf{U}_i \left[\mathbf{x}_1\ \mathbf{x}_2\ \dots\ \mathbf{x}_n\right] = \left[\mathbf{y}_1\ \mathbf{y}_2\ \dots\ \mathbf{y}_n\right]$ where the vectors $\mathbf{x}_k$ and $\mathbf{y}_k$, for $k = 1, \dots, n$, are the columns of matrices $\mathbf{X}_i$ and $\mathbf{Y}_i$ respectively. Routine 4 shows our implementation of the batched backward substitution method in GPUs.

Algorithm 4: GPU kernel to solve set of linear equations $\mathbf{U}_i \left[\mathbf{x}_1\ \mathbf{x}_2\ \dots\ \mathbf{x}_n\right] = \left[\mathbf{y}_1\ \mathbf{y}_2\ \dots\ \mathbf{y}_n\right]$ for $i = 1, \dots, m$.

Input: Packed matrix $\mathbf{U}$ with size $n*(b+1)*batchSize$, where n = number of columns, b = size of the band, and $batchSize$ = number of packed matrices in $\mathbf{U}$; matrices $\mathbf{Y} = \left[\mathbf{Y}_1\ \mathbf{Y}_2\ \dots\ \mathbf{Y}_n\right]$, each with size $n \times n$.
Output: Matrices $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_m$ each size $n \times n$

```
1  /* Working with zero-based indexing */
2  matrixId = blockIdx.y /* Matrix index */
3  colId = blockDim.x * blockIdx.x + threadIdx.x /* Column index */
4  SA[(b+1)(BLOCKSIZE)] /* Tile in shared memory */
5  Location = (b+1) * threadIdx.x
6  Register[(b+1)] /* Local array */
7  Register[0] = Y[n*n*matrixId + n*(n-1) + coldId]
8  Register[1] = Complex(0,0)
9  Register[2] = Complex(0,0)
10 RowtoRead = n - 1
11 ReadRow(U, n, RowtoRead, threadId.x, SA)
12 RowtoRead = RowtoRead - 1
13 ThreadBarrier
14 for j = (n-1) to 0 do
15     z = 1
16     t = Complex(0,0)
17     for k = (j+1) to min(j+b, n-1) do
18         t = t + SA[0][Location + z] * Register[z]
           z = z + 1
       end
20     Result = (Register[0] - t)/SA[0][Location]
21     X[n*n*matrixID + n*j + colId] = Result
22     Register[2] = Register[1]
23     Register[1] = Result
24     ThreadBarrier
       if (RowtoRead ≥ 0) then
25         ReadRow(X, n, RowtoRead, threadId.x, SA)
26         Register[0] = Y[n*n*matrixId + n*RowtoRead + colId]
       end
27     ThreadBarrier
28     RowtoRead = RowtoRead - 1
   end
```

Algorithm 4 is very similar to Algorithm 3, and as a result, we only comment on the main differences. Similar to Algorithm 3, we use a *Register* to store the needed values of the vectors $\mathbf{x}_k$ and $\mathbf{y}_k$. In the beginning, $Register[0] = \mathbf{y}_k[n-1]$ and the other elements of *Register* are set to $Complex(0,0)$. In lines 15 to line 20 the routine solves for $x_k[j]$. At the end of the first iteration, the values of the registers are $Register[2] = Complex(0,0)$, $Register[1] = x_k[n-1]$, and $Register[0] = y_k[n-2]$. At the end of the second iteration, the values of the registers are $Register[2] = x_k[n-1]$, $Register[1] = x_k[n-2]$, and $Register[0] = y_k[n-3]$. The rest of the instructions are similar to the ones described in Algorithm 3 with the difference that the main loop is executed starting from the last row.

It is worth noting that Algorithm 2 described above corresponds to a summarized version of our batched LU decomposition code in GPUs.

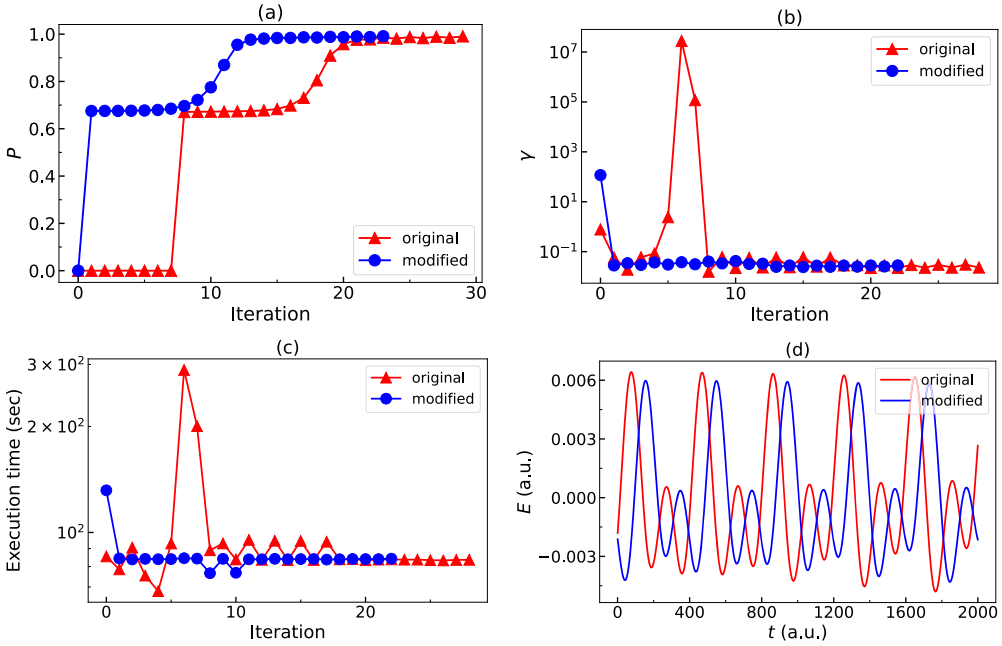

Fig. 3. Comparison of conventional and amplified gradient methods for QOC calculations for the $v_i = 1 \rightarrow v_f = 3$ transition in the Morse potential. (a) The amplified gradient method requires fewer iterations to reach a probability of 1. (b) Plot of the update rate, γ, for the gradient in each iteration. (c) Execution time of the line search in each iteration, which is positively correlated to γ in panel (b). (d) Optimal control pulses, $E(t)$, generated by the original and modified gradient methods. The electric fields have nearly the same functional form but differ by a global phase factor, which are physically insignificant for QOC calculations.

In our actual implementation, we have taken additional steps to improve performance by including one more row in the tile to allow data prefetching. In addition, we allow for lazy writing, i.e., instead of writing the matrices **L** and **U** directly to GPU memory, we write the data to an auxiliary tile and subsequently execute the writes further down in the pipeline. Moreover, all I/O operations are coalesced as the threads in the block read (write) from (to) contiguous memory addresses. The shared memory is used extensively, and the code does not have thread divergences due to **if** conditions. In addition, given enough matrices in the batch, multiple GPU blocks are created to ensure all the multiprocessors are busy. Algorithms 3 and 4 also take advantage of coalesced reads and writes, efficient use of the shared memory, efficient use of the register file, and high occupancy due to the large number of matrices being processed.

5. Computational results

Before examining the execution times of our GPU implementation, we compare the performance of the original gradient vs. our new amplified gradient modification.

5.1. Comparison between original and amplified gradient modification

We used the TRAVOLTA software package to examine QOC between the first and third excited states of the Morse potential in Eq. (8) and found that it takes fewer iterations to converge when the gradient is amplified. Fig. 3a shows that the amplified gradient enables a sudden update to P in the first iteration, while the conventional method maintains a nearly stagnant value of P for 7 iterations. Fig. 3b shows that the update rate, γ, in the conventional method only becomes sufficiently large after the 6th iteration, which prevents rapid convergence of $\epsilon(t)$ prior to that step. In contrast, a sufficiently large value of γ is obtained in the first iteration of the amplified gradient modification, which remedies the floating point underflow error for determining the update rate, explained previously in Sec. 3.

Our amplified gradient modification accelerates convergence not only by lowering the number of iterations but also by reducing the total execution time in the bisection line search. As explained in Section 3, a large γ increases the number of function calls to the forward propagation algorithm in line 7 of Algorithm 1. When the gradient is amplified, γ becomes much smaller, and the line search for γ can be calculated in less time, as shown in Fig. 3c. As discussed in Sec. 3, we typically set a threshold ($\gamma_{\text{thres}} = 10^8$ in this case) for the upper bound of the search interval to compute γ. However, one typically finds that the optimal γ may be larger than γ_{thres} when the probability, P, is small. The amplified gradient resolves this conflict. As shown in Fig. 3b, since the true γ in the 7th iteration ($\sim 10^{12}$) is much larger than γ_{thres}, the conventional approach requires two iterations (i.e., the 7th and 8th iterations shown near the top of panel 3b) to evaluate γ, which otherwise would have been accomplished in one iteration (i.e., the product of the computed γ in these two iterations, $\sim 10^7$ and $\sim 10^5$, is $\sim 10^{12}$). In contrast, our amplified gradient modification always gives a much better initial guess for calculating the optimal γ after only one iteration.

Fig. 3d compares the optimized control pulses generated by the original and our amplified gradient modification. The electric fields have nearly the same functional form but differ by a global phase factor, which we previously demonstrated to be physically insignificant for QOC calculations [20]. As such, our amplified gradient modification in the TRAVOLTA software package accelerates convergence (and reduces total execution time) without affecting any of the final results.

5.2. Accuracy of the hybrid implementation

To demonstrate the accuracy and computational performance of our GPU-enhanced TRAVOLTA code, we present two representative QOC examples: the Morse and asymmetric double-well potential. Section 3 in the Supplemental Material gives additional examples of input parameters that can be used and/or modified to carry out QOC calculations of other general potentials. Fig. 4 shows our QOC results executed with our custom CPU+GPU implementation for the Morse potential (Eq. (8)), and Fig. 5 shows results for the double-well potential given by

$$V(x) = \frac{x^4}{64} - \frac{x^2}{4} + \frac{x^3}{256}. \tag{11}$$

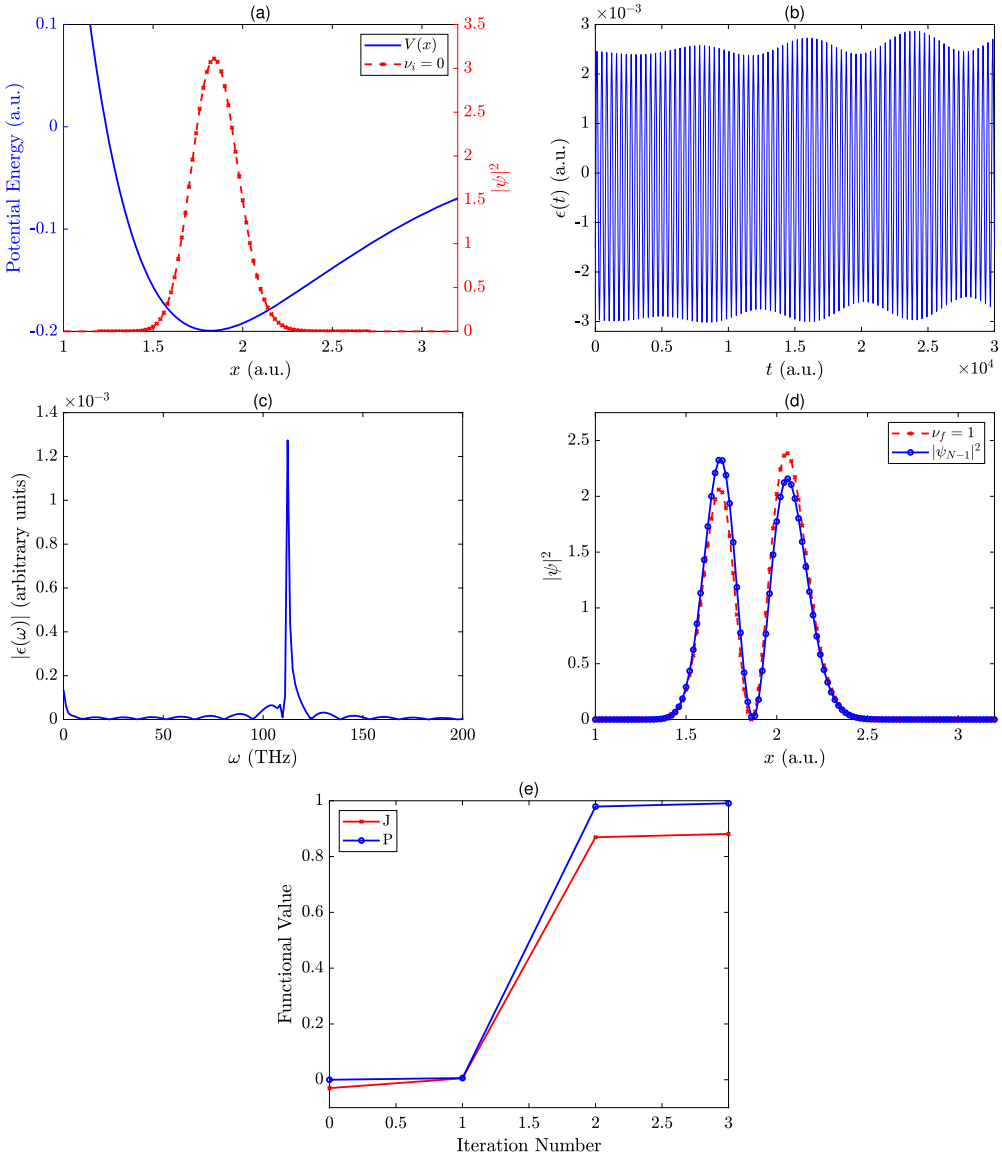

Fig. 4. (a) Morse potential energy (solid blue line) and norm-squared initial vibrational eigenstate, $|\psi_i(x)|^2$, with $\nu_i = 0$ (red dashed line). (b) Optimized electric field as a function of time for the $\nu_i = 0 \rightarrow \nu_f = 1$ transition. (c) Power spectrum (i.e., the Fourier transform) of the optimized electric field. (d) Norm-squared final target wavefunction $|\psi_f(x)|^2$ with $\nu_f = 1$ and the propagated wavefunction $|\psi_{(N-1)}|^2$ which achieves a transition probability of $P = 0.99$. (e) Objective functional, J, and transition probability, P, as a function of the number of iterations for a quantum control optimization of the $\nu_i = 0 \rightarrow \nu_f = 1$ transition in the Morse potential.

In both cases, the optimized electrical fields and power spectra plots are nearly identical with the results reported by Raza et al. [19]. It is worth noting that the optimized electric field in Fig. 5 differs by an overall global phase from its counterpart in Ref. 19, which we previously showed to give the same physical results and is, therefore, immaterial [20].

5.3. Computational performance on GPUs and comparisons

To illustrate the computational performance of our GPU parallelization scheme, we report the execution times of Algorithm 1 on one compute node of the *Perlmutter* supercomputer at NERSC [36]. Each GPU compute node has one CPU socket containing 64 AMD EPYC-7763 CPU cores 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, and we use one GPU. To assess the computational performance of Algorithm 1 across different hardware platforms, we compare execution times for three implementations: (1) a CPU baseline implementation that utilizes tuned numerical routines in the Cray BLAS LibSci library [37], (2) a standard CPU+GPU implementation that utilizes the CUDA BLAS (CUBLAS) libraries [38] (release 11.7), and (3) our tailored CPU+GPU implementation that utilizes the kernels described in Sec. 4. Our code is compiled with the HPE Cray GCC compiler, a wrapper based on the GNU GCC compiler (version 11.2) along with the CUDA compiler (release 11.7).

In our GPU implementations, we offloaded all operations involving matrices to the GPU. To compute line 10 in Algorithm 1, the CUBLAS library has routines for computing the inverse of multiple square matrices of $\mathbf{U}_{k+1/2}$ via a single call. In our custom code, instead of calling the CUBLAS libraries to compute the inverses of $\mathbf{U}_{k+1/2}$, we call our tailored GPU kernels described in Sec. 4. We set the size of the batch (i.e., the number of inverses that are computed in parallel) to 512. In addition, the time to read (write) the matrices from (to) disk is not reported, and we only record the time it takes for each hardware platform to execute the computations.

As mentioned previously, the results of our GPU-accelerated TRAVOLTA code are similar to the ones obtained in Ref. [19], which verifies

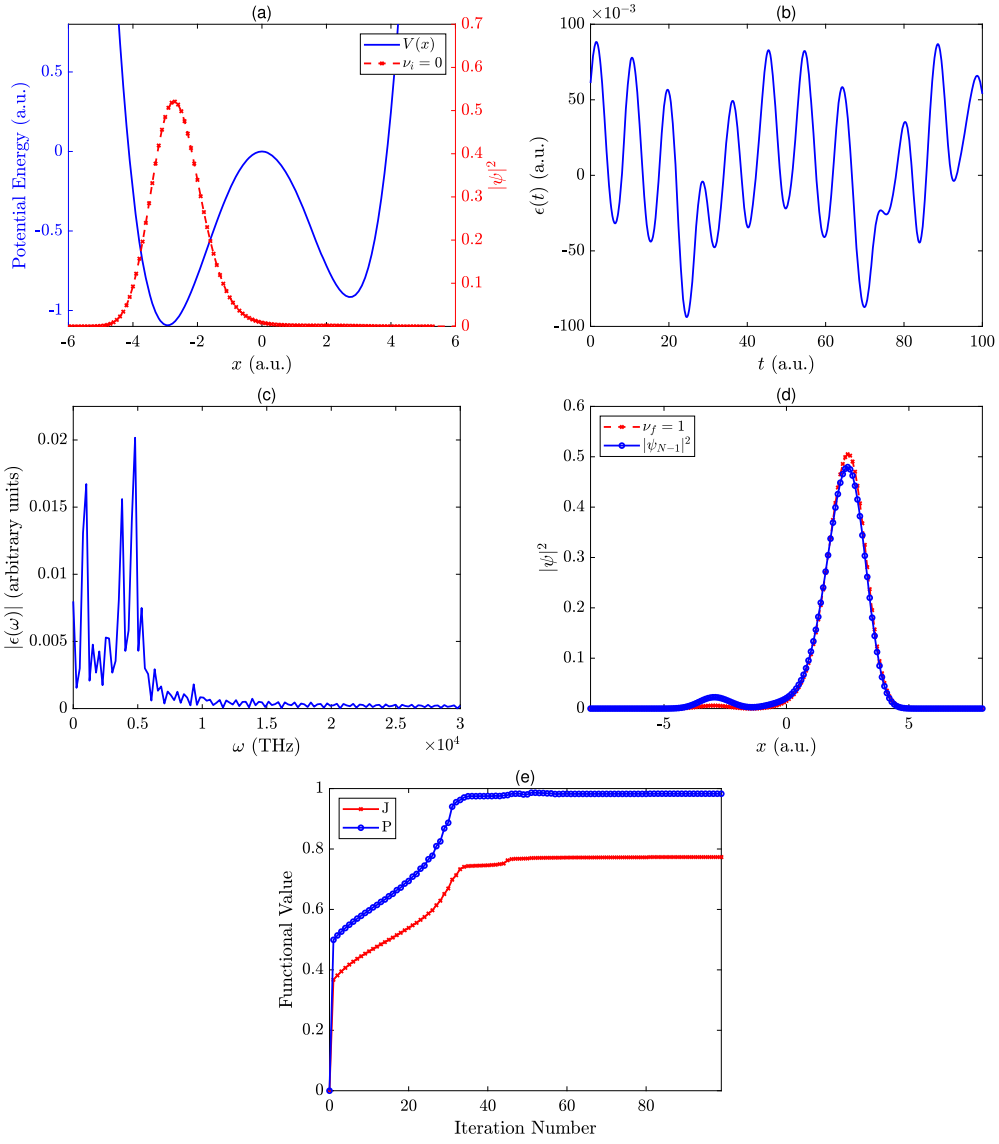

Fig. 5. (a) Asymmetric double-well potential energy (solid blue line) and norm-squared initial vibrational eigenstate, $|\psi_i(x)|^2$, with $\nu_i = 0$ (red dashed line). (b) Optimized electric field as a function of time for the $\nu_i = 0 \rightarrow \nu_f = 1$ transition. (c) Power spectrum (i.e., the Fourier transform) of the optimized electric field. (d) Norm-squared final target wavefunction $|\psi_f(x)|^2$ with $\nu_f = 1$ and the propagated wavefunction $|\psi_{(N-1)}|^2$ which achieves a transition probability of $P = 0.983$. (e) Objective functional, J, and transition probability, P, as a function of the number of iterations for a quantum control optimization of the $\nu_i = 0 \rightarrow \nu_f = 1$ transition in the asymmetric double-well potential for $T = 100$ and $\tau = 0.01$.

the accuracy of our implementation. To further test the performance of our GPU implementation to handle larger system sizes, we varied the parameter Δx (see Eq. (3)) from 0.1 to 0.0133, which increases the rows in the $\mathbf{U}_{k+1/2}$ matrices from 161 to 1021. Fig. 6 shows the execution times of our calculations for one iteration of the TRAVOLTA code (additional iterations involve the same computations as shown in Algorithm 1). In this figure, the speedup (shown in parentheses) is the ratio of the CPU code execution time to the standard/custom GPU code execution time for the same task (raw numerical values used to generate the plot are given in the Supplemental Material).

Fig. 6 shows that the standard CPU code (red vertical bars) is the slowest, even when using the high-performance Cray BLAS library, which has optimized routines to solve banded linear systems and compute inverses of banded matrices. It is worth noting that increasing the number of threads beyond 8 in our standard CPU code did not increase the performance substantially. In contrast, by offloading all matrix computations to the GPU, the standard CPU+GPU implementation (blue vertical bars in Fig. 6) decreases the execution time of the CPU implementation by a factor of 3.7 in the best case. Transferring matrices from the CPU to the GPU, and vice versa, is computationally expensive; nonetheless, the highly optimized routines in the CUBLAS library are able to significantly reduce computational execution times.

Most importantly, our custom kernels described in Section 4 speed up calculations by a factor of 6.7 on average; for large matrices, the speedup is even more impressive with a factor of 11.4. In addition to offloading compute-intensive operations, such as matrix·matrix and matrix·vector calculations to the GPU, we attribute the gains in performance to the following 5 improvements: (1) Our custom kernels work in batches and exploit parallel processing in Algorithm 1, line 7, which take a large number of banded complex matrices as input and compute the inverses simultaneously. Working with large batches allows for the efficient movement of matrices between the CPU and GPU and vice versa (i.e., it is more efficient to move large chunks of data between these devices than moving small chunks of data). In our implementation, the size of the batch (512) is limited by the size of the GPU memory and not by our algorithm. For GPUs having larger main

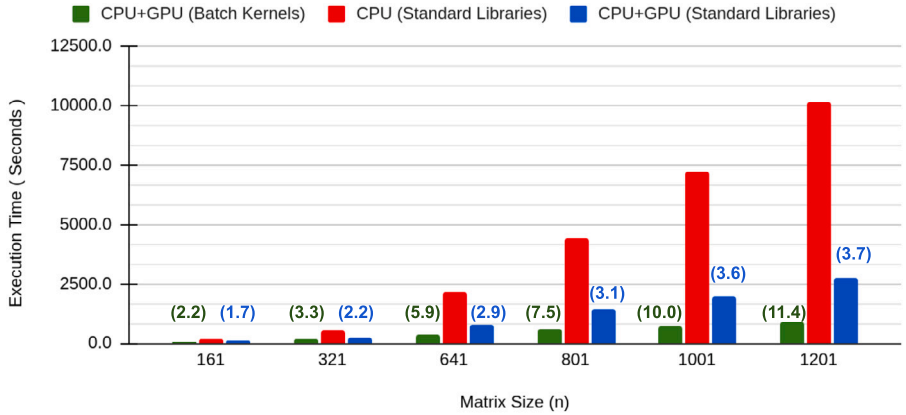

Fig. 6. Comparison of execution times as a function of matrix size. Computational speedups with respect to the standard CPU implementation are shown in parentheses on top of the bars.

memories or systems with multiple GPUs, this parameter can be increased effortlessly. (2) Our custom kernels exploit the banded structure of the matrices as shown in Algorithms 2, 3, and 4. Executing the LU decomposition on banded matrices is faster than decomposing square matrices [27]. As noted by Dongarra et al. [39], major improvements in performance can be achieved when the routines exploit the sparsity (i.e., the band) of the matrices. (3) The layout of the rows of the banded matrices, shown in Fig. 2, allows for coalesced reads and writes when Algorithm 2 is executed; as a result, our kernels are able to move data efficiently within the GPU. Also the data layout of the $\mathbf{L}_i$ and $\mathbf{Y}_i$ matrices allows for the efficient execution of I/O operations during the execution of Algorithm 3. Likewise, the data layout of the $\mathbf{U}_i$, $\mathbf{Y}_i$, and $\mathbf{X}_i$ matrices allows for the efficient execution of Algorithm 4. (4) By keeping the input matrices in tiles, our kernels can reuse the rows of the input matrices. For instance, in the case of Algorithm 2, one single row is reused up to $(b+1)$ times. (5) The GPU threads in Algorithms 2, 3, and 4 work in a cooperative and independent fashion. In the case of routine 2, operations such as reads, writes, and the movement of data within a tile is executed by all threads in the block in a cooperative fashion, which increases its efficiency. In this routine, once the data is in the tile, the threads are allowed to work in an independent fashion; i.e., a GPU thread is responsible for executing all the arithmetic calculations required by the decomposition method. The same ideas apply for routines 3, and 4. In addition, our kernels make use of other methods, including prefetching, lazy writing, extensive use of the register file, and minimal use of thread barriers, among others, which further increases computational efficiency.

6. Conclusions

In this work, we have developed and provided the open-source TRAVOLTA software package for accelerating QOC calculations on massively-parallelized GPUs. The TRAVOLTA code utilizes a new amplified gradient modification that prevents floating point underflow errors and accelerates the bisection line-search process for improved convergence. To enable additional performance enhancements, we offloaded computationally intensive operations such as matrix inverses, matrix·matrix, and matrix·vector multiplications to high-performance GPUs. To efficiently compute matrix inverses on GPUs, we implemented three customized high-performance kernels in our batched approach. The first kernel computes the LU decomposition of multiple banded matrices simultaneously, and two additional kernels compute the inverses via forward and backward substitution methods. In addition, our tailored kernels implement computational techniques such as data prefetching, coalesced read and writes, efficient utilization of shared memory and shared registers, efficient distribution of work among the GPU cores, minimization of thread divergences and synchronization points, lazy writing, and efficient movement of data between the CPU and GPU. These computational techniques are used in conjunction with recent batch computation methods to enable impressive parallelization of QOC calculations on modern multi-core GPUs.

To assess the accuracy and efficiency of our implementation, we applied the GPU-accelerated TRAVOLTA code to a variety of QOC systems and benchmarked its performance on state-of-the-art A100 GPUs on the *Perlmutter* supercomputer at NERSC. From these computational timing tests, we show that the TRAVOLTA code generates the same results as previous QOC benchmark calculations on CPUs but with a speedup that is more than 10 times faster. Most notably, our computational timings of the TRAVOLTA code demonstrate that its computational performance actually increases with system size compared to its CPU implementation. Looking forward, these algorithmic improvements and GPU-parallelization techniques could enable QOC calculations of larger systems that would otherwise be too time-consuming to run on CPUs. For example, QOC calculations in higher dimensions are intrinsically more computationally difficult since the size of the basis set used to construct the Hamiltonian scales exponentially (i.e., the basis set for a 2-dimensional example is a tensor product of two 1-dimensional basis sets). Another example where our GPU-parallelization techniques could enable significant performance gains is QOC in quantum computing, since the Hamiltonian increases as 2^n, where n is the number of qubits [7]. Since our GPU-accelerated routines show better performance on larger matrices, the techniques used in this work are expected to show even larger performance gains for all of these large quantum systems. Similarly, we anticipate that some of our computational techniques could be extremely useful for QOC calculations of systems with computationally intensive many-body quantum interactions (which would require additional mathematical operations on large matrices) that would significantly benefit from the algorithmic/parallelization enhancements used in this work.

CRediT authorship contribution statement

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

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.

Data availability

Data will be made available on request.

Acknowledgements

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.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cpc.2023.109017.

References

[1] D. Keefer, S. Thallmair, J.P.P. Zauleck, R. de Vivie-Riedle, A multi target approach to control chemical reactions in their inhomogeneous solvent environment, J. Phys. B, At. Mol. Opt. Phys. 48 (23) (2015) 234003.
[2] B.L. Brown, A.J. Dicks, I.A. Walmsley, Coherent control of ultracold molecule dynamics in a magneto-optical trap by use of chirped femtosecond laser pulses, Phys. Rev. Lett. 96 (17) (2006) 173002.
[3] D. Keefer, R. de Vivie-Riedle, Pathways to new applications for quantum control, Acc. Chem. Res. 51 (9) (2018) 2279–2286.
[4] K.C. Nowack, F. Koppens, Y.V. Nazarov, L. Vandersypen, Coherent control of a single electron spin with electric fields, Science 318 (5855) (2007) 1430–1433.
[5] M. Kues, C. Reimer, P. Roztocki, L.R. Cortés, S. Sciara, B. Wetzel, Y. Zhang, A. Cino, S.T. Chu, B.E. Little, et al., On-chip generation of high-dimensional entangled quantum states and their coherent control, Nature 546 (7660) (2017) 622–626.
[6] E.M. Fortunato, M.A. Pravia, N. Boulant, G. Teklemariam, T.F. Havel, D.G. Cory, Design of strongly modulating pulses to implement precise effective hamiltonians for quantum information processing, J. Chem. Phys. 116 (17) (2002) 7599–7606.
[7] 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) 043801.
[8] J. Cheng, H. Deng, X. Qia, AccQOC: accelerating quantum optimal control based pulse generation, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2020, pp. 543–555.
[9] F. Sauvage, F. Mintert, Optimal control of families of quantum gates, Phys. Rev. Lett. 129 (5) (2022) 050507.
[10] W. Zhu, J. Botina, H. Rabitz, Rapidly convergent iteration methods for quantum optimal control of population, J. Chem. Phys. 108 (5) (1998) 1953–1963.
[11] A.P. Peirce, M.A. Dahleh, H. Rabitz, Optimal control of quantum-mechanical systems: existence, numerical approximation, and applications, Phys. Rev. A 37 (12) (1988) 4950–4964.
[12] P. Brumer, M. Shapiro, Coherence chemistry: controlling chemical reactions [with lasers], Acc. Chem. Res. 22 (12) (1989) 407–413.
[13] M. Lysebo, L. Veseth, Quantum optimal control theory applied to transitions in diatomic molecules, Phys. Rev. A 90 (6) (2014) 063427.
[14] K. Kormann, S. Holmgren, H.O. Karlsson, A Fourier-coefficient based solution of an optimal control problem in quantum chemistry, J. Optim. Theory Appl. 147 (2010) 491–506.
[15] N. Dupont, G. Chatelain, L. Gabardos, M. Arnal, J. Billy, B. Peaudecerf, D. Sugny, D. Guéry-Odelin, Quantum state control of a Bose-Einstein condensate in an optical lattice, PRX Quantum 2 (4) (2021) 040303.
[16] 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.
[17] T. Caneva, T. Calarco, S. Montangero, Chopped random-basis quantum optimization, Phys. Rev. A 84 (2) (2011) 022326.
[18] V.F. Krotov, I. Feldman, An iterative method for solving optimal-control problems, Eng. Cybern. 21 (2) (1983) 123–130.
[19] 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.
[20] X. Wang, A. Kumar, C.R. Shelton, B.M. Wong, Harnessing deep neural networks to solve inverse problems in quantum dynamics: machine-learned predictions of time-dependent optimal control fields, Phys. Chem. Chem. Phys. 22 (40) (2020) 22889–22899.
[21] Y. Gao, X. Wang, N. Yu, B.M. Wong, Harnessing deep reinforcement learning to construct time-dependent optimal fields for quantum control dynamics, Phys. Chem. Chem. Phys. 24 (39) (2022) 24012–24020.
[22] 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: ACM Proceedings of the 3st Workshop on General-Purpose Computation on Graphics Processing Units, ACM, New York, NY, USA, 2010, pp. 63–74.
[23] J.M. Rodríguez, Acceleration of Compute-Intensive Applications on Field Programmable Gate Arrays, University of California, Riverside, 2020.
[24] G. Von Winckel, A. Borzì, Computational techniques for a quantum control problem with H^1-cost, Inverse Probl. 24 (3) (2008) 034007.
[25] M. Sprengel, G. Ciaramella, A. Borzì, A COKOSNUT code for the control of the time-dependent Kohn–Sham model, Comput. Phys. Commun. 214 (2017) 231–238.
[26] G. Paramonov, Coherent control of linear and nonlinear multiphoton excitation of molecular vibrations, Chem. Phys. 177 (1) (1993) 169–180.
[27] 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.
[28] D.S. Watkins, Fundamentals of Matrix Computations, 1st edition, John Wiley & Sons, 111 River St, Hoboken, NJ, USA, 1991.
[29] L.N. Trefethen, D. Bau III, Numerical Linear Algebra, 1st edition, Siam, 3600 Market Street, 6th Floor, Philadelphia, PA, 19104, USA, 1997.
[30] M. Jacquelin, L. Lin, N. Wichmann, C. Yang, Enhancing scalability and load balancing of parallel selected inversion via tree-based asynchronous communication, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 192–201.
[31] S. Peng, S.X.-D. Tan, GLU3. 0: fast GPU-based parallel sparse LU factorization for circuit simulation, IEEE Des. Test 37 (3) (2020) 78–90.
[32] 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.
[33] 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.
[34] D.B. Kirk, W.-M.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 2nd edition, Morgan Kaufmann, 225 Wyman Street, Waltham, MA, 02451, USA, 2013.
[35] 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.
[36] NERSC, NERSC technical documentation, https://docs.nersc.gov. (Accessed 22 November 2023), 2022.
[37] NERSC, The cray BLAS libraries, https://docs.nersc.gov/development/libraries/libsci. (Accessed 22 November 2023), 2022.
[38] NVIDIA Incorporated, CUDA toolkit documentation, https://docs.nvidia.com/cuda/. (Accessed 22 November 2023), 2013.
[39] J. Dongarra, L. Grigori, N.J. Higham, Numerical algorithms for high-performance computational science, Philos. Trans. R. Soc. A 378 (2166) (2020) 20190066.