Lawrence Berkeley National Laboratory
LBL Publications

Title
GPU acceleration of non-equilibrium Green's function calculation using OpenACC and CUDA FORTRAN

Permalink
https://escholarship.org/uc/item/6zp847d9

Journal
The International Journal of High Performance Computing Applications

ISSN
1094-3420

Authors
Yin, Jia
Ibrahim, Khaled Z
Del Ben, Mauro
et al.

Publication Date
2026-07-08

DOI
10.1177/10943420261465935

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

GPU Acceleration of Non-equilibrium Green's Function Calculation using OpenACC and CUDA FORTRAN

Jia Yin*, Khaled Z. Ibrahim†, Mauro Del Ben†, Jack Deslippe‡, Yang-hao Chan§ and Chao Yang†
*School of Mathematical Sciences, Fudan University
Shanghai, China 200433
Email: jiayin@fudan.edu.cn (Jia Yin)
†Applied Mathematics & Computational Research Division (AMCRD), Lawrence Berkeley National Laboratory
Berkeley, California 94720
Email: cyang@lbl.gov (Chao Yang), kzibrahim@lbl.gov (Khaled Z. Ibrahim), mdelben@lbl.gov (Mauro Del Ben)
‡National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory
Berkeley, California 94720
Email: jrdeslippe@lbl.gov (Jack Deslippe)
§Institute of Atomic and Molecular Sciences, Academia Sinica
Taipei, Taiwan 10617
Email: yanghao@gate.sinica.edu.tw (Yang-hao Chan)

arXiv:2505.19467v1 [cs.DC] 26 May 2025

***Abstract*—The numerical solution of the Kadanoff-Baym nonlinear integro-differential equations, which yields the non-equilibrium Green's functions (NEGFs) of quantum many-body systems, poses significant computational challenges due to its high computational complexity. In this work, we present efficient implementations of a numerical method for solving these equations on distributed-memory architectures, including many-core CPUs and multi-GPU systems. For CPU-based platforms, we adopt a hybrid MPI/OpenMP programming model to exploit both inter-node and intra-node parallelism. On GPU-accelerated systems, we implement the method using two distinct approaches: MPI/OpenACC and MPI/CUDA FORTRAN. Several optimization strategies are employed to enhance GPU performance, including techniques to maximize computational resource utilization and minimize the overhead associated with kernel launches and memory management. Although OpenACC is easy to use, CUDA FORTRAN provides more advanced features for configuring and managing multiple levels of concurrency, while also simplifying memory allocation and data movement between host and device. This flexibility translates into significant performance improvements. We compare the performance of the three implementations and demonstrate that the GPU-based approaches achieve substantial speedups over CPU-based implementations. Furthermore, both CPU and GPU versions exhibit excellent strong and weak scaling, confirming the scalability and efficiency of our approach for large-scale NEGF computations.**

I. Introduction

Simulating the dynamics of a nonequilibrium quantum many-body system with a large number of particles N is computationally challenging due to the exponential scaling of the number of degrees of freedom in the many-body wavefunction with respect to N. However, many properties of a quantum many-body system can be ascertained by working with its observables such as the single-particle Green's function $G(r,t,r',t')$, which can be interepreted as the probability of detecting a particle at the position r and time t when a particle is created at position r' and time t'. For simplicity, we will leave out the spatial coordinates in G below. For systems that are out of equilibrium, the non-equilibrium Green's function (NEGF) is a two-time function $G(t,t')$ that satisfies an equation of motion known as the Kadanoff-Baym equation [1]–[4], which is a set of nonlinear integral differential equation defined on a contour $\mathcal{C}$ in the complex plane known as the Keldysh contour [5], [6].

The Kadanoff-Baym equations (KBEs) [3], [7] have the form

$$\begin{cases} \left[i\dfrac{d}{dt} - h(t)\right] G(t,t') = \delta(t,t') + \displaystyle\int_{\mathcal{C}} \Sigma(t,\bar{t})G(\bar{t},t')d\bar{t}, \\ \left[-i\dfrac{d}{dt'} - h(t')\right] G(t,t') = \delta(t,t') + \displaystyle\int_{\mathcal{C}} G(t,\bar{t})\Sigma(\bar{t},t')d\bar{t}, \end{cases} \tag{1}$$

where $h(t)$ is a single-particle Hamiltonian, and $\Sigma(t,t')$ is the self-energy that accounts for many-body interaction among different particles. The true analytical form of $\Sigma(t,t')$ is generally unknown, and is often approximated through many-body perturbation theory.

The numerical solution of (1) requires propagating $G(t,t')$ on a two-time grid. The time evolution procedure is computationally expensive due to the coupled nature of the equations, the nonlinearity of integro-differential equation, and the need to evaluate the integral term at each time step in addition to the self-energy evaluations. These self-energy terms are dependent on the Green's function, and their evaluation requires several tensor contractions.

Fortunately, multiple levels of concurrency exist in both the

self-energy and collision integral calculations. The algorithmic concurrency allows us to implement these calculations efficiently on many-core CPUs using the MPI+OpenMP programming model. However, performing a long-time NEGF dynamics simulation on a many-core CPU based distributed memory high performance computer can still take a prohibitively long wall clock time.

In recent years, there has been widespread availability of high-performance computers equipped with graphics processing unit (GPU) accelerators. GPUs possess significantly more hardware threads than CPUs and offer much higher memory bandwidth. They appear to be the natural computing platform for carrying out NGEF simulations.

However, there are some unique features of the GPUs that make it challenging to take full advantage of the computational resources available on such an architecture. To optimize the efficiency of numerical KBE solvers on multiple GPUs, several modifications of the MPI-OpenMP implementation of the KBE solver are required to maximize resource utilization and minimize computational overhead.

We discuss the changes we need to make in an MPI+OpenACC implementation of the self-energy and collision integral calculations in order to achieve satisfactory performance on NVIDIA GPUs. We show how performance profiling tools such as NVIDIA's Nsight Compute can be used to guide performance optimization. Although OpenACC is relatively easy to use, it has some limitation on resource allocation and utilization. We show that CUDA FORTRAN can provide more flexibility and allow us to fine tune the performance by managing how thread blocks and threads are configured and utilized to achieve multiple levels of concurrency as well as reducing memory allocation and host-device data transfer overhead.

The paper is organized as follows. In Section II, we describe the main computational tasks in the numerical solution of the Kadanoff-Baym equation and examine different levels of concurrency in main computational kernels. We discuss how the main computational tasks in a KBE solver can be parallelized on distributed memory many-core CPUs by using a hybrid MPI/OpenMP implementation and show that such an implementation has a good parallel scaling in Section III. In Sections IV, we discuss how OpenACC can be used to replace OpenMP and enable the KBE solver to run efficiently on NVIDIA GPUs. The advantage of using CUDA FORTRAN to improve the performance of the KBE solver on NVIDIA GPUs is shown in Section V. Several computational experiments are presented in Section VI to compare the performance of different programming models on CPUs and GPUs. We also show the strong and weak parallel scaling of the CPU and GPU implementation of the KBE solver in that section.

II. Computational Kernels

In this section, we will use a specific model problem to illustrate the main computational kernels used in a numerical time evolution scheme for solving the KBE (1). The model problem is a so-called two-band Hubbard model [8]–[12]. The periodic boundary condition for the N_0-site hopping term allows us to define the Hamiltonian

$$H_{\rm total}(t) = H_{\rm s} + H_{\rm ext}(t), \tag{2}$$

in the momentum space. In particular, the system Hamiltonian $H_{\rm s}$ can be written as

$$\begin{aligned} H_{\rm s} = &\sum_{\mathbf{k}} (\epsilon_{v\mathbf{k}} c^\dagger_{v\mathbf{k}} c_{v\mathbf{k}} + \epsilon_{c\mathbf{k}} c^\dagger_{c\mathbf{k}} c_{c\mathbf{k}}) - U(t) \sum_{\mathbf{k}} c^\dagger_{c\mathbf{k}} c_{c\mathbf{k}} \\ &+ \frac{U(t)}{N} \sum_{\mathbf{k}_1,\mathbf{k}_2,\mathbf{q}} c^\dagger_{v\mathbf{k}_1+\mathbf{q}} c^\dagger_{c\mathbf{k}_2-\mathbf{q}} c_{c\mathbf{k}_2} c_{v\mathbf{k}_1}, \end{aligned} \tag{3}$$

where $\mathbf{k}$ denotes k-points sampled from the Brillouin zone associated with the N-site lattice, $c^\dagger_{v\mathbf{k}}$ and $c_{v\mathbf{k}}$ are creation and annihilation operators associated with the valence band v, $c^\dagger_{c\mathbf{k}}$ and $c_{c\mathbf{k}}$ are creation and annihilation operators associated with the conduction band c, $\epsilon_{v\mathbf{k}}$ and $\epsilon_{c\mathbf{k}}$ are the band energies of the valence and conduction bands with momentum $\mathbf{k}$ respectively, $U(t)$ is the time-dependent on-site interaction between the two bands. The external field $H_{\rm ext}(t)$ in (2) is the light-matter coupling within the dipole approximation defined by

$$H_{\rm ext}(t) = E(t) \sum_{\mathbf{k}} (d_{\mathbf{k}} c^\dagger_{c\mathbf{k}} c_{v\mathbf{k}} + d^*_{\mathbf{k}} c^\dagger_{v\mathbf{k}} c_{c\mathbf{k}}), \tag{4}$$

where $E(t) = I\delta(t-0.5)$ is a time-dependent pulse centered at $t = 0.5$ with intensity I, and $d_{\mathbf{k}}$ is the dipole matrix element. For simplicity we set $d_{\mathbf{k}} = 1$.

The KBE associated with such a model problem can be written as

$$\begin{cases} \left[i\frac{d}{dt} - h(\mathbf{k};t) \right] G^<_{jm}(\mathbf{k};t,t') = \delta(t,t') + I^<_{jm}(\mathbf{k};t,t'), \\ \left[-i\frac{d}{dt'} - h(\mathbf{k};t') \right] G^>_{jm}(\mathbf{k};t,t') = \delta(t,t') + I^>_{jm}(\mathbf{k};t,t'), \end{cases} \tag{5}$$

where the $\mathbf{k}$-dependent $G^{\lessgtr}_{jm}(\mathbf{k};t,t')$ represents the lesser/-greater Green's function with j and m being the band indices, and $I^{\lessgtr}_{jm}(\mathbf{k};t,t')$ is known as a collision integral defined as

$$\begin{aligned} I^<_{jm}(\mathbf{k};t,t') = &\int_0^t d\bar{t} \left[(G^>(\mathbf{k};t,\bar{t}) - G^<(\mathbf{k};t,\bar{t})) \Sigma^<(\mathbf{k};\bar{t},t') \right]_{jm} \\ &+ \int_0^t d\bar{t} \left[G^<(\mathbf{k};t,\bar{t}) (\Sigma^<(\mathbf{k};\bar{t},t') - \Sigma^>(\mathbf{k};\bar{t},t')) \right]_{jm}, \end{aligned} \tag{6}$$

with a similar expression for $I^>_{jm}(\mathbf{k};t,t')$.

Although the integro-differential equations (5) may appear to be completely independent for different $\mathbf{k}$'s, they are actually coupled through the self-energy terms $\Sigma^{\lessgtr}_{jm}(\mathbf{k},t,t')$.

When the second-Born approximation is used to define the self-energy terms, we can write $\Sigma^<_{jm}(\mathbf{k};t,t')$ as

$$\begin{aligned} &\Sigma^<_{jm}(\mathbf{k};t,t') \\ =&\frac{U(t)U(t')}{n_k^2} \sum_{\mathbf{q}\mathbf{k}'} G^<_{j\bar{m}}(\mathbf{k}'+\mathbf{q};t,t') G^>_{\bar{m}\bar{j}}(\mathbf{k}';t',t) G^<_{jm}(\mathbf{k}-\mathbf{q};t,t') \\ &- \frac{U(t)U(t')}{n_k^2} \sum_{\mathbf{q}\mathbf{k}'} G^<_{j\bar{m}}(\mathbf{k}';t,t') G^>_{\bar{m}\bar{j}}(\mathbf{k}'+\mathbf{q}-\mathbf{k};t',t) G^<_{jm}(\mathbf{q};t,t'). \end{aligned} \tag{7}$$

Similar expressions can be written for $\Sigma^>(\mathbf{k};\bar{t},t)$.

The numerical method we use to solve (5) advances $G^{\lessgtr}(t,t')$ along both the t and the t' directions. If we let

$(t_i, t'_j) \equiv (i\Delta t, j\Delta t)$, be a grid point on a uniform two-time grid, the numerical propagation of $G^<$ according to the first equation in (1) is described in Algorithm 1.

Algorithm 1 The propagation of $G^<$ at the n_t-th time step

Input: $G^{\lessgtr}(t_j, t'_\ell)$, $j, \ell = 0, ..., n_t - 1$, max, ε
Output: $G^<(t_{n_t}, t'_\ell)$, $\ell = 0, ..., n_t$

for $\ell \leftarrow 0$ to n_t **do**
  Evaluate $h(t_{n_t-1} + \Delta t/2)$
  Evaluate $\Sigma^{\lessgtr}(t_j, t'_{n_t-1})$ and $\Sigma^{\lessgtr}(t_{n_t-1}, t'_\ell)$ by (7)
  Evaluate $I^<(t_{n_t-1}, t'_\ell)$ by (6)
  Solve (5) to get $G^<(t_{n_t}, t'_\ell)$
end for
for $i \leftarrow 1$ to max **do**
  for $\ell \leftarrow 0$ to n_t **do**
    $G_0^<(t_{n_t}, t'_\ell) \leftarrow G^<(t_{n_t}, t'_\ell)$
    Evaluate $h(t_{n_t})$
    Evaluate $\Sigma^{\lessgtr}(t_j, t_{n_t})$ and $\Sigma^{\lessgtr}(t_{n_t}, t'_\ell)$ by (7)
    Evaluate $I^<(t_{n_t}, t'_\ell)$ by (6)
    $I_1^<(t_{n_t-1}, t'_\ell) \leftarrow \frac{1}{2}\left(I^<(t_{n_t-1}, t'_\ell) + I^<(t_{n_t}, t'_\ell)\right)$
    Solve (5) with $I_1^<$ to get $G^<(t_{n_t}, t'_\ell)$
  end for
  if $\|G^<(t_{n_t}, \cdot) - G_0^<(t_{n_t}, \cdot)\| \le \varepsilon$ **then**
    break
  end if
end for

At the n_t-th time step, when advancing from $G^<(t_{n_t-1}, t'_\ell)$ to $G^<(t_{n_t}, t'_\ell)$, $\ell = 0, .., n_t - 1$ we need to

- Evaluate the single-particle Hamiltonian $h(t_{n_{t_1}} + \Delta t/2)$;
- Evaluate the self-energies $\Sigma^{\lessgtr}(t_j, t'_{n_t-1})$ and $\Sigma^{\lessgtr}(t_{n_t-1}, t'_\ell)$ defined by (7) with $j, \ell = 0, ...n_t - 1$, i.e., the self energies on the orange time grid points in Fig. 1;
- Evaluate the collision integral $I^<(t_{n_t-1}, t'_\ell)$ defined by (6), $\ell = 0, ..., n_t - 1$, i.e. the collision integrals along the n_tth column of the two time grid points (marked in orange in Fig. 1;
- Numerically evolve $G^<(t_{n_t-1}, t_\ell)$ to $G^<(t_{n_t}, t_\ell)$, $\ell = 0, ..., n_t - 1$ using an appropriate time stepping scheme [13]–[17]. Because (5) is nonlinear, a non-linear iteration is often performed to ensure the consistency of G and the right-hand side of (5). To reach self-consistency, we need to repeatedly evaluate the single-particle Hamiltonian $h(t_{n_t})$, the self-energies $\Sigma^{\lessgtr}(t_j, t'_{n_t})$, $\Sigma^{\lessgtr}(t_{n_t}, t'_\ell)$ and the collision integral $I^<(t_{n_t}, t'_\ell)$ with $j, \ell = 0, ...n_t$, i.e., self-energies and collision integrals along the yellow time grid points marked in Fig. 1.

A similar set of operations are performed to advance $G^>(t_j, t_{n_t-1})$ to $G^>(t_j, t_{n_t})$ with $j = 0, ..., n_t$.

Note that the self-energy $\Sigma^<$ defined in (7) is the difference of two terms. We denote the first and second terms by $\Sigma^1_{jm}(\mathbf{k}; t, t')$ and $\Sigma^2_{jm}(\mathbf{k}; t, t')$ respectively.

We compute $\Sigma^1_{jm}(\mathbf{k}; t, t')$ in two steps. In the first step, we compute the polarizability $P_{jm}(\mathbf{q}; t, t')$

$$P_{jm}(\mathbf{q}; t, t') = \sum_{\mathbf{k}'} G^<_{jm}(\mathbf{k}' + \mathbf{q}; t, t') G^>_{mj}(\mathbf{k}'; t', t), \qquad (8)$$

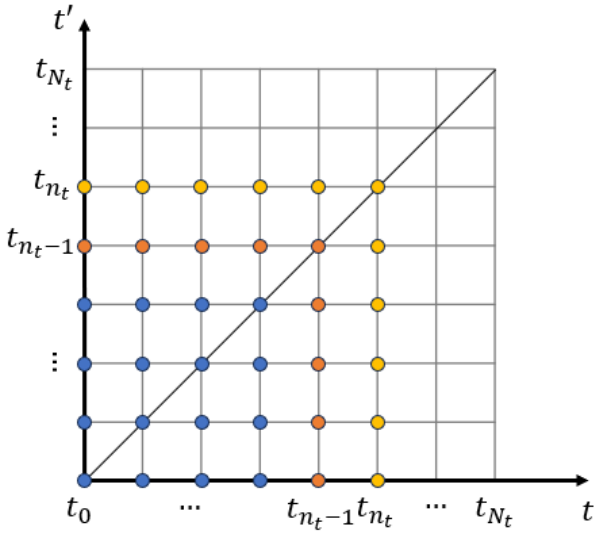


Fig. 1. Illustration of the evaluation at the n_t-th time step for $\Sigma^{\lessgtr}$ and $I^{\lessgtr}$. The blue dots represent values computed at previous time steps, and the orange dots give the $2n_t - 1$ pairs of (t, t') at which the values of $\Sigma^{\lessgtr}$ and $I^{\lessgtr}$ are computed at the n_t-th time step.

This step contracts $G^<_{jm}$ and $G^>_{mj}$ over $\mathbf{k}'$. The complexity for such a contraction is $\mathcal{O}(n_k)$. Since we need to perform the contraction for all k-points, for given j, m, t, t', the complexity to compute $P_{jm}(\mathbf{q}; t, t')$ is $\mathcal{O}(n_k^2)$.

In the second step, we contract $P_{\bar{j}\bar{m}}$ obtained above with $G^<_{jm}$ over $\mathbf{q}$ to get $\Sigma^1_{jm}(\mathbf{k}; t, t')$

$$\Sigma^1_{jm}(\mathbf{k}; t, t') = U(t)U(t') \sum_{\mathbf{q}} P_{\bar{j}\bar{m}}(\mathbf{q}; t, t') G^<_{jm}(\mathbf{k} - \mathbf{q}; t, t'). \quad (9)$$

This contraction also costs $\mathcal{O}(n_k)$ for one k-point, and $\mathcal{O}(n_k^2)$ considering all k-points.

By breaking the contractions over $\mathbf{k}'$ and $\mathbf{q}$ required in (7) into two steps, we reduce the k-point complexity from $\mathcal{O}(n_k^3)$ to $\mathcal{O}(n_k^2)$.

However, such a reduction in complexity is not applicable to $\Sigma^2_{jm}(\mathbf{k}; t, t')$, i.e., the contractions over $\mathbf{k}'$ and $\mathbf{q}$ cannot be decoupled. As a result, the evaluation of $\Sigma^2_{jm}(\mathbf{k}; t, t')$ takes $\mathcal{O}(n_k^3)$ operations. The implementation of such a calculation follows the nested loops shown in the pseudocode given in Algorithm 2. We omit the time arguments t and t' in the pseudocode for clarity.

The collision integral (6) can be evaluated as the sum of two terms $I^1_{jm}(\mathbf{k}; t, t')$ and $I^2_{jm}(\mathbf{k}; t, t')$ respectively.

Since the evaluations of I^1 and I^2 are similar once we have $G^{\lessgtr}$ and $\Sigma^{\lessgtr}$ computed and stored, we will only discuss how I^2 is computed below.

The main computation performed in the evaluation of I^2 is the convolution of $G^<(\mathbf{k}, t, \bar{t})$ and $\Sigma^<(\mathbf{k}, \bar{t}, t') - \Sigma^>(\mathbf{k}, \bar{t}, t')$. Using a standard second order numerical integration scheme to compute such a convolution requires $\mathcal{O}(n_t)$ floating point operations at the n_t-th time step for each k-point.

Algorithm 1 shows that we need to perform at least n_t self-energy and collision integral calculation in order to

Algorithm 2 The evaluation of $\Sigma^2(\mathbf{k})$

$\Sigma^2(\mathbf{k}) \leftarrow 0$
for $k \leftarrow 1$ to n_k **do**
  for $j \leftarrow 1$ to 2 **do**
    for $m \leftarrow 1$ to 2 **do**
      for $q \leftarrow 1$ to n_k **do**
        for $k' \leftarrow 1$ to n_k **do**
          $\Sigma^2_{jm}(k) \leftarrow \Sigma^2_{jm}(k) + G^{<}_{j\bar{m}}(k') G^{>}_{\bar{m}\bar{j}}(k'+q-k) G^{<}_{\bar{j}m}(q)$
        end for
      end for
    end for
  end for
end for

obtain approximations to $G^{\lessgtr}(t_j, t'_{n_t})$ and $G^{\lessgtr}(t_{n_t}, t'_j)$, for $j = 1, 2, ..., n_t$. As a result, the overall complexities for the self-energy and the collision integral computation become $\mathcal{O}(n_k^3 n_t)$ and $\mathcal{O}(n_k n_t^2)$ respectively. Once these quantities become available, evolving $G^{\lessgtr}$ from (t_j, t'_{n_t-1}) to (t_j, t'_{n_t}) and from (t_{n_t-1}, t'_j) to (t_{n_t}, t'_j) takes only $\mathcal{O}(n_k n_t)$ operations, which is considerably lower. Therefore, when n_t is small, the time stepping cost is dominated by the self-energy computation. The collision integral evaluation only becomes more expensive when n_t becomes much larger than n_k. If we evolve the equation for a large N_t time steps, the overall computational cost will be $\mathcal{O}(n_k^3 N_t^3)$. This cost increases rapidly with increasing n_k or N_t. A simple implementation shows that for $n_k = 1024$, performing 10 time steps sequentially takes more than 1 day to compute on a modern CPU, which is prohibitively expensive for solving problems of interest.

III. MPI/OpenMP implementation

The separation of the Kadanoff-Baym equation (5) by k-points for the two-band Hubbard model as shown in (7) naturally suggests that we can distribute the time evolution of $G^{\lessgtr}_{jm}(\mathbf{k}, t, t')$ among different MPI ranks by k-points. Note that the single-particle Hamiltonian h is replicated on all MPI ranks, the collision integral $I^{\lessgtr}$, which is distributed among different MPI ranks by k-points, and the update of $G^{\lessgtr}$ can all be computed independently for different $\mathbf{k}$'s. The evaluations of the self-energy terms $\Sigma^{\lessgtr}$, which require contractions of $G^{\lessgtr}$ over all k-points, can be distributed by k-points also. In particular, we can distribute the outermost loop in Algorithm 2 among n_p MPI ranks, with each MPI rank loop over $\mathrm{my_nk} = n_k/n_p$ k-points (assuming n_p divides n_k.) Collective communications (`MPI_Allreduce`) are required to replicate the Green's functions on all MPI ranks so that we can use the complete Green's function to compute distributed (local) self-energy tensor simultaneously. The main steps of the KBE solver are shown in Fig. 2. In each time step, we evaluate the single particle Hamiltonian by calling the subroutine `get_h`. This is followed by calling self-energy and collision integral evaluation subroutines `get_SE_at_t` and `get_CI_at_t`. A time evolution method is then used in `time_step` to advance the Green's function to the next time slices on the two-time grid.

```
do it=1,nt
  ! compute the single-particle Hamiltonian
  call get_h(ham,...)
  ! compute the self-energy
  call get_SE_at_t(se,...)
  ! compute the collision integral
  call get_CI_at_t(ci,...)
  ! evolve the Green's function
  call time_step(Gg,Gl,...)
  ! MPI reduce
  call MPI_Allreduce(Gg,gt(...,it+1,:it),size(Gg),&
              MPI_COMPLEX_DPC,MPI_SUM,...)
  call MPI_Allreduce(Gl,gt(...,:it+1,it+1),size(Gl),&
              MPI_COMPLEX_DPC,MPI_SUM,...)
enddo
```

Fig. 2. The main steps in the outermost loop of the KBE solver.

We use CrayPat to analyze the performance of the MPI implementation. In a run that propagates the Green's function for 10 time steps on 1024 MPI ranks with $n_k = 1024$ k-points, the profiler shows that the self-energy computation takes over 95% of the total time. This observation is consistent with our previous cost analysis that suggests self energy evaluate would dominate the entire computation when $n_t \ll n_k$. Therefore, in the following, we focus first on investigating the performance of the self-energy computation.

In Fig. 3, we show a code snippet for computing $\Sigma^2_{jm}(\mathbf{k}, t, t')$ at a fixed (t, t') pair in parallel among n_p MPI ranks. We recall that the Green's functions are updated locally and stored globally. Arrays `Gtmp1` and `Gtmp2` in the code snippet store the Green's functions at all n_k k-points.

```
! loop over local k-points
do kk=1,my_nk
  ! map local k-point indices to global indices
  ikk = bz%my_kpts(kk)
  do jj=1,dim
    do ii=1,dim
      do kp=1,nk
        do kpp=1,nk
          iqk = bz%allind_kpk(kp,kpp)
          iqk = bz%allind_kmk(iqk,ikk)
          se2(kk,ii,jj) = se2(kk,ii,jj)+Ut*Utp*&
                  Gtmp1(kp,ii,dim-jj+1)*&
                  Gtmp2(iqk,dim-jj+1,dim-ii+1)*&
                  Gtmp1(kpp,dim-ii+1,jj)
        enddo
      enddo
    enddo
  enddo
enddo
```

Fig. 3. The MPI implementation of $\Sigma^2_{jm}(\mathbf{k}; t, t')$ computation at a given (t, t') pair.

The nested loop structure shown in Algorithm 2 and Fig. 3 indicates that within each MPI rank, additional levels of concurrency for the self-energy evaluation can be exploited by using OpenMP.

Because the number of bands (2) in the model Hamiltonian is far fewer than the number of threads within each MPI rank (on a manycore compute node) compared to the number of k-points, we reorder the nested loops in Fig. 3 to

move the kp and kpp loops out and keep the jj and ii loops as the inner loops. An !OMP PARALLEL DO directive is placed right before the kp loop to execute the outer loop in parallel. A local private temporary array se2tmp is created to accumulate the intermediate contraction results. A reduction is performed through the use of the directive REDUCTION(+:se2tmp) to sum up contributions from all threads. The hybrid MPI/OpenMP implementation of the $\Sigma^2_{jm}(\mathbf{k}, t, t')$ calculation is shown in Fig. 4.

```
do kk=1,my_nk
  ikk = bz%my_kpts(kk)
  !$OMP PARALLEL DO PRIVATE(iqk) REDUCTION(+:se2tmp)
  do kp=1,nk
    do kpp=1,nk
      iqk = bz%allind_kpk(kp,kpp)
      iqk = bz%allind_kmk(iqk,ikk)
      do jj=1,dim
        do ii=1,dim
          se2tmp(ii,jj,kk) = se2tmp(ii,jj,kk)+&
                  Ut*Utp*Gtmp1(kp,ii,dim-jj+1)*&
                  Gtmp2(iqk,dim-jj+1,dim-ii+1)*&
                  Gtmp1(kpp,dim-ii+1,jj)
        enddo
      enddo
    enddo
  enddo
  !$OMP END PARALLEL DO
  se2(kk,:,:) = se2tmp(:,:,kk)
enddo
```

Fig. 4. The hybrid MPI/OPenMP implementation of the $\Sigma^2_{jm}(\mathbf{k}; t, t')$ computation at a given (t, t') pair.

Fig. 5 shows how the non-equilibrium Green's function computation scales with respect to the number of MPI ranks. This calculation is performed on the Perlmutter CPU nodes maintained at National Energy Research Scientific Computing (NERSC) Center. Each CPU node is composed of 2 AMD EPYC 7763 (Milan) CPUs with 64 cores per CPU, and 2 threads per core. In total, there is 512 GB DDR4 memory for each node, and for each CPU, the memory bandwidth is 204.8 GB/s. In this experiment, we set the number of k-points to $n_k = 1024$, the timing measurement for the self-energy and the collision integral calculation is collected for the first 10 time steps.

We observe that the wall clock time used decreases almost linearly with respect to the number of MPI ranks, which indicates that the overhead incurred in MPI_Allreduce is relatively small.

In Fig. 5, we also show how the hybrid MPI/OpenMP version of the program scales with respect to MPI ranks. In these calculations, we use 64 threads within each MPI rank, and each MPI rank is bind to a compute node. We observe a nearly perfect strong scaling when the number of MPI ranks is less than 128. Increasing the number of MPI ranks beyond 128 results in suboptimal strong scaling due to the OpenMP reduction overhead relative to the amount of work per MPI rank, which decreases as the number of MPI ranks increases.

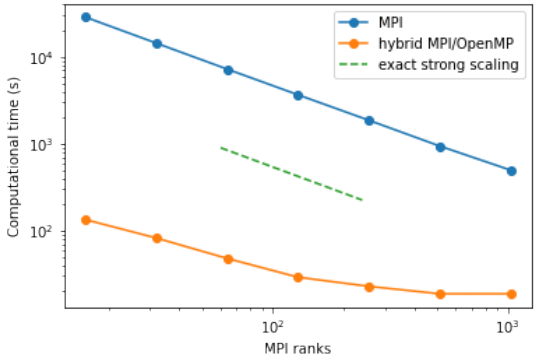


Fig. 5. The performance of the MPI and the hybrid MPI/OpenMP implementation to the program with varying number of MPI ranks. In the test problem, $n_k = 1024$, and we compute the first 10 time steps.

IV. GPU Implementation with OpenACC

Although the MPI implementation exhibits nearly perfect strong scaling, Fig. 5 shows that wall clock time required to evolve the Green's function for 10 time steps already exceeds 8 minutes even with 1024 MPI ranks. As mentioned before, the calculation of the self-energy and the collision integral takes over 95% of the total computational time. The use of hybrid MPI/OpenMP parallelization allows us to reduce the computation time by a factor of 26 on 1024 MPI ranks with 64 threads per MPI rank. However, when the number of time steps increases, the self-energy and collision integral evaluation will become progressively more time-consuming. To obtain a trajectory with $N_t = 500$ time steps, the hybrid MPI/OpenMP implementation uses more than 3 hours to compute the self-energy and about 2 hours to evaluate the collision integral.

To further improve the throughput of NEGF computation, we investigated the possibility of running the KBE solver efficiently on high performance computers equipped with accelerators. In this work, we focus on enabling the KBE solver to run efficiently on NVIDIA graphics processing units (GPUs).

Compared to CPUs, GPUs have a large number of streaming multiprocessors (SMs) with multiple cores each and thousands of threads organized into thread blocks and warps. The Single Instruction, Multiple Thread (SIMT) execution model and the availability of a large amount of high bandwdith shared memory and local cache makes GPUs ideal for computations that have high arithmetic intensity and regular memory access patterns.

To exploit the tremendous amount of computational resources available on GPUs, we need to design the solver to map concurrent tasks performed in the KBE solver within an MPI rank to different threads or thread blocks excecuted simultaneously on a GPU device. On NVIDIA GPUs, the easiest programming model that allows us to achieve such a mapping is the OpenACC directive-based programming [18], [19], which is an open source parallel programming standard.

Although a GPU node has much higher peak performance than a CPU node, there exist a few challenges that can hinder the optimal execution of code on a GPU. In particular, repeated memory allocations on the GPU device and excessive data movement between the host CPU and the GPU device can introduce a significant amount of overhead. The execution of a computational task on the GPU requires a kernel launch. A large number of repeated kernel launches can also degrade the performance of the overall computation. Furthermore, a sub-optimal memory access pattern that results in a memory bank conflict can also contribute to sub-optimal performance of the code.

In this section, we describe how to overcome these challenges when OpenACC directives are used to offload the Σ^2 and I^2 calculations onto NVIDIA GPUs. We can use OpenACC directives to manage memory access and map computational work to multiple thread blocks and threads to achieve optimal utilization of GPU resources.

We will demonstrate various performance improvement resulting from the use of OpenACC directives through a test problem that uses $n_k = 1024$, $n_t = 500$. The problem is solved on 4 MPI ranks.

A. *Exploiting GPU concurrency by loop fusion and reordering*

OpenACC provides a `parallel loop` directive similar to the `parallel do` directive in OpenMP. Therefore, we can simply replace the OpenMP directive in Fig. 4 with the OpenACC directive to enable the self-energy calculation to be performed on a GPU. However, because the number of threads on a GPU far exceeds that on a CPU, one cannot effectively utilize all threads if `nk` is not sufficiently large. To address this issue, we use the `collapse` clause in OpenACC to fuse the `kp` and `kpp` loops as shown in Fig. 6 to create more concurrent work that can be mapped to a large number of threads and executed simultaneously. A reduction operation is performed to sum up contributions from different threads into a scalar temporary variable `tmp` because OpenACC does not support reduction onto an element of an array directly. This limitation also requires us to move the `jj` and `ii` loops outside of the `kp` and `kpp` loops so that reduction is performed over `kp` and `kpp` for each (`ii`, `jj`) pair. Such a simple modification significantly improves the performance of the self-energy calculation on a GPU and reduces the wallclock time used to complete this part of calculation for the test problem by 86.78% compared to the baseline implementation without using `collapse`.

Another way to leverage the GPU architecture is to exploit multiple levels of concurrency that exists in the nested loops in Algorithm 2. This can be achieved by using the `loop gang` and `loop vector` directives provided in OpenACC to leverage the thread block, warp and thread hierarchy on NVIDIA GPUs. The code snippet displayed in Fig. 7 shows where the `loop gang` and `loop vector` directives are placed. Both the outer loops (at `kk`, `jj` and `ii` levels) and the inner loops (at `kp` and `kpp` levels) are automatically fused by the compiler to maximize thread block and thread utilization.

```
do kk=1,my_nk
  ikk = bz%my_kpts(kk)
  do jj=1,dim
    do ii=1,dim
      tmp = (0d0,0d0)
!$acc parallel loop reduction(+:tmp) collapse(2)
      do kp=1,nk
        do kpp=1,nk
          iqk = bz%allind_kpk(kp,kpp)
          iqk = bz%allind_kmk(iqk,ikk)
          tmp = tmp+Ut*Utp*Gtmp1(kp,ii,dim-jj+1)&
              * Gtmp2(iqk,dim-jj+1,dim-ii+1)&
              * Gtmp1(kpp,dim-ii+1,jj)
        enddo
      enddo
!$acc end parallel loop
      se2(kk,ii,jj) = tmp
    enddo
  enddo
enddo
```

Fig. 6. Using OpenACC `parallel loop` directive and loop fusion to accelerate $\Sigma^2_{jm}(\mathbf{k}; t, t')$ computation for a (t, t') pair on a GPU.

This modification results in 85.32% performance improve compared to the baseline implementation. The slightly smaller performance improvement compared to simply using the `collapse` directive to fuse the loops over `kp` and `kpp` is likely due to the small range of `my_nk`, `jj` and `ii` values that limit the number of thread blocks that can be fully utilized.

```
!$acc kernels loop gang present(se2)
do kk=1,my_nk
  do jj=1,dim
    do ii=1,dim
      ikk = bz%my_kpts(kk)
      tmp = (0d0,0d0)
      !$acc loop vector reduction(+:tmp)
      do kp=1,nk
        do kpp=1,nk
          iqk = bz%allind_kpk(kp,kpp)
          iqk = bz%allind_kmk(iqk,ikk)
          tmp = tmp+Ut*Utp*Gtmp1(kp,ii,dim-jj+1)*&
                Gtmp2(iqk,dim-jj+1,dim-ii+1)*&
                Gtmp1(kpp,dim-ii+1,jj)
        enddo
      enddo
      !$acc end loop
      se2(kk,ii,jj) = tmp
    enddo
  enddo
enddo
!$acc end kernels
```

Fig. 7. The OpenACC implementation for computing the $\Sigma^2_{jm}(\mathbf{k}; t, t')$ (saved in `se2`) at a given (t, t') pair. Here the `ikk` assignment is placed inside `ii` loop to ensure the `kk`, `jj` and `ii` loops can be fused.

B. *Updating self-energy using a single kernel launch*

In a naive implementation of Algorithm 1 that loops over the j and ℓ indices to update $\Sigma^<(t_j, t'_{n_t-1})$ and $\Sigma^>(t_{n_t-1}, t'_\ell)$, multiple kernel launches are issued as we execute the code shown in Fig. 7, which is contained in a subroutine named `get_2B_SE`. Fig. 8 shows how `get_2B_SE` is called repeatedly in an outer loop over t_l and t'_k.

Because each `itp` iterate in Fig 8 is completely independent from each other, we can exploit this level of concurrency by moving the `do itp=1,nt-1` loop inside the `!$acc kernels loop gang` directive within the `get_2B_SE` subroutine which we rename to `get_2B_SE_kernel`. A snippet of the restructured subroutine is shown in Fig. 9. In this subroutine, the `kernels loop gang` is placed right before the outermost `it` loop to launch the kernel once. Several temporary arrays such as `Gtmp1` and `Gtmp2` as well as input arguements such as `Ut` and `Utp` are copied from the host to the device before the kernel is launched. Once the self-energy calculation is completed on the device, the result is updated on the host at the end of the subroutine.

Fig. 10 shows how this subroutine is called. Note that instead of passing a single time snapshot of the `se` and `G` arrays, we pass the entire time slice of these arrays in either t or t' up to `nt-1` time steps. These time slices are shown as the red dots in Fig. 1.

```
do itp=1,nt-1
  ! compute SE_lesser
  call get_2B_SE(se(:,:,:,itp,nt), bz, &
          G(:,:,:,itp,nt), G(:,:,:,nt,itp), Ut, Utp)
  ! compute SE_greater
  call get_2B_SE(se(:,:,:,nt,itp), bz, &
          G(:,:,:,nt,itp), G(:,:,:,itp,nt), Utp, Ut)
enddo
```

Fig. 8. The main loop for computing the self-energy.

A similar code restructuring is used to implement a version of collision integral computation that uses a single kernel launch. We omit the corresponding code snippets here for brevity. This approach allows us to achieve high concurrency and considerably reduces the amount of time spent in the self-energy and collision integral calculations. For example, using this restructured code results in a 90.17% performance improvement in the self-energy calculation compared with the version that implements only the loop reordering shown in Fig. 7 for the model problem described in Section II.

C. Reducing memory allocation and data movement overhead

There are several allocatable arrays used to hold intermediate results in the self-energy and the collision integral calculations. For example, in the MPI/OpenMPI version of the self-energy computation, the temporary array `se2tmp` that appears in Fig. 4 is used to hold intermediate self-energy contraction results produced by each thread. This array is only used inside the subroutine which computes $\Sigma^<$ at a given (t, t') pair. Therefore, it is allocated and deallocated within such a subroutine. The repeated allocation and deallocation does not introduce much overhead on CPUs. However, such overhead cannot be ignored on GPUs.

To reduce this overhead, we modified the code to allocate these arrays only once on the host, prior to calling the self-energy calculation subroutine. Once allocated, these arrays are immediately copied to the GPU device. There is no need for repeated allocation and data transfer inside the loop over t or t'.

```
! initialize P,se1,se2 to 0
...
!$acc data copyin(Gtmp1, Gtmp2, Ut, Utp)
!$acc update device(P,se1,se2)

!$acc kernels loop present(P)
...
!$acc end kernels

!$acc kernels loop present(P,se1)
...
!$acc end kernels

!$acc kernels loop gang present(se2)
do it=1,nt-1
  do jj=1,dim
    do ii=1,dim
      do kk=1,my_nk
        ikk = start_kpts+kk-1
        tmp = (0d0,0d0)
        !$acc loop vector reduction(+:tmp)
        do kp=1,nk
          do kpp=1,nk
            ! compute the k-point index on the fly
            ...
            tmp = tmp+Ut(it)*Utp(it)*&
                  Gtmp1(kp,ii,dim-jj+1,it)*&
                  Gtmp2(iqk,dim-jj+1,dim-ii+1,it)*&
                  Gtmp1(kpp,dim-ii+1,jj,it)
          enddo
        enddo
        !$acc end loop
        se2(kk,ii,jj,it) = tmp
      enddo
    enddo
  enddo
enddo
!$acc end kernels

!$acc update self(se1,se2)
!$acc end data
```

Fig. 9. Kernel launches inside the subroutine `get_2B_SE_kernel` which contains an OpenACC implementation of the self-energy computation at all time locations with a single kernel for each step.

```
!$acc enter data copyin(P,se1,se2)
call get_2B_SE_kernel(se(:,:,:,1:nt-1,nt), my_nk,&
      G(:,:,:,1:nt-1,nt), G(:,:,:,nt,1:nt-1),&
      U_t2(1:nt-1), U_t1(1:nt-1), P, se1, se2)
call get_2B_SE_kernel(se(:,:,:,nt,1:nt-1), my_nk,&
      G(:,:,:,nt,1:nt-1), G(:,:,:,1:nt-1,nt),&
      U_t1(1:nt-1), U_t2(1:nt-1), P, se1, se2)
!$acc exit data delete(P,se1,se2)
```

Fig. 10. The calls to the self-energy computation subroutine with the code restructure.

The self-energy computational kernels are launched inside the subroutine `get_2B_SE_kernel` as shown in Fig. 9. We use `!$acc update` to update these arrays either on the host or on the device outside the kernels. The memory allocated for these arrays is released after we exit from the main loop. This memory management scheme is much faster than frequent data allocation, transfer and deallocation.

Moreover, since we use `Gtmp1` and `Gtmp2` arrays repeatedly within the kernels launched to compute (8), (9) and implement Algorithm 2 for self-energy computation, we use

`!$acc data` to declare an OpenACC data region within the subroutine `get_2B_SE_kernel` in Fig. 9. The use of such an OpenACC directive eliminates the need for repeated movement of arrays between the host and device, thereby further reducing the data transfer cost.

For the test problem, optimizing memory allocation and data transfer between the host and device yields 8.78% performance improvement in the self-energy computation. Similar techniques can be used to reduce memory allocation and data transfer overhead in the evaluation of the collision integral.

D. Trading on-the-fly index calculation for reduced irregular memory access

In the OpenMP implementation shown in Fig. 4, the k-point index `iqk` over which the Green's function contractions are performed in the self-energy evaluation are retrieved indirectly from arrays `allind_kpk` and `allind_kmk` which store the sums and differences of two given k-points. These arrays are derived from the FORTRAN structure `bz`. On CPUs, as long as these arrays, which are not very large, can fit in the L2 cache, retrieving indices stored in these arrays does not introduce much overhead. However, on GPUs, the indirect addressing facilitated by these indices breaks coalesced memory access pattern required to enable concurrency in fetching data in parallel. Furthermore, the use of derived data type makes it difficult to ensure data structures are correctly mapped and synchronized. Because the indices stored in these arrays can all be computed by simple formulae, we can evaluate them on-the-fly.

In the model problem, we sample the k-points uniformly on the interval $[-\pi, \pi)$ to have

$$k_j = -\pi + \frac{2(j-1)}{n_k}\pi, \quad j = 1, ..., n_k. \tag{10}$$

The sum of two sampled k-points falls within $[-2\pi, 2\pi)$. By taking into account the periodicity of the k-points, we can obtain the index of $k_{j_1} + k_{j_2}$ among all sampled k-points in (10), for $j_1, j_2 = 1, ..., n_k$, which was originally stored in the two-dimensional array `bz%allind_kpk` as follows,

$$\begin{aligned} &\texttt{bz\%allind_kpk}(j_1, j_2) \\ &= \begin{cases} j_1 + j_2 + n_k/2 - 1, & \text{if } k_{j_1} + k_{j_2} < -\pi; \\ j_1 + j_2 - n_k/2 - 1, & \text{if } -\pi \le k_{j_1} + k_{j_2} < \pi; \\ j_1 + j_2 - 3n_k/2 - 1, & \text{if } k_{j_1} + k_{j_2} \ge \pi. \end{cases} \end{aligned} \tag{11}$$

A similar expression can be used to obtain the index of $k_{j_1} - k_{j_2}$ among the sampled k-points for $j_1, j_2 = 1, 2, ...n_k$.

By evaluating these expressions on the fly, we can avoid unnecessary memory access that impedes concurrent executions of the self-energy and collision integral calculations. This optimization gives a 3.35% improvement in the performance of self-energy computation.

E. Performance assessment

The benefits of all strategies we discussed above are displayed and compared in Fig. 11. We can see that loop reordering and the use of `loop gang` and `loop vector` (the third bar) yields the largest initial gain in performance, reducing the total wallclock time from 182 seconds to 27 seconds. The code restructure to increase the workload in kernels reduces the wall clock time by another factor of 10 (the fifth bar). Overall, we achieve a 98.73% reduction in wall clock time compared to a naive OpenACC implementation after all the optimizations techniques are applied.

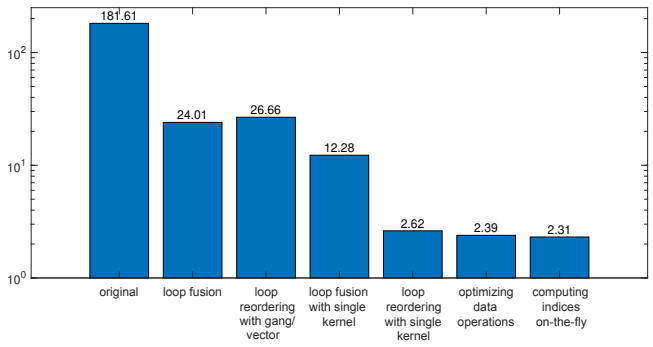


Fig. 11. The wall clock time (seconds) used by different variants of the OpenACC implementation for the model problem.

We also used NVIDIA's Nsight Systems to generate performance profiles of the self-energy calculation before and after optimizations are applied. The Nsight Systems profiles shown in Figures 12 and 13 are generated from a run with $n_k = 1024$ and $n_t = 10$. We used 16 MPI ranks in the computation.

Fig. 12 shows the profile result with direct application of the OpenACC directive `parallel loop` to the computing kernels in the self-energy calculation. We can observe that the kernels are launched frequently, and the kernel to compute Σ^2 as shown in Algorithm 2 costs much more than the other two kernels. It takes about 18 seconds in total to compute the self-energy.

After we apply all the optimizations discussed in the previous subsections, the profile result turns to Fig. 13. Compared to Fig. 12, there are fewer memory transfers and fewer kernel launches. It only takes about 100 milliseconds for the calculation, which shows a significant improvement from 18 seconds.

V. CUDA FORTRAN IMPLEMENTATION

Although OpenACC provides a user friendly programming model to launch concurrent computation on GPU devices, it has some inherent limitations on the management and optimization of GPU resource utilization. Because OpenACC relies on the compiler to partition computational work loads and map them to thread blocks and threads at the compile time, programmers lack complete control over thread utilization and optimization. CUDA FORTRAN [20] provides mechanisms for programmers to explicitly define the granularity and hierarchy of concurrent execution, thereby optimizing GPU resource utilization. However, utilizing these mechanisms requires extra effort and meticulous planning. In this section, we show how we use CUDA FORTRAN to implement the self-energy

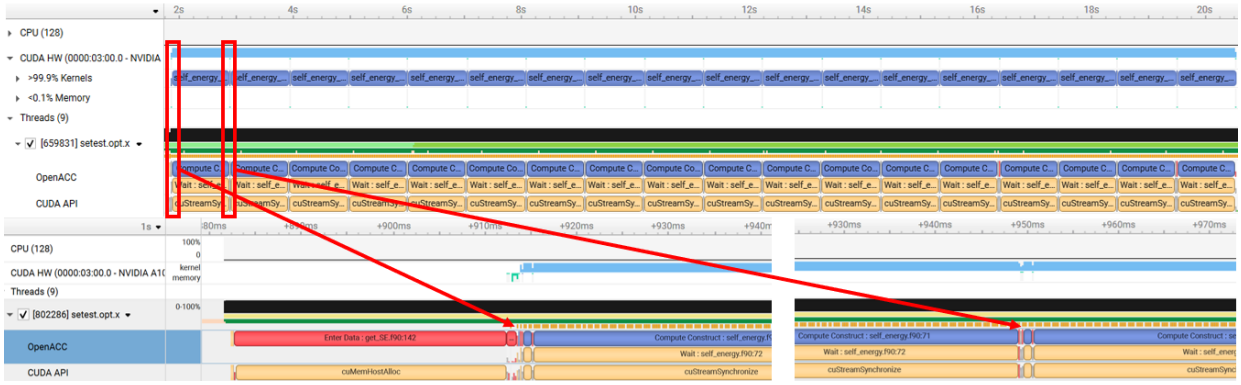


Fig. 12. Performance profile generated from Nsight Systems for self-energy calculation. The top figure captures the sequence of events occurred in the entire self-energy calculation, while the bottom figure highlights work performed in the first two launched kernels.

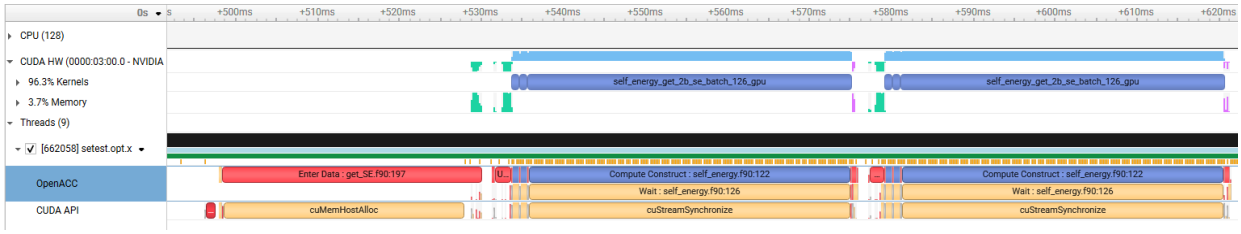


Fig. 13. The profile result from Nsight Systems for the OpenACC implementations after the optimizations to compute $\Sigma(\mathbf{k}; t, t')$.

computation. Similar techniques can be used for the collision integral calculations as well.

A. Kernel launch

One advantage of CUDA FORTRAN is the direct control it provides over GPUs thread utilization, allowing for customized GPU kernels optimized for performance. A kernel launch requires one to specify the number of thread blocks and the number of threads contained in each block. In CUDA FORTRAN, thread block and thread indices can be organized on a configurable three-dimension grid so that one can easily designate a subset of thread blocks or threads to perform a certain task while other thread blocks or threads can be used to perform other tasks concurrently. We have the flexibility to tailor the grid and block dimensions for each kernel and specify the level of parallelism for individual indices.

Upon analyzing data dependencies, we determine that it is feasible to launch three kernels when calling the subroutine `get_2B_SE_kernel` for self-energy calculation. The first kernel handles the computation described in (8), the second carries out the contraction of P and $G^<$ defined in (9) as well as the computation of Σ^2 shown in Algorithm 2 concurrently. The last kernel sums up the first and second terms in (7) by combining results produced from the preceding kernels.

All the kernels in the `get_2B_SE_kernel` subroutine, are launched by specifying a number of thread blocks and threads in the `<<<blocks,threads>>>` clause as shown in Figure 14 with both `blocks` (representing thread blocks) and `thread` (representing threads within each thread block) arranged as three-dimensional grids.

```
...
integer, parameter :: THREADS_PER_BLOCK=128
...
! kernel 2
blocks = dim3(my_nk,batch_size,2)
threads = dim3(THREADS_PER_BLOCK,1,1)
call get_2B_SE_batch_kernel<<<blocks,threads>>>&
          (se_d, ...)
...
```

Fig. 14. CUDA FORTRAN kernel launch for self-energy calculation with thread block and thread configurations specified by 3D arrays `blocks` and `threads`.

The dimension of `block` is defined to be `my_nk` $\times$ `nt` $\times 2$. In the second kernel, the third dimension of `blocks` is used to separate thread blocks into two groups. As shown in Figure 15, the group of thread blocks indexed by `gridIdx%z=1` are designated to compute (9), whereas the group indexed by `gridIdx%z=2` is scheduled to compute Σ^2 at the same time. It is worth noting that OpenACC does not support this simultaneous computation of (9) and Σ^2. This distinction highlights the superior parallelization capabilities offered by CUDA FORTRAN compared to OpenACC and results in enhanced performance. Within the group of thread blocks designated to compute $\Sigma^2(\mathbf{k}, t, t')$, thread blocks indexed by different `gridIdx%y` values are assigned to compute $\Sigma^2(\mathbf{k}, t, t')$ associated with different t or t' values. Thread blocks with the group indexed by the same `gridIdx%y` values and `gridIdx%z=2` are used to execute the k loop in Algorithm 2 in parallel.

All threads within each thread block are used to execute

```
kk = blockIdx%x
it = blockIdx%y
blocksize = blockDim%x
red_size = MIN(blocksize, nk)
tid = threadIdx%x

if (blockIdx%z==1) then
  ... ! compute (9)
else
  Ptmp11(tid) = (0d0,0d0)
  ...
  ikk = start_kpts+kk-1
  do kp=tid,nk,blocksize
    do kpp=1,nk
      iqk = ...
      Ptmp11(tid) = Ptmp11(tid)+Ut(it)*...
      ...
    enddo
  enddo
  call syncthreads()
  ! reduction to thread 1
  s = 1
  do while (s<red_size)
    if (MOD(tid, 2*s)==1) then
      if (tid+s<=red_size) then
        Ptmp11(tid) = Ptmp11(tid)+Ptmp11(tid+s)
        ...
      endif
    endif
    call syncthreads();
    s = s*2
  enddo
  if (tid==1) then
    se2_d(kk,1,1,it) = Ptmp11(1)
    ...
  endif
endif
```

Fig. 15. The CUDA Fortran implementation of the $\Sigma^2_{jm}(\mathbf{k};t,t')$ calculation at a particular (t,t') pair.

the nested j, m, q, k' loops in parallel. The dimension of `threads` is set to `THREADS_PER_BLOCK` $\times 1 \times 1$, where `blockDim%x=THREADS_PER_BLOCK` is a pre-determined and optimized thread count. In Fig 14, it is set to 128. However, the optimial choice of `THREADS_PER_BLOCK` depends on the size of the problem, i.e., the number of k-points and the number of bands.

.

B. Reduction in the shared memory for summation

As is the case in the OpenMP and OpenACC implementations of self-energy calculation, a reduction is needed to sum up all terms in $\Sigma^2_{jm}(\mathbf{k};t,t')$ from (7) if the arrays used to hold the G and Σ are allocated in the shared memory which has a much higher memory bandwidth. To avoid bank conflict, we unroll the `ii` and `jj` loop and define four temporary arrays `Ptmp11`, `Ptmp12`, `Ptmp21` and `Ptmp22` to hold Σ^2_{11}, Σ^2_{12}, Σ^2_{21} and Σ^2_{22} separately.

CUDA FORTRAN offers several options to perform the reduction required to sum up several arrays. The easiest option is to use the intrinsic function `sum`, which performs reduction automatically. For example, summing up all elements of `Ptmp11` can be done by calling `sum(Ptmp11)`. When `THREADS_PER_BLOCK` is set to 128, this implementation option uses 2.23 seconds to evaluate the self-energy for the model problem defined earlier.

It is not clear which reduction algorithm is implemented in the `sum` function. An efficient binary-tree based reduction algorithm can be easily implemented in CUDA FORTRAN by mapping each element of the array to be summed to a node in a binary tree, and summing these elements pairwise in a hierarchical fashion from the leaves of the tree towards the root as shown in Fig. 16. One way to implement this binary-tree based reduction is to use CUDA FORTRAN's warp-level primitives to repeatedly reshuffle elements of the array to be summed to enable concurrent pairwise summation in a recursive fashion. Fig. 17 shows how such a summation is performed for `Ptmp11`. At the end of the `while` loop, the first element of `Ptmp11` contains the sum of all its elements. If the `Ptmp11` has a size of `red_size`, the number of loop iterations is $\lceil \log_2(\texttt{red_size}) \rceil$ – significantly fewer than the `red_size` iterations required for sequential summation when `red_size` is large. Our experiments shows that takes 2.10 seconds to execute the this implementaiton for the model problem. This is slightly faster than using the `sum` function.

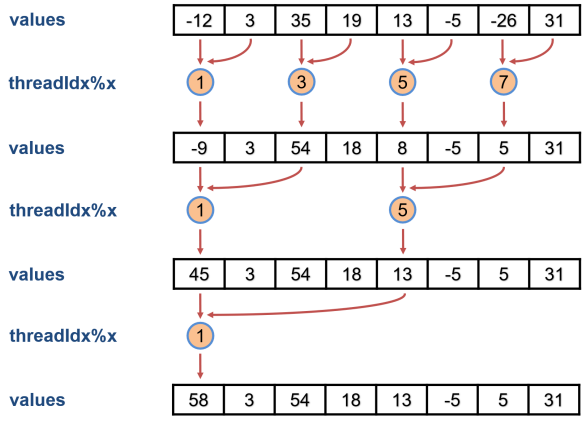


Fig. 16. An illustration of a parallel binary-tree based reduction scheme for summing up elements of an array of size 8.

```
s = 1
do while (s<red_size)
  tmp = __shfl_xor(Ptmp11, 2*(s-1)))
  Ptmp11 = Ptmp11 + tmp
  s = s*2
enddo
```

Fig. 17. A temporary array `tmp` is created by warp-level primitive `__shfl_xor` to enable recursive pairwise summation in a binary-tree based reduction algorithm.

One drawback of the implementation shown in Fig. 17 is that all elements of `Ptmp11` and `tmp` are summed in every loop iteration, even though the number of elements that need to be added is halved each time. In the final iteration, only `Ptmp11(1)` and `tmp(1)` need to be summed. To reduce the number of additions, we use the thread index `tid = threadIdx%x` and an offset `s`, which is set to an integer of

power of 2, to select the elements of the Ptmp11, Ptmp12, Ptmp21, Ptmp22 arrays to be summed pairwise in parallel within the while loop shown in Fig. 15. A call to the syncthread() function is made at the end of each loop iterate to ensure all summations at each level of the binary tree are completed before we move on to the next level. For the same test problem, this approach reduces the summation time so that the whole computation takes 1.95 seconds, which represents a 13% improvement over the use of the intrinsic sum directive. This also gives a 16% improvement compared to the OpenACC implementation, which takes 2.31 seconds for the model problem. Clearly, CUDA FORTRAN provides more flexibility to implement efficient concurrent reduction in shared memory compared to OpenACC which relies completely on the compiler for reduction. Such flexibility results in significant performance improvement.

C. Memory management

In CUDA FORTRAN, explicit memory management helps reduce memory allocation and data transfer overheads. Arrays can be directly allocated on the GPU if they are used exclusively on the GPU. Such an array can be passed as an argument to a subroutine and be used on the GPU within the subroutine. This feature eliminates the need to transfer data from a CPU to a GPU and update the data in a coherent fashion, which is required in OpenACC.

By allocating the arrays se1_d and se2_d that hold Σ^1_{jm} and Σ^2_{jm} respectively on the GPU directly as shown in Fig. 18 and passing them to the subroutine get_2B_SE_kernel, we can avoid the copying and update required in the OpenACC implementation shown in Fig. 9 and reduce memory management overhead.

```
complex(DPC), device, allocatable ::&
    Gtmp1_d(:,:,:,:), Gtmp2_d(:,:,:,:),&
    U_t1_d(:), U_t2_d(:), P_d(:,:,:,:),&
    se1_d(:,:,:,:), se2_d(:,:,:,:)
...
allocate(Gtmp1_d(nk,dim,dim,nt))
...
allocate(se2_d(my_nk,dim,dim,nt))
...
Gtmp1_d = G(:,:,:,&
    ((itp-1)*nt+1):(itp*nt),nt)
...
call get_2B_SE_kernel(se_d, my_nk, Gtmp1_d,&
    Gtmp2_d, U_t2_d, U_t1_d, P_d, se1_d, se2_d)
se(:,:,:,((itp-1)*nt+1):(itp*nt),nt)= se_d
...
```

Fig. 18. Device memory allocation and data transfer between host and device in CUDA FORTRAN.

CUDA FORTRAN also allows data allocated on the host to be copied to the device and vice versa through a simple assignment statement. For example, in Fig. 18, the assignment Gtmp1_d = G(:,:,:,((itp-1)*nt+1):(itp*nt),nt) is used to copy a portion of the array G allocated on the host to the Gtmp1_d array allocated on the device. Similarly, the assignment of se_d to se(:,:,:,((itp-1)*nt+1):(itp*nt),nt) at the end of the do loop in Fig. 18 copies data from the device to the host.

VI. Computational results

A. The hardware

We conduct all computational experiments on the Perlmutter supercomputer maintained at National Energy Scientific Computing Center (NERSC). The CPU nodes on Perlmutter were described in Section III, and each GPU node on Perlmutter is composed of 1 Milan CPU and 4 NVIDIA A100 (Ampere) GPUs. Each GPU is equipped with 40 GB of high bandwidth memory (HBM) with 1555.2 GB/s GPU memory bandwidth. The peak performance is 9.7 GPU TFlops for the FP64 data type.

B. Comparison of the OpenMP, OpenACC and CUDA FORTRAN implementations

In this subsection, we compare the performance of the self-energy as well as the collision integral computation implemented with OpenMP (for CPUs), OpenACC and CUDA FORTRAN (for GPUs). CUDA aware MPI is used to for internode communication.

In the first example, we set $n_k = 1024$, $n_t = 500$ to ensure that a significant amount of work is performed in these calculations and the performance of these calculations can be measured accurately. We then vary the number of MPI ranks used to parallelize the computation among different nodes.

The OpenMP implementation is always run with 64 threads within each MPI rank, 4 MPI ranks are launched on each CPU node.

The performance of different implementations for self-energy and collision integral calculations is presented in Table I and Table II, respectively.

In Table I, we report both the wall clock times used by each implementation to compute the self-energy and the speedup achieved by the OpenACC and CUDA FORTRAN implementations executed on GPUs relative to the OpenMP implementation executed on CPUs. We observe that both the OpenACC and CUDA FORTRAN implementations executed on GPUs are more than 100x faster than the OpenMP implementation executed on CPUs. The CUDA FORTRAN implementation consistently outperforms the OpenACC implementation, which underscores the advantage of having full control over the memory management and kernel launch. On both CPUs and GPUs, the self-energy computation has nearly perfect strong MPI parallel scaling up to 128 MPI ranks. Performance improvement can still be observed with 1024 MPI ranks. However, the scaling beyond 128 MPI ranks is less than perfect. This is due to increased MPI communication overhead relative to reduced computational workload on each MPI rank. Because the computational time consumed on each GPU is significantly lower than that on a CPU, communication overhead becomes more pronounced in GPU runs. As a result, the OpenACC and CUDA FORTRAN speedup over the OpenMP implementation also decreases slightly in general.

TABLE I
COMPARISON OF WALL CLOCK TIME (IN SECONDS) TAKEN TO PERFORM SELF-ENERGY CALCULATIONS IN PARALLEL ON CPUS AND GPUS USING DIFFERENT PROGRAMMING MODELS. THESE CALCULATIONS USE $n_k = 1024$ K-POINTS AND $n_t = 500$ TIME STEPS.

	number of MPI ranks						
	16	32	64	128	256	512	1024
OpenMP (CPU)	1589.12	816.04	438.65	204.39	164.28	117.00	97.62
OpenACC (GPU)	11.21	5.96	3.32	1.99	1.33	1.03	0.91
Speedup	141.76	136.92	132.12	102.71	123.52	113.59	107.27
CUDA FORTRAN (GPU)	9.06	4.84	2.72	1.66	1.13	0.94	0.77
Speedup	175.40	168.60	161.27	123.13	145.38	124.47	126.78

TABLE II
COMPARISON OF WALL CLOCK TIME (IN SECONDS) TAKEN TO PERFORM THE COLLISION INTEGRAL CALCULATION IN PARALLEL ON CPUS AND GPUS USING DIFFERENT PROGRAMMING MODELS. THESE CALCULATIONS USE $n_k = 1024$ K-POINTS AND $n_t = 500$ TIME STEPS.

	number of MPI ranks						
	16	32	64	128	256	512	1024
OpenMP (CPU)	124.93	74.89	31.25	19.93	9.42	5.06	2.84
OpenACC (GPU)	7.83	3.95	2.03	1.04	0.56	0.27	0.16
Speedup	15.96	18.96	15.39	19.16	16.82	18.74	17.75

The performance of the collision integral calculation is reported in Table II. We observe that the OpenACC implementation of the collision integral calculation executed on GPUs is consistently 15–19 times faster than the OpenMP implementation executed on CPUs. Because the amount of work in the collision integral calculation is much less than for that in the self-energy calculation, we cannot take full advantage of the concurrency provided by a large number of thread blocks and threads. When $n_t = 500$, the collision integral calculation time is significantly less than that required for self energy calculation on CPUs. However, when ported to GPUs, the collision integral calculation time is close to that used for self-energy calculation. Furthermore, excellent strong scaling is observed for collision integral calculation with up to 512 MPI ranks in the OpenACC implementation. Communication overhead contributed to a slight slow down when the computation is performed on 1024 MPI ranks.

In addition to strong scaling, we also examined the weak scaling of the GPU implementations of the self-energy and collision integral calculation by fixing the number of k-points on each MPI rank to 16. As a result, in each run the total number of k-points which defines the size of the problem to be solved is set to $n_k = 16\times$ the number of MPI ranks. For this experiment, we keep the values of n_t unchanged.

As we discussed in section II, the overall complexity of the self-energy calculation is $\mathcal{O}(n_k^3)$. However, the evaluation of Σ^1 only takes $\mathcal{O}(n_k^2)$ operations. As a result, in the weak scaling experiments, when the number of MPI ranks is doubled, we expect the wallclock time to increase by a factor that is between 2 and 4. Such scaling is indeed observed in Table III. We also observe that, when the number of MPI ranks is fewer than 8, the OpenACC implementation takes significantly more time than the CUDA FORTRAN version. This performance gap is likely due to suboptimal thread block and thread configurations selected by the compiler during the launch of kernels in OpenACC-decorated loops.

Because the complexity of collision integral evaluation is $\mathcal{O}(n_k)$, we expect the wall clock time to remain close to constant when we double the number of MPI ranks. Such scaling is indeed observed in Table IV.

C. Tuning the grid and block sizes in OpenACC and CUDA FORTRAN

The performance of OpenACC and CUDA FORTRAN implementations varies with respect to several parameters such as the gang and vector size in OpenACC and dimension of the thread block and thread grids. In this subsection, we examine how different choices of these parameter affect the performance of self-energy calculation. The test results presented below uses $n_k = 1024$, $n_t = 500$. The computation is distributed among 16 MPI ranks. The wall clock time associated with different choices of vector size and gangle size in OpenACC in shown in Fig. 19 as a heatmap. Fig. 20 shows the wall clock time of self-energy calculation used by the CUDA FORTRAN implementation for different choices of blockDim and gridDim. These figures illustrate the impact of various gang sizes and vector sizes in OpenACC, as well as different grid and block dimensions in CUDA FORTRAN, respectively.

From these figures, a general trend can be observed where the computational time tends to decrease as gang and vector sizes in OpenACC, as well as grid and block dimensions in CUDA FORTRAN, increase.

In the OpenACC implementation, the choice of the gang size exhibits a more significant impact on the result. After increasing the gang size to 1024, there is little difference when we increase the vector size from 128 to 512. Surprisingly, a further improvement in performance is achieved when using

TABLE III

THE WALL CLOCK TIME (IN SECONDS) OF THE SELF-ENERGY CALCULATION FOR PROBLEMS IN WHICH n_k IS SET TO 16 TIMES THE NUMBER OF MPI RANKS ON CPUS AND GPUS. THE NUMBER OF TIME STEPS IS SET TO $n_t = 500$.

	number of MPI ranks						
	1	2	4	8	16	32	64
OpenMP (CPU)	1.91	3.68	7.97	19.87	52.96	185.59	438.65
OpenACC (GPU)	0.49	0.43	1.00	0.08	0.25	0.91	3.33
CUDA Fortran (GPU)	0.01	0.02	0.03	0.09	0.24	0.78	2.73

TABLE IV

THE WALL CLOCK TIME (IN SECONDS) OF THE COLLISION INTEGRAL CALCULATION FOR PROBLEMS IN WHICH n_k IS SET TO 16 TIMES THE NUMBER OF MPI RANKS ON CPUS AND GPUS. THE NUMBER OF TIME STEPS IS SET TO $n_t = 500$.

	number of MPI ranks						
	1	2	4	8	16	32	64
OpenMP (CPU)	28.72	29.09	30.41	31.50	30.62	31.56	31.25
OpenACC (GPU)	2.55	2.91	3.52	2.17	2.20	2.21	2.22

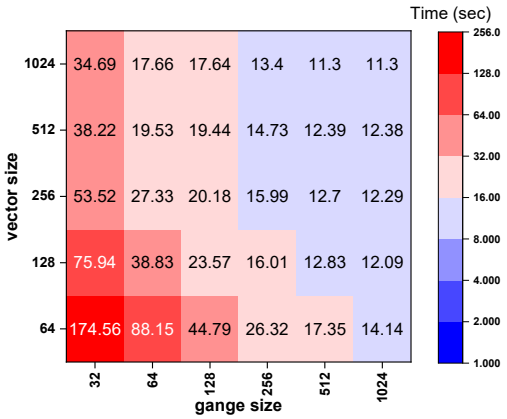


Fig. 19. Comparison of wall clock time (in seconds) used in the OpenACC implementation of the self-energy calculation with different gang and vector sizes. These calculations use $n_k = 1024$ k-points and $n_t = 500$ time steps, and are run on 16 MPI ranks.

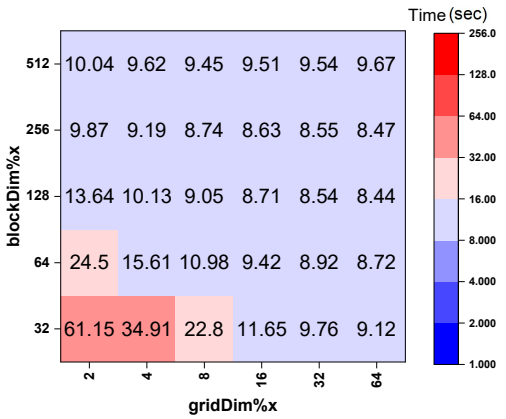


Fig. 20. Comparison of wall clock time (in seconds) used in the CUDA FORTRAN implementation of the self-energy calculation with different thread grid and block sizes. These calculations use $n_k = 1024$ k-points and $n_t = 500$ time steps, and are run on 16 MPI ranks.

a vector size of 1024. Based on the data presented in Fig. 19, optimal choices for this example appear to be a gang size of 512, and a vector size of 1024.

In the CUDA FORTRAN implementation, the key grid and block dimensions to consider are `gridDim%x` and `blockDim%x`. Given the current problem setting, `gridDim%x` is limited to a maximum of 64 since `my_nk` = 64. Analysis of the results in Fig. 20 reveals that once we increase `blockDim%x` to 128, further increasing it do not lead to noticable improvements. The performance even becomes worse when increasing it to 512, possibly due to excessive GPU resource requirements or significant thread divergence. The limited shared memory restricts setting `blockDim%x` to 1024 as with the vector size in OpenACC. The optimal choice, based on the results in Fig. 20, is to set `gridDim%x` = 64 and `blockDim%x` = 128. This choice results in a 1.34x speedup when compared to the best computational time achieved with the OpenACC implementation.

VII. CONCLUSION

In this work, we developed high-performance implementations of a numerical method for solving the Kadanoff-Baym equations (KBE) on both CPU and GPU architectures, using a range of parallel programming models. For GPU-based acceleration, we developed several optimization techniques aimed at improving performance by maximizing computational resource utilization and minimizing overhead associated with kernel launches and memory management.

Our experiments demonstrate that the GPU implementation of the self-energy computation, developed using both OpenACC and CUDA FORTRAN, achieves over 100× speedup on NVIDIA A100 GPUs compared to an AMD EPYC many-core CPU baseline. For the collision integral computation, GPU acceleration yields more than a 10× performance gain.

We found that the explicit control over thread hierarchy and memory enabled by CUDA FORTRAN leads to better performance than OpenACC, highlighting the value of fine-grained optimization in GPU programming.

All three implementations (MPI/OpenMP CPU, MPI/OpenACC GPU, and MPI/CUDA FORTRAN GPU) exhibit excellent strong and weak parallel scaling, underscoring the portability and scalability of our approach across modern high-performance computing systems. These results demonstrate the viability of our method for large-scale non-equilibrium Green's function calculation for quantum many-body systems and contribute to advancing the state of the art in GPU-accelerated scientific computing.

ACKNOWLEDGMENT

This work is supported by the Center for Computational Study of Excited-State Phenomena in Energy Materials (C2SEPEM) at the Lawrence Berkeley National Laboratory, which is funded by the U. S. Department of Energy, Office of Science, Basic Energy Sciences, Materials Sciences and Engineering Division, under Contract No. DE-AC02-05CH11231, as part of the Computational Materials Sciences Program. This work is also supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. The authors acknowledge the computational resources of the National Energy Research Scientific Computing (NERSC) center using ASCR-ERCAP-m1027.

REFERENCES

[1] L. P. Kadanoff and G. Baym, *Quantum Statistical Mechanics*. New York: W. A. Benjamin, Inc., 1962.

[2] G. Stefanucci and R. van Leeuwen, *Nonequilibrium Many-Body Theory of Quantum Systems: A Modern Introduction*. Cambridge University Press, 2013.

[3] L. P. Kadanoff and G. Baym, *Quantum statistical mechanics: Green's function methods in equilibrium and nonequilibrium problems*. CRC Press, 2018.

[4] A. Fetter and J. Walecka, *Quantum Theory of Many-Particle Systems*. New York: McGraw-Hill Book Company, 1971.

[5] G. Baym and L. P. Kadanoff, "Conservation laws and correlation functions," *Physical Review*, vol. 124, no. 2, p. 287, 1961.

[6] L. Keldysh, "Diagram technique for nonequilibrium processes," *Soviet Physics JETP*, vol. 20, no. 4, pp. 1018–1026, 1965.

[7] P. Lipavskỳ, V. Špička, and B. Velickỳ, "Generalized Kadanoff-Baym ansatz for deriving quantum transport equations," *Phys. Rev. B*, vol. 34, no. 10, p. 6933, 1986.

[8] J. E. Hirsch, "Two-dimensional Hubbard model: Numerical simulation study," *Physical Review B*, vol. 31, no. 7, p. 4403, 1985.

[9] S. R. White, D. J. Scalapino, R. L. Sugar, E. Loh, J. E. Gubernatis, and R. T. Scalettar, "Numerical study of the two-dimensional Hubbard model," *Physical Review B*, vol. 40, no. 1, p. 506, 1989.

[10] A. Macridin, M. Jarrell, T. Maier, and G. Sawatzky, "Physics of cuprates with the two-band Hubbard model: The validity of the one-band Hubbard model," *Physical Review B*, vol. 71, no. 13, p. 134527, 2005.

[11] P. Werner, E. Gull, O. Parcollet, and A. J. Millis, "Momentum-selective metal-insulator transition in the two-dimensional Hubbard model: An 8-site dynamical cluster approximation study," *Physical Review B*, vol. 80, no. 4, p. 045120, 2009.

[12] P. Pudleiner, T. Schäfer, D. Rost, G. Li, K. Held, and N. Blümer, "Momentum structure of the self-energy and its parametrization for the two-dimensional Hubbard model," *Physical Review B*, vol. 93, no. 19, p. 195134, 2016.

[13] K. Balzer, "Nonequilibrium Green's Function Approach to Artificial Atoms," Ph.D. dissertation, Inst. für Theoretische Physik und Astrophysik der Christian-Albrechts-Univ., 2007.

[14] A. Stan, N. Dahlen, and R. van Leeuwen, "Time propagation of the Kadanoff-Baym equations for inhomogeneous systems," *J. Chem. Phys.*, vol. 130, p. 224101, 2009.

[15] M. Bonitz, D. Kremp, D. Scott, R. Binder, W.-D. Kraeft, and H. S. Köhler, "Numerical analysis of non-Markovian effects in charge-carrier scattering: one-time versus two-time kinetic equations," *Journal of Physics: Condensed Matter*, vol. 8, no. 33, p. 6057, 1996.

[16] F. Hoppensteadt, Z. Jackiewicz, and B. Zubik-Kowal, "Numerical solution of Volterra integral and integro-differential equations with rapidly vanishing convolution kernels," *BIT Numerical Mathematics*, vol. 47, no. 2, pp. 325–350, 2007.

[17] C. A. Kennedy and M. H. Carpenter, *Diagonally Implicit Runge-Kutta methods for ordinary differential equations, a review*. National Aeronautics and Space Administration, Langley Research Center, 2016.

[18] "The OpenACC Application Programming Interface Version 3.3," https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf, november 2022.

[19] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 2017.

[20] "NVIDIA CUDA Fortran Programming Guide," https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/index.html.