Reducing Memory Requirements of Quantum Optimal Control*

Sri Hari Krishna Narayanan[1], Thomas Propson[2], Marcelo Bongarti[3], Jan Hückelheim[1], and Paul Hovland[1]

[1] Argonne National Laboratory, Lemont, IL 60439, USA
{snarayan, jhueckelheim, hovland}@anl.gov
[2] University of Chicago, Chicago, IL, USA
tcpropson@uchicago.edu
[3] Weierstrass Institute for Applied Analysis and Stochastics, Berlin, Germany
bongarti@wias-berlin.de

Abstract. Quantum optimal control problems are typically solved by gradient-based algorithms such as GRAPE, which suffer from exponential growth in storage with increasing number of qubits and linear growth in memory requirements with increasing number of time steps. These memory requirements are a barrier for simulating large models or long time spans. We have created a nonstandard automatic differentiation technique that can compute gradients needed by GRAPE by exploiting the fact that the inverse of a unitary matrix is its conjugate transpose. Our approach significantly reduces the memory requirements for GRAPE, at the cost of a reasonable amount of recomputation. We present benchmark results based on an implementation in JAX.

Keywords: Quantum · Autodiff · Memory.

1 Introduction

Quantum computing is computing using quantum-mechanical phenomena, such as superposition and entanglement. It holds the promise of being able to efficiently solve problems that classical computers practically cannot. In quantum computing, quantum algorithms are often expressed by using a quantum circuit model, in which a computation is a sequence of quantum gates. Quantum gates are the building blocks of quantum circuits and operate on a small number of qubits, similar to how classical logic gates operate on a small number of bits in conventional digital circuits.

* This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under the Accelerated Research in Quantum Computing and Applied Mathematics programs, under contract DE-AC02-06CH11357, and by the National Science Foundation Mathematical Sciences Graduate Internship. We gratefully acknowledge the computing resources provided on Bebop and Swing, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

Practitioners of quantum computing must map the logical quantum gates onto the physical quantum devices that implement quantum gates through a process called *quantum control*. The goal of quantum control is to actively manipulate dynamical processes at the atomic or molecular scale, typically by means of external electromagnetic fields. The objective of *quantum optimal control* (QOC) is to devise and implement shapes of pulses of external fields or sequences of such pulses that reach a given task in a quantum system in the best way possible.

We follow the QOC model presented in [20]. Given an intrinsic Hamiltonian H_0, an initial state $|\psi_0\rangle$, and a set of control operators $H_1, H_2, \ldots H_m$, one seeks to determine, for a sequence of time steps $t_0, t_1, \ldots, t_N$, a set of control fields $u_{k,j}$ such that

$$\mathbb{H}_j = H_0 + \sum_{k=1}^{m} u_{k,j} H_k \tag{1}$$

$$U_j = e^{-i\mathbb{H}_j(t_j - t_{j-1})} \tag{2}$$

$$K_j = U_j U_{j-1} U_{j-2} \ldots U_1 U_0 \tag{3}$$

$$|\psi_j\rangle = K_j |\psi_0\rangle. \tag{4}$$

An important observation is that the dimensions of K_j and U_j are $2^q \times 2^q$, where q is the number of qubits in the system. One possible objective is to minimize the trace distance between K_N and a target quantum gate K_T:

$$F_0 = 1 - |\operatorname{Tr}(K_T^\dagger K_N)/D|^2, \tag{5}$$

where D is the Hilbert space dimension. The complete QOC formulation includes secondary objectives and additional constraints One way to address this formulation is by adding to the objective function weighted penalty terms representing constraint violation: $\min_{u_{k,j}} \left(\sum_{i=0}^{2} w_i F_i + \sum_{i=3}^{6} w_i G_i \right)$, where

$$F_1 = 1 - \frac{1}{n} \sum_j |\operatorname{Tr}(K_T^\dagger K_j)/D|^2 \tag{6}$$

$$F_2 = 1 - \frac{1}{n} \sum_j |\langle \psi_T | \psi_j \rangle|^2 \tag{7}$$

$$G_3 = 1 - |\langle \psi_T | \psi_n \rangle|^2 \tag{8}$$

$$G_4 = |u|^2 \tag{9}$$

$$G_5 = \sum_{k,j} |u_{j,k} - u_{k,j-1}|^2 \tag{10}$$

$$G_6 = \sum_j |\langle \psi_F | \psi_j \rangle|^2 \tag{11}$$

and ψ_F is a forbidden state.

Algorithm 1 Pseudocode for the GRAPE algorithm.

```
Guess initial controls $u_{k,j}$.
repeat
    Starting from $H_0$, calculate
        $\rho_j = U_j U_{j-1} \dots U_1 H_0 U_1^\dagger \dots U_{j-1}^\dagger U_j^\dagger$.
    Starting from $\lambda_N = K_T$, calculate
        $\lambda_j = U_{j+1}^\dagger \dots U_N^\dagger K_T U_N \dots U_j$.
    Evaluate $\frac{\partial \rho_j \lambda_j}{\partial u_{k,j}}$
    Update the $m \times N$ control amplitudes:
        $u_{j,k} \to u_{j,k} + \epsilon \frac{\partial \rho_j \lambda_j}{\partial u_{k,j}}$
until $\mathrm{Tr}\,(K_T^\dagger K_N) <$ threshold
return $u_{j,k}$
```

QOC can be solved by several algorithms, including the gradient ascent pulse engineering (GRAPE) algorithm [17]. A basic version of GRAPE is shown in Algorithm 1. The derivatives $\frac{\partial \rho_j \lambda_j}{\partial u_{k,j}}$ required by GRAPE can be calculated by hand coding or finite differences. Recently, these values have been calculated efficiently by automatic differentiation (AD or autodiff) [20].

AD is a technique for transforming algorithms that compute some mathematical function into algorithms that compute the derivatives of that function [3, 12, 21]. AD works by differentiating the functions intrinsic to a given programming language (`sin()`, `cos()`, `+`, `-`, etc.) and combining the partial derivatives using the chain rule of differential calculus. The associativity of the chain rule leads to two main methods of combining partial derivatives. The forward mode combines partial derivatives starting with the independent variables and propagating forward to the dependent variables. The reverse mode combines partial derivatives starting with the dependent variables and propagating back to the independent variables. It is particularly attractive in the case of scalar functions, where a gradient of arbitrary length can be computed at a fixed multiple of the operations count of the function.

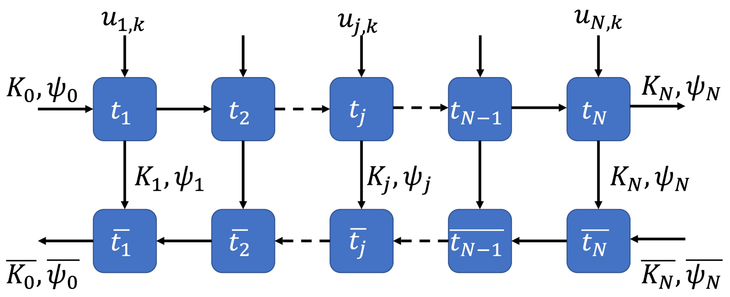

Fig. 1. Reverse-mode gradient computation for QOC. Each forward time step, j, computes K_j, U_j, and ψ_j and stores them in memory. The reverse sweep starts at time step N. Each reverse time step j uses the previously stored K_j, U_j, and ψ_j.

The reverse mode of AD is appropriate for QOC because of the large number $(m \times N)$ of inputs and the small number of outputs (the cost function(s)). As shown in Figure 1, standard reverse-mode AD stores the results of intermediate time steps $(K_j,\ U_j,\ \psi_j)$ in order to compute $\frac{\partial \rho_j \lambda_j}{\partial u_{k,j}}$. This implies that reverse-mode AD requires additional memory that is exponentially proportional to q. Current QOC simulations therefore are limited in both the number of qubits that can be simulated and the number of time steps in the simulation. In this work we explore the suitability of *checkpointing* as well as *unitary matrix reversibility* to overcome this additional memory requirement.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents our approach to reducing the memory requirements of QOC. Section 4 details the QOC implementation in JAX, and the evaluation of this approach is presented in Section 5. Section 6 concludes the paper and discusses future work.

2 Related Work

QOC has been implemented in several packages, such as the Quantum Toolbox in Python (QuTIP) [15,16]. In addition to GRAPE, QOC can be solved by using the chopped random basis (CRAB) algorithm [6, 9]. The problem is formulated as the extremization of a multivariable function, which can be numerically approached with a suitable method such as steepest descent or conjugate gradient. If computing the gradient is expensive, CRAB can instead use a derivative-free optimization algorithm. In [20], the AD capabilities of TensorFlow are used to compute gradients for QOC.

Checkpointing is a well-established approach in AD to reduce the memory requirements of reverse-mode AD [10, 18]. In short, checkpointing techniques trade recomputation for storing intermediate states; see Section 3.1 for more details. For time-stepping codes, such as QOC, checkpointing strategies can range from simple periodic schemes [12], through binomial checkpointing schemes [11] that minimize recomputation subject to memory constraints, to multilevel checkpointing schemes [2, 23] that store checkpoints to a multilevel storage hierarchy. Checkpointing schemes have also been adapted to deep neural networks [4, 7, 14, 22] and combined with checkpoint compression [8, 19].

3 Reducing Memory Requirements

We explore three approaches to reduce the memory required to compute the derivatives for QOC.

3.1 Approach 1: Checkpointing

Checkpointing schemes reduce the memory requirements of reverse-mode AD by recomputing certain intermediate states instead of storing them. These schemes checkpoint the inputs of selected time steps in a *plain-forward* sweep. To compute the gradient, a stored checkpoint is read, followed by a forward sweep and a reverse sweep for an appropriate number of time steps. Figure 2 illustrates periodic checkpointing for a computation of 10 time steps and 5 checkpoints. In the case of QOC with N time steps and periodic checkpointing interval C, one must store $O(C + \frac{N}{C})$ matrices of size $2^q \times 2^q$.

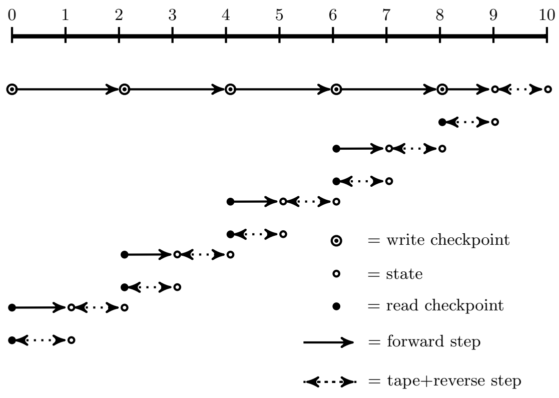

Fig. 2. Periodic checkpointing schedule for $N = 10$ time steps and 5 checkpoints ($C = 2$).

3.2 Approach 2: Reversibility of Unitary Matrices

The second approach is to exploit the property of unitary matrices that the inverse of a unitary matrices is its conjugate transpose.

$$U^\dagger U = UU^\dagger \tag{12}$$
$$U^\dagger = U^{-1} \tag{13}$$

Computing the inverse by exploiting the reversibility property of unitary matrices is an exact and inexpensive process. The use of the inverse allows us to compute K_{j-1} from K_j and ψ_{j-1} from ψ_j.

$$K_j = U_j U_{j-1} U_{j-2} \dots U_1 U_0 \tag{14}$$
$$K_{j-1} = U_j^\dagger K_j \tag{15}$$
$$\psi_{j-1} = \psi_0 K_{j-1} \tag{16}$$

Thus, one does not have to store any of the K_j matrices required to compute the adjoint of a time step. This approach reduces the memory requirement by half.

More importantly, using reversibility can unlock a further reduction in the memory requirements by not storing U_j, but rather using only the $u_{j,k}$ control values to recompute U_j. As a result, no intermediate computations need to be stored, and therefore the only additional requirements are to store the derivatives of the function with respect to the controls and other variables, thus basically doubling the memory requirements relative to the function itself.

3.3 Approach 3: Periodic Checkpointing Plus Reversibility

The reversibility property of unitary matrices is exact only in real arithmetic. A floating-point implementation may incur roundoff errors. Therefore, Equation 15 might not hold exactly, especially for large numbers of time steps. That is,

$$K_{j-1} \approx U_j^\dagger K_j \tag{17}$$
$$\psi_{j-1} \approx \psi_0 K_{j-1} \tag{18}$$

in floating-point arithmetic. Because K_{j-1} is computed each time step, the error continues to grow as the computation proceeds in the reverse sweep.

To mitigate this effect, we can combine the two approaches, checkpointing every C time steps and, during the reverse pass, instead of computing forward from these checkpoints, computing backward from the checkpoints by exploiting reversibility. Thus, floating-point errors in Equation 15 are incurred over a maximum of C time steps, and we reduce the number of matrices of size $2^q \times 2^q$ stored from $O(C + \frac{N}{C})$ to $O(\frac{N}{C})$.

Table 1. Overview of the object sizes, number of object instances stored for the forward computation, and number of additional instances that need to be stored for store-all, checkpointing, reversibility, and checkpointing plus reversibility. The total memory size in the last row is the product of the object size and the number of instances.

Variable	**Size**	**Forward**	**Store**	**Checkpoint**	**Revert**	**Rev + Ckp**
$u_{k,j}$	1	Nm	+0	+0	+0	+0
H	$2^q \cdot 2^q$	m	+0	+0	+0	+0
$\mathbb{H}_j$	$2^q \cdot 2^q$	1	+0	+0	+0	+0
U_j	$2^q \cdot 2^q$	1	$+N$	$+C$	+0	+0
K_j	$2^q \cdot 2^q$	1	$+N$	$+\frac{N}{C}+C$	+0	$+\frac{N}{C}$
ψ_j	2^q	1	$+N$	$+\frac{N}{C}+C$	+0	$+\frac{N}{C}$
Mem ($\mathcal{O}$)		$2^{2q}m + Nm$	$+2^{2q}N$	$+2^{2q}\left(\frac{N}{C}+C\right)$	+0	$+2^{2q}\frac{N}{C}$

3.4 Analysis of memory requirements

Table 1 summarizes the memory requirements for the forward pass of the function evaluation as well as the added cost of the various strategies for computing the gradient. Conventional AD, which stores the intermediate states U_j, K_j, and ψ_j at every time step, incurs an additional storage cost proportional to the number of time steps *times* the size of the $2^q \times 2^q$ matrices. Periodic checkpointing reduces the number of matrices stored to $\frac{N}{C} + C$. The checkpointing interval that minimizes this cost occurs when $\frac{\partial}{\partial C}\left(\frac{N}{C} + C\right) = \frac{-N}{C^2} + 1 = 0$, or $C = \sqrt{N}$. Exploiting reversibility enables one to compute U_j from $u_{j,k}$ and H_k and K_{j-1} from K_j and U_j, resulting in essentially zero additional memory requirements, beyond those required to store the derivatives themselves. Combining reversibility with periodic checkpointing eliminates the number of copies of U_j and K_j to be stored from $\frac{N}{C} + C$ to $\frac{N}{C}$.

4 Implementation

As an initial step we have ported to the JAX machine learning framework [5] a version of QOC that was previously implemented in TensorFlow. JAX provides a NumPy-style interface and supports execution on CPU systems as well as GPU and TPU (tensor processing unit) accelerators, with built-in automatic differentiation and just-in-time compilation capability. JAX supports checkpointing through the use of the `jax.checkpoint` decorator and allows custom derivatives to be created for functions using the `custom_jvp` decorator for forward mode and the `custom_vjp` decorator for reverse mode. To enable our work, we have contributed `jax.scipy.linalg.expm` to JAX to perform the matrix exponentiation operation using Padé approximation and to compute its derivatives. This code is now part of standard JAX releases.

Our approach requires us to perform checkpointing or use custom derivatives only for the Python function that implements Equations 1–4 for a single time step j or a loop over a block of time steps. Standard AD can be used as before for the rest of the code. By implication, our approach does not change for different objective functions.

We show here the implementation of the periodic checkpointing plus reversibility approach and direct the reader to our open source implementation for further details [1]. Listing 1 shows the primal code that computes a set of time steps. The function `evolve_step` computes Equations 1–4.

Listing 1 Simplified code showing a loop to simulate QOC for N time steps

```python
def evolve_step_loop(start, stop, cost_eval_step, dt, states, K,
                     control_eval_times, controls):
    for step in  range(start,stop):
        # Evolve the states and K to the next time step.
        time = step * dt
        states, K  = evolve_step(dt, states, K, time,
                                 control_eval_times, controls)
    return states, K
```

Listing 2 is a convenience wrapper with for the primal code. We decorate the wrapper with `jax.custom_vjp` to inform JAX that we will provide custom derivatives for it. User-provided custom derivatives for a JAX function consist

Listing 2 A wrapper to `evolve_step_loop()`, which will have derivatives provided by the user.

```
@jax.custom_vjp
def evolve_step_loop_custom(start, stop, cost_eval_step, dt, states,
                            K, control_eval_times, controls):
    states, K = evolve_step_loop(start, stop, cost_eval_step, dt,
                                 states,K,control_eval_times,
                                 controls)
    return states, K
```

of a forward sweep and a reverse sweep. The forward sweep must store all the information required to compute the derivatives in the reverse sweep. Listing 3 is the provided forward sweep. Here, as indicated in Table 1, we are storing the K matrix and the state vector. Note that this form of storage is effectively a checkpoint, even though it does not use `jax.checkpoint`.

Listing 3 Forward sweep of the user-provided derivatives.

```
def evolve_loop_custom_fwd(start, stop,cost_eval_step, dt, states, K,
                           control_eval_times, controls):
    states, K = _evaluate_schroedinger_discrete_loop_inner(
                           start, stop, cost_eval_step, dt, states, K,
                           control_eval_times, controls)
    #Here we store the final state and K for use in the backward pass
    return (states,K), (start, stop, cost_eval_step, dt, states,
           K, control_eval_times,controls)
```

Listing 4 is a user-provided reverse sweep. It starts by restoring the values passed to it by the forward sweep. While looping over time steps in reverse order, it recomputes Equations 1–2. It then computes Equations 13, 15, and 16 to retrieve K_{j-1} and ψ_{j-1}, which are then used to compute the adjoints for the time step. The code to compute the adjoint of the time step was obtained by the source transformation AD tool Tapenade [13].

5 Experimental Results

We compared standard AD, periodic checkpointing, and full reversibility or periodic checkpointing with reversibility, as appropriate. We conducted our experiments on a cluster where each compute node was connected to 8 NVIDIA A100 40GB GPUs. Each node contained 1TB DDR4 memory and 320GB GPU memory. We validated the output of the checkpointing and reversibility approaches against the standard approach implemented using JAX. We used the JAX memory profiling capability in conjunction with `GO pprof` to measure the memory needs for each case. We conducted three sets of experiments to evaluate the approaches, varying the number of qubits, the number of time steps, or the checkpoint period.

Listing 4 User-provided reverse sweep that exploits reversibility.

```
def evolve_loop_custom_bwd(res,g_prod):
  #Restore all the values stored in the forward sweep
  start, stop, cost_eval_step, dt, states,
        K, control_eval_times, controls = res
  _M2_C1 = 0.5
  controlsb = jnp.zeros(controls.shape, states.dtype)
  #Go backwards in time steps
  for i in  range(stop-1,start-1,-1):
    #Reapply controls to compute a step unitary matrix
    time = i * dt
    t1 = time + dt * _M2_C1
    x = t1
    xs = control_eval_times
    ys = controls
    index = jnp.argmax(x <= xs)
    y = ys[index - 1] + (((ys[index] - ys[index - 1]) /
        (xs[index] - xs[index - 1])) * (x - xs[index - 1]))
    controls_ = y
    hamiltonian_ = (SYSTEM_HAMILTONIAN
                + controls_[0] * CONTROL_0
                + jnp.conjugate(controls_[0]) * CONTROL_0_DAGGER
                + controls_[1] * CONTROL_1
                + jnp.conjugate(controls_[1]) * CONTROL_1_DAGGER)
    a1 = -1j * hamiltonian_
    magnus = dt * a1
    step_unitary, f_expm_grad = jax.vjp(jax.scipy.linalg.expm, (magnus),
    has_aux=False)
    #Exploit reversibility of unitary matrix
    #and calculate previous state and K
    step_unitary_inv=jnp.conj(jnp.transpose(step_unitary))
    states=(jnp.matmul(step_unitary_inv,states))
    K=(jnp.matmul(step_unitary_inv,K))
    _, f_matmul = jax.vjp(jnp.matmul,step_unitary, states)
    _, f_matmul_K = jax.vjp(jnp.matmul,step_unitary, K)
    #Go backwards for the timestep
    step_unitaryb,Kb=f_matmul_K(g_prod[1])
    step_unitaryb,statesb=f_matmul(g_prod[0])
    magnusb = f_expm_grad(step_unitaryb)
    a1b=dt*magnusb[0]
    hamiltonian_b = jnp.conjugate(-1j)*a1b
    controls1b=jnp.array((jnp. sum(jnp.conjugate(CONTROL_0)*hamiltonian_b) +
      jnp.conjugate(jnp. sum(jnp.conjugate(CONTROL_0_DAGGER)*hamiltonian_b)),
      jnp. sum(jnp.conjugate(CONTROL_1)*hamiltonian_b) +
      jnp.conjugate(jnp. sum(jnp.conjugate(CONTROL_1_DAGGER)*hamiltonian_b))),
      dtype=hamiltonian_b.dtype)
    tempb = (x-control_eval_times[index-1])*controls1b/
      (control_eval_times[index]-control_eval_times[index-1])
    controlsb=jax.ops.index_update(controlsb,
      jax.ops.index[index-1],controlsb[index-1]+controls1b - tempb)
    controlsb=jax.ops.index_update(controlsb,
      jax.ops.index[index],controlsb[index]+tempb)
    g_prod=statesb,Kb
  return (0.0,0.0,0.0,0.0,statesb,Kb,0.0,-1*controlsb)

  evolve_loop_custom.defvjp(evolve_loop_custom_fwd, evolve_loop_custom_bwd)
```

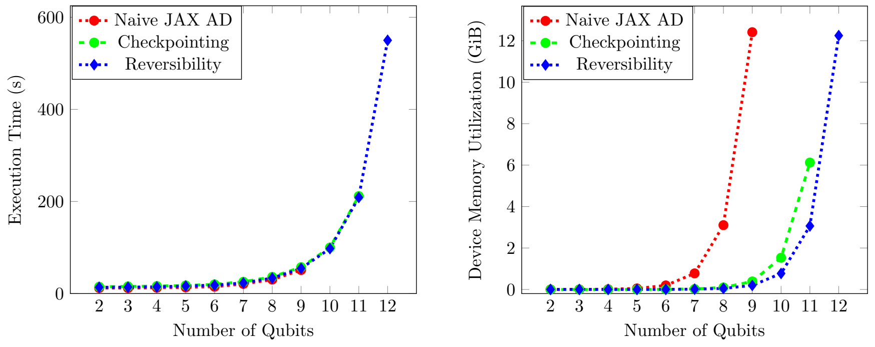

Fig. 3. Comparison of execution time and device memory requirements for standard AD, periodic checkpointing, and full reversibility with increasing number of qubits. The QOC simulation consisted of 100 time steps with a checkpoint period of 10.

5.1 Vary Qubits

We first varied the number of qubits q, keeping the number of time steps fixed at 100 and the checkpoint period fixed at $C = \sqrt{N} = 10$. Figure 3 shows the memory consumed by standard AD, periodic checkpointing, and full reversibility. One can see that the device memory requirements for the standard approach are highest whereas the requirements for reversibility are lowest, although all three grow exponentially as a function of q, as predicted by the analysis in Section 3. Furthermore, we note that the standard approach can be executed for a maximum of 9 qubits and runs out of available device memory on the 10th qubit. The periodic checkpointing approach can be run for 11 qubits and runs out of available device memory on the 12th. The full reversibility approach can be run for 12 qubits and exceeds available device memory on the 13th. Figure 3 (left) also shows the execution time for the various approaches. The times are similar for the cases that can be executed before running out of memory. As expected, the time grows exponentially as a function of q.

5.2 Vary Time Steps

Next we fixed the number of qubits at $q = 8$ or $q = 9$ and varied the number of time steps, N. For periodic checkpointing we used the optimal checkpoint period, $C = \sqrt{N}$. We expect the time to be roughly linear in N and independent of C because every U_j and K_j must be computed once during the forward pass and one more time on the reverse pass. We expect periodic checkpointing and full reversibility to be slower than standard AD because they both trade some amount of recomputation for reduced storage requirements. We expect full reversibility to be somewhat faster than periodic checkpointing alone because periodic checkpointing must compute forward from the checkpoint, storing intermediate K_j along the way, while full reversibility skips the second forward pass and is able to restore K_j during the reverse pass directly from the controls

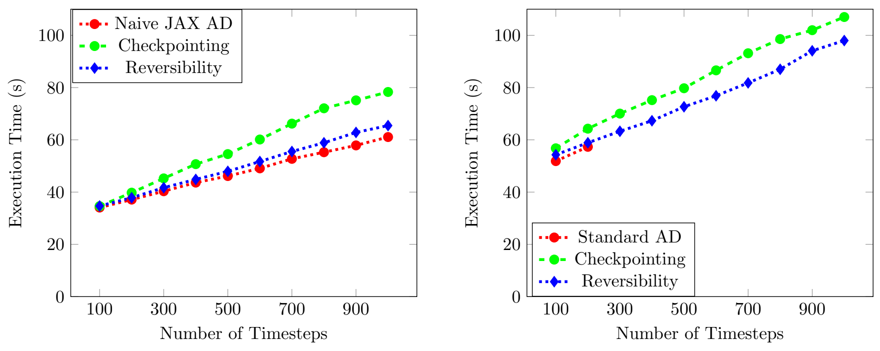

Fig. 4. Comparison of the execution time for standard AD, periodic checkpointing, and periodic reversibility approaches with increasing number of time steps. The QOC simulation consisted of 8 (left) or 9 (right) qubits. The checkpoint period was chosen to be the square root of the number of time steps.

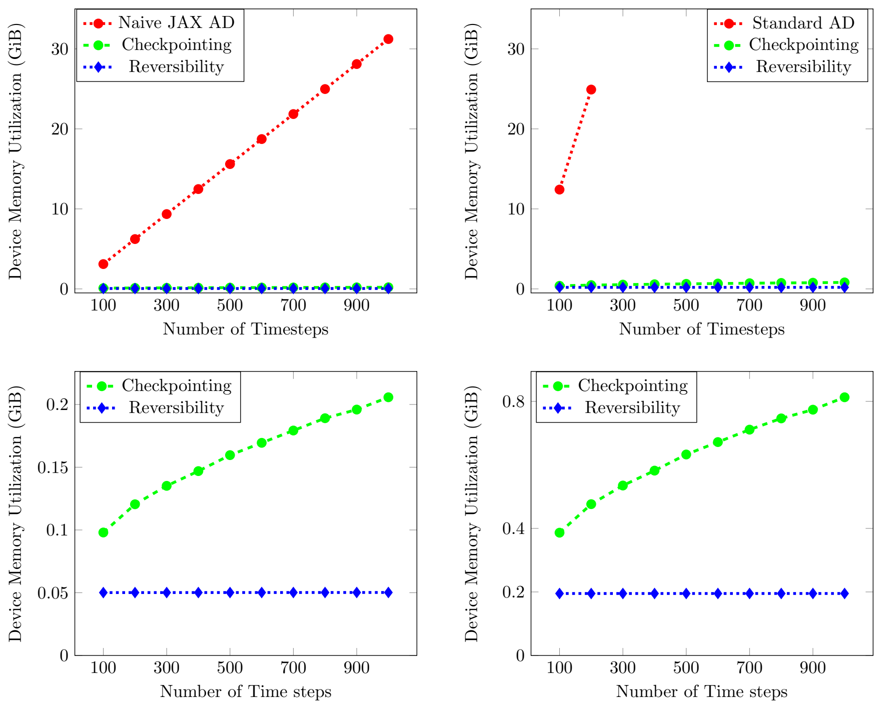

Fig. 5. Comparison of the device memory requirements for standard AD, periodic checkpointing, and periodic reversibility approaches with increasing number of time steps. The QOC simulation consisted of 8 (left) or 9 (right) qubits. The checkpoint period was chosen to be the square root of the number of time steps. Top row shows all three approaches; bottom row omits standard AD.

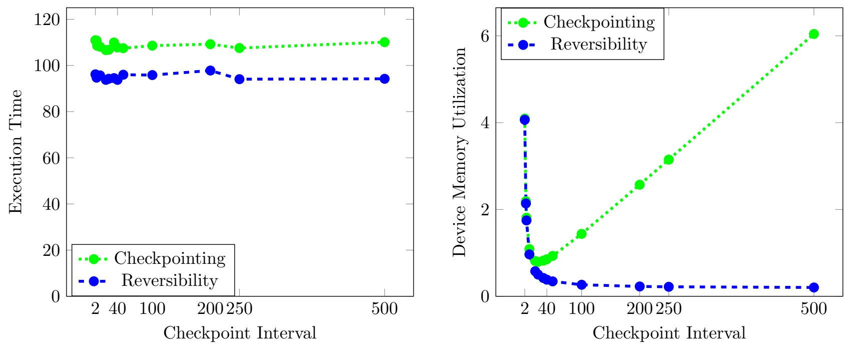

Fig. 6. Comparison of the execution time and device memory requirements for periodic checkpointing and checkpointing plus reversibility approaches with increasing number of time steps. The QOC simulation consisted of $1,000$ time steps and 9 qubits.

and K_{j+1}. Figure 4 is consistent with these expectations, although standard AD quickly runs out of memory for the case $q = 9$.

Based on the analysis in Section 3, we expect the memory requirements of standard AD to be linear in the number of time steps and the memory requirements of full reversibility to be independent of the number of time steps. We expect the memory requirements of periodic checkpointing to vary as a function of $\frac{N}{C} + C$; since C is chosen to be $C = \sqrt{N}$, the memory should vary as a function of $\sqrt{N}$. Figure 5 clearly shows the linear dependence of standard AD and independence of full reversibility on the number of time steps. The memory requirements for periodic checkpointing are also consistent with expectations.

5.3 Vary Checkpointing Period

We examined the dependence of execution time and memory requirements on the checkpointing period, C, keeping the number of qubits fixed at $q = 9$ and the number of time steps fixed at $N = 1,000$. We expect the time to be roughly independent of C because every U_j and K_j must be computed once during the forward pass and one more time on the reverse pass. We expect periodic checkpointing with reversibility to be somewhat faster than periodic checkpointing alone because periodic checkpointing must compute forward from the checkpoint, storing intermediate K_j along the way, while periodic checkpointing with reversibility skips the second forward pass and is able to restore K_j during the reverse pass directly from the controls and K_{j+1}. The timing results in Figure 6 (left) are consistent with these expectations.

We expect the memory requirements of periodic checkpointing with reversibility to vary as a function of $\frac{N}{C}$ or, since N is constant, as a function of $\frac{1}{C}$. We expect the memory requirements of periodic checkpointing alone to vary as a function of $\frac{N}{C} + C$, with a minimum at $C = \sqrt{N} \approx 32$. Again, the memory utilization results in Figure 6 (right) are consistent with these expectations.

6 Conclusion and Future Work

We have implemented a version of quantum optimal control (QOC) using the JAX framework. We have compared standard automatic differentiation (AD), periodic checkpointing, and reversibility—a nonstandard AD approach that recognizes that the inverse of a unitary matrix is its conjugate transpose. Checkpointing and reversibility are both superior to standard AD. The reversibility approach, however, allows more qubits to be simulated when the number of time steps is large. Recognizing that reversibility (Equation 15) is precise in real arithmetic but is not precise in floating-point arithmetic, we demonstrated that reversibility can be combined with periodic checkpointing, reducing memory requirements relative to periodic checkpointing alone while ensuring that roundoff errors are not accumulated over a period of more than C time steps.

In the future, we will study methods to estimate the amount of roundoff error as a function of C in order to choose a period that minimizes memory requirements while incurring acceptable roundoff errors. We will investigate applying lossy compression to the checkpoints and compare the trade-offs in storage and accuracy between periodic checkpointing with lossy compression and periodic checkpointing with reversibility. Moreover, we will combine periodic checkpointing, lossy compression, and reversibility to enable QOC to be applied to even larger numbers of qubits and time steps.

References

1. https://github.com/sriharikrishna/qoc (2022)
2. Aupy, G., Herrmann, J., Hovland, P., Robert, Y.: Optimal multistage algorithm for adjoint computation. SIAM Journal on Scientific Computing **38**(3), C232–C255 (2016)
3. Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M.: Automatic differentiation in machine learning: A survey. Journal of Machine Learning Research **18**(153), 1–43 (2018), http://jmlr.org/papers/v18/17-468.html
4. Beaumont, O., Herrmann, J., Pallez, G., Shilova, A.: Optimal memory-aware backpropagation of deep join networks. Philosophical Transactions of the Royal Society A **378**(2166), 20190049 (2020)
5. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., Zhang, Q.: JAX: composable transformations of Python+NumPy programs (2018), http://github.com/google/jax
6. Caneva, T., Calarco, T., Montangero, S.: Chopped random-basis quantum optimization. Phys. Rev. A **84**, 022326 (Aug 2011). https://doi.org/10.1103/PhysRevA.84.022326
7. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016)
8. Cyr, E.C., Shadid, J., Wildey, T.: Towards efficient backward-in-time adjoint computations using data compression techniques. Computer Methods in Applied Mechanics and Engineering **288**, 24–44 (2015)

9. Doria, P., Calarco, T., Montangero, S.: Optimal control technique for many-body quantum dynamics. Phys. Rev. Lett. **106**, 190501 (May 2011). https://doi.org/10.1103/PhysRevLett.106.190501
10. Griewank, A.: Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optimization Methods and software **1**(1), 35–54 (1992)
11. Griewank, A., Walther, A.: Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Trans. Math. Softw. **26**(1), 19–45 (mar 2000). https://doi.org/10.1145/347837.347846
12. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 105 in Other Titles in Applied Mathematics, SIAM, Philadelphia, PA, 2nd edn. (2008), http://bookstore.siam.org/ot105/
13. Hascoet, L., Pascual, V.: The Tapenade automatic differentiation tool: principles, model, and specification. ACM Transactions on Mathematical Software (TOMS) **39**(3), 1–43 (2013)
14. Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., Stoica, I.: Checkmate: Breaking the memory wall with optimal tensor rematerialization. Proceedings of Machine Learning and Systems **2**, 497–511 (2020)
15. Johansson, J., Nation, P., Nori, F.: QuTiP: An open-source python framework for the dynamics of open quantum systems. Computer Physics Communications **183**(8), 1760–1772 (2012). https://doi.org/10.1016/j.cpc.2012.02.021
16. Johansson, J., Nation, P., Nori, F.: QuTiP 2: A Python framework for the dynamics of open quantum systems. Computer Physics Communications **184**(4), 1234–1240 (2013). https://doi.org/10.1016/j.cpc.2012.11.019
17. Khaneja, N., Brockett, R., Glaser, S.J.: Time optimal control in spin systems. Phys. Rev. A **63**, 032308 (Feb 2001), 10.1103/PhysRevA.63.032308
18. Kubota, K.: A Fortran77 preprocessor for reverse mode automatic differentiation with recursive checkpointing. Optimization Methods and Software **10**(2), 319–335 (1998). https://doi.org/10.1080/10556789808805717
19. Kukreja, N., Hückelheim, J., Louboutin, M., Washbourne, J., Kelly, P.H., Gorman, G.J.: Lossy checkpoint compression in full waveform inversion. Geoscientific Model Development Discussions pp. 1–26 (2020)
20. Leung, N., Abdelhafez, M., Koch, J., Schuster, D.: Speedup for quantum optimal control from automatic differentiation based on graphics processing units. Phys. Rev. A **95**, 042318 (Apr 2017), 10.1103/PhysRevA.95.042318
21. Naumann, U.: The Art of Differentiating Computer Programs. Society for Industrial and Applied Mathematics (2011). https://doi.org/10.1137/1.9781611972078
22. Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., He, Y.: ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3458817.3476205
23. Schanen, M., Marin, O., Zhang, H., Anitescu, M.: Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver Nek5000. Procedia Comput. Sci. **80**(C), 1147—1158 (jun 2016). https://doi.org/10.1016/j.procs.2016.05.444