Commun. Comput. Phys.
doi: 10.4208/cicp.OA-2024-0072

Vol. **36**, No. 5, pp. 1157-1185
November 2024

A Simple GPU Implementation of Spectral-Element Methods for Solving 3D Poisson Type Equations on Rectangular Domains and Its Applications

Xinyu Liu[1], Jie Shen[2] and Xiangxiong Zhang[1,*]

[1] *Purdue University, 150 N. University Street, West Lafayette, IN 47907-2067, USA.*
[2] *Eastern Institute of Technology, Ningbo, Zhejiang 315200, P.R. China.*

Received 26 March 2024; Accepted (in revised version) 13 June 2024

Abstract. It is well known since 1960s that by exploring the tensor product structure of the discrete Laplacian on Cartesian meshes, one can develop a simple direct Poisson solver with an $\mathcal{O}(N^{\frac{d+1}{d}})$ complexity in d-dimension, where N is the number of the total unknowns. The GPU acceleration of numerically solving PDEs has been explored successfully around fifteen years ago and become more and more popular in the past decade, driven by significant advancement in both hardware and software technologies, especially in the recent few years. We present in this paper a simple but extremely fast MATLAB implementation on a modern GPU, which can be easily reproduced, for solving 3D Poisson type equations using a spectral-element method. In particular, it costs less than one second on a Nvidia A100 for solving a Poisson equation with one billion degree of freedoms. We also present applications of this fast solver to solve a linear (time-independent) Schrödinger equation and a nonlinear (time-dependent) Cahn-Hilliard equation.

AMS subject classifications: 65F05, 65F60, 65Y05, 65M06, 65M60, 65M70, 65N06, 65N30, 65N35

Key words: 3D Poisson equation, direct solver, spectral element methods, rectangular domain, GPU, tensor matrix multiplication.

1 Introduction

It is well known that the tensor product structure of the discrete Laplacian on Cartesian meshes can be used to invert the Laplacian since 1960s [15]. This approach has been particularly popular for spectral and spectral-element methods [2, 8, 12, 17, 18]. In fact,

*Corresponding author. *Email addresses:* liu1957@purdue.edu (X. Liu), jshen@eitech.edu.cn (J. Shen), zhan1966@purdue.edu (X. Zhang)

this method can be used for any discrete Laplacian on a Cartesian mesh. In this paper, as an example, we focus on the Q^k spectral-element method, which is equivalent to the classical continuous finite element method with Lagrangian Q^k basis implemented with the $(k+1)$-point Gauss–Lobatto quadrature [13,16].

Tensor based solvers naturally fit the design of graphic processing units (GPUs). The earliest successful attempts to accelerate the computation of high order accurate methods in scientific computing communities include nodal discontinuous Galerkin method [11] almost fifteen years ago. These pioneering efforts of GPU acceleration of high order methods, or even those ones published later such as [3] in 2013, often rely on intensive coding.

In recent years, the surge in computational demands from machine learning and neural network based approaches has led to the evolution of modern GPUs. Correspondingly, software technologies have advanced considerably, streamlining the utilization of GPU computing. The landscape of both hardware and software has dramatically transformed, differing substantially from what existed a decade or even just two years ago.

In this paper, we present a straightforward yet robust implementation of accelerating the spectral-element method for three-dimensional discrete Laplacian on modern GPUs. In particular, for a total number of degree of freedoms as large as 1000^3, the inversion of the 3D Laplacian using an arbitrarily high order Q^k spectral-element method, takes no more than one second on one Nvidia A100 GPU card with 80G memory. While this impressive computational speed is naturally contingent on the hardware, it is noteworthy that our approach is grounded in a minimalist MATLAB implementation, ensuring ease of replication. In the Appendix, we give a full MATLAB code for solving a 3D Poisson equation on a rectangular domain using Q^k spectral-element method.

We remark that a similar simple implementation on GPUs can also be achieved using Python using the Python package JAX, which however provides better performance than MATLAB only for single precision computation under TensorFloat-32 precision format. Our numerical tests and comparison on one Nvidia A100 GPU card suggest that MATLAB implementation performs better than Python for double precision computation for large problems like one billion DoFs.

We emphasize that the ability of solving Poisson type equation fast can play an important role in many fields of science and engineering. In fact, a large class of time dependent nonlinear systems, after a suitable implicit-explicit (IMEX) time discretization, often reduces to solving Poisson type equations at each time step (see, for instance, [19]). Therefore, having a simple, accurate and very fast solver for Poisson type equations can lead to very efficient numerical algorithms on modern GPUs for these nonlinear systems which include, e.g., Allen-Cahn and Chan-Hillard equations and related phase-field models [19], nonlinear Schrödinger equations, Navier-Stokes equations and related hydro-dynamic equations through a decoupled (projection, pressure correction etc.) approach [7]. In particular, by using the code provided in the Appendix, one can build, with a relatively easy effort, very efficient numerical solvers on modern GPUs for these time dependent complex nonlinear systems.

The rest of the paper is organized as follows. In Section 2, we give the implementation

details for 3D problems, which is robust for very high order elements with the computation approach in Section 2.3. In Section 3, we demonstrate the good performance of this simple implementation for equations including the Poisson equation, a variable coefficient elliptic problem solved by the preconditioned conjugate gradient descent using Laplacian as a preconditioner, as well as a Cahn–Hilliard equation. Although our focus in this paper remains on the spectral-element method for these particular equations, similar results can be obtained for other problems with the same tensor product structure, e.g., finite difference schemes in implementing the matrix exponential in the exponential time differencing [6] and spectral fractional Laplacian [4]. We also compare it with a similar simple implementation in Python, and the numerical results suggest that MATLAB implementation is better than Python for double precision computation on A100. Some concluding remarks are given in Section 4.

2 A spectral-element method for Poisson type equations

To fix the idea, we describe the implementation details for solving the Poisson type equation

$$\alpha u - \Delta u = f, \tag{2.1}$$

with a constant coefficient $\alpha > 0$ and homogeneous Neumann boundary conditions on a rectangular domain Ω. In this paper, we only consider the spectral-element method with continuous piecewise Q^k polynomial basis on uniform rectangular meshes, and all integrals are approximated by $(k+1)$-point Gauss–Lobatto quadrature [14].

2.1 The spectral-element method in two dimensions

We first consider the two dimensional case. As shown in Fig. 1, such quadrature points naturally define all degree of freedoms since a single variable polynomial of degree k is uniquely determined by its values at $k+1$ points.

On a rectangular mesh for Q^k basis as shown in Fig. 1(a) or (c), let (x_i,y_j) $(i=1,\cdots,N_x; j=1,\cdots N_y)$ denote all the points in Fig. 1(b) or (d). We consider the following finite element space of continuous piecewise Q^k polynomials:

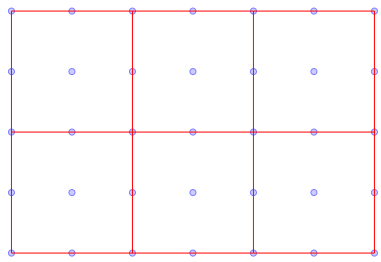
$$V^h = Span\{\phi_i(x)\phi_j(y),\ 1\le i\le N_x,\ 1\le j\le N_y\},$$

where $\phi_i(x)$ $(i=1,\cdots,N_x)$ denotes the i-th Lagrangian interpolation polynomial of degree k in the x direction as shown in Fig. 1(b) or (d).

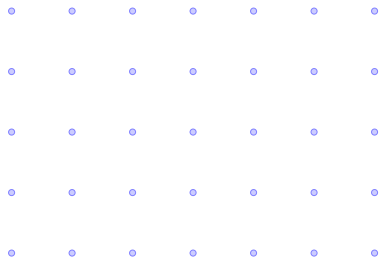
Then, the Q^k spectral-element method for solving (2.1) on a rectangular domain Ω is to seek $u_h \in V^h$ satisfying

$$\alpha\langle u_h, v_h\rangle + \langle \nabla u_h, \nabla v_h\rangle = \langle f, v_h\rangle, \quad \forall v_h \in V^h, \tag{2.2}$$

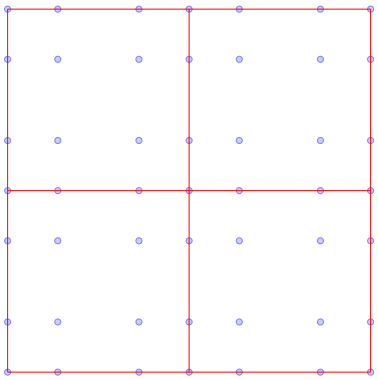
where $\langle f,g\rangle$ denotes the approximation to the integral $\iint_\Omega f(x,y)g(x,y)dxdy$ by the $(k+1)\times(k+1)$ Gauss-Lobatto quadrature rule in each cell as shown in Fig. 1.



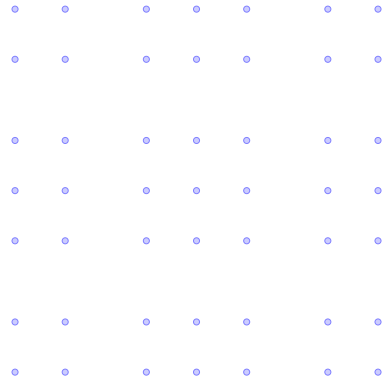
(a) A 3×2 mesh for Q^2 element and the 3×3 Gauss-Lobatto quadrature.



(b) All the quadrature points for Q^2 element.



(c) A 2×2 mesh for Q^3 element and the 4×4 Gauss-Lobatto quadrature.



(d) All the quadrature points for Q^3 element.

Figure 1: An illustration of Lagrangian Q^k element and the $(k+1)\times(k+1)$ Gauss-Lobatto quadrature.

The numerical solution $u_h(x,y)\in V^h$ can be expressed by the basis as

$$u_h(x,y)=\sum_{i=1}^{N_x}\sum_{j=1}^{N_y}u_{i,j}\phi_i(x)\phi_j(y),$$

where the coefficients $u_{i,j}=u_h(x_i,y_j)$ since $\phi_i(x)$ and $\phi_j(y)$ are chosen as the Lagrangian interpolation polynomials at x_i and y_j.

Next, we define the one-dimensional stiffness matrix and the mass matrix as follows. The stiffness matrix S_x is a matrix of size $N_x\times N_x$ with (i,j)-th entry being $\langle\phi_i'(x),\phi_j'(x)\rangle$. The mass matrix M_x is a matrix of size $N_x\times N_x$ with (i,j)-th entry being $\langle\phi_i(x),\phi_j(x)\rangle$. The matrices S_y and M_y are similarly defined, with a size $N_y\times N_y$. Since the basis polynomials $\phi_i(\cdot)$ are Lagrangian interpolants at the quadrature points, the mass matrices M_x and M_y are diagonal.

Remark 2.1. Instead of the Lagrangian basis, we can also use the modal basis $L_j(x)-L_{j+2}(x)$ in each cell, where $L_k(x)$ is the Legendre polynomial of degree k, as the interior

basis functions and the piecewise linear hat function at the intersecting points of two subintervals. This leads to sparse mass and stiffness matrices [2, 12].

Let U be a matrix of size $N_x \times N_y$ with $u_{i,j}$ being its (i,j)-entry. Then $u_h(x,y)$ can be equivalently represented by the matrix U. Similarly, let F be a matrix of size $N_x \times N_y$ with $f(x_i,y_j)$ being its (i,j)-entry. Then the scheme (2.2) can be equivalently written as

$$\alpha M_x U M_y^T + S_x U M_y^T + M_x U S_y^T = M_x F M_y^T. \tag{2.3}$$

Detailed derivations of (2.3) can be found in [10, 14, 20].

2.2 Inversion by eigenvalue decomposition

For any matrix X of size $m \times n$, define a vectorization operation $vec(\cdot)$, and let $vec(X)$ be the vector of size mn obtained by reshaping all entries of X into a column vector in a column by column order. Then for any two matrices A_1, A_2 of proper sizes, it satisfies

$$vec(A_1 X A_2^T) = (A_2 \otimes A_1) vec(X), \tag{2.4}$$

where $\otimes$ denotes the Kronecker product. With (2.4), the Q^k spectral-element method (2.3) is also equivalent to

$$(\alpha M_y \otimes M_x + M_y \otimes S_x + S_y \otimes M_x) vec(U) = (M_y \otimes M_x) vec(F). \tag{2.5}$$

We remark that many other types of boundary conditions on a cubic domain can be written in the form of (2.3) or (2.5) include periodic boundaries and the simple homogeneous Dirichlet boundary treatment described in [14]. For non-homogeneous Neumann and Dirichlet boundary conditions, the boundary conditions will be added to the right hand side of (2.3) or (2.5), and the left hand side of (2.3) or (2.5) stays the same, i.e., the matrix to be inverted stays the same. The left hand side of (2.5) would no longer hold if the boundary value problem does not have a tensor product structure, e.g., on a curved domain.

The linear system (2.3), or equivalently (2.5), can be solved by the following well-known method by using eigenvalue decomposition for only small matrices such as S and M. For convenience, we only consider the simplified equivalent system

$$\alpha U + H_x U + U H_y^T = F, \tag{2.6}$$

or

$$(\alpha + I_y \otimes H_x + H_y \otimes I_x) vec(U) = vec(F), \tag{2.7}$$

where I is the identity matrix and $H = M^{-1}S$.

First, solve a generalized eigenvalue problem for small matrices S and M, i.e., finding eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ satisfying

$$S\mathbf{v}_i = \lambda_i M \mathbf{v}_i. \tag{2.8}$$

Regardless of what kind of basis functions is used in a spectral-element method, the variational form of (2.2) ensures the symmetry of S and M, thus a complete set of eigenvectors exists for (2.8). Let Λ be a diagonal matrix with all eigenvalues λ_i being diagonal entries, and let T be the matrix with all corresponding eigenvectors $\mathbf{v}_i$ as its columns. Then

$$ST = MT\Lambda \Rightarrow H = M^{-1}S = T\Lambda T^{-1}.$$

Thus (2.7) becomes

$$[\alpha + (T_y I_y T_y^{-1}) \otimes (T_x \Lambda_x T_x^{-1}) + (T_y \Lambda_y T_y^{-1}) \otimes (T_x I_x T_x^{-1})] vec(U) = vec(F),$$

which is equivalent to

$$(T_y \otimes T_x)[\alpha + I_y \otimes \Lambda_x + \Lambda_y \otimes I_x](T_y^{-1} \otimes T_x^{-1}) vec(U) = vec(F). \tag{2.9}$$

Notice that $\alpha + I_y \otimes \Lambda_x + \Lambda_y \otimes I_x$ is a diagonal matrix, thus its inverse is simple to compute. Let $\Lambda 2D$ be a matrix of size $N_x \times N_y$ with its (i,j) entry being equal to $\alpha + (\Lambda_x)_{i,i} + (\Lambda_y)_{j,j}$, then (2.9) can be solved by

$$U = T_x[(T_x^{-1} F T_y^{-T})./\Lambda 2D]T_y^T, \tag{2.10}$$

where ./ denotes the entrywise division between two matrices.

2.3 Robust computation of the generalized eigenvalue problem

In our spectral-element implementation, M is diagonal. So we can consider an eigenvalue problem instead of the generalized eigenvalue problem (2.8). A numerically robust method, especially for very high order polynomial basis, is to solve the following symmetric eigenvalue problem. Let

$$H = M^{-1}S = M^{-1/2}(M^{-1/2}SM^{-1/2})M^{1/2}.$$

Since $S_1 = M^{-1/2}SM^{-1/2}$ is real and symmetric, we can first find its eigenvalue decomposition as $S_1 = Q\Lambda Q^T$ where Λ is a diagonal matrix and Q is an orthogonal matrix. Then, we have

$$H = M^{-1/2}(Q\Lambda Q^T)M^{1/2} = T\Lambda T^{-1},$$

with $T = M^{-1/2}Q$ and $T^{-1} = Q^T M^{1/2}$. In Section 3.4, we will show numerical tests validating the robustness of this implementation for very high order elements.

2.4 Implementation for the three-dimensional case

On a three dimensional rectangular mesh, any continuous piecewise Q^k polynomial u_h can be uniquely represented by a 3D array U of size $N_x \times N_y \times N_z$ with (i,j,k)-th entry denoting the point value $u_h(x_i,y_j,z_k)$, where (x_i,y_j,z_k) (i=1,$\cdots$,N_x;j=1,$\cdots$,N_y;k=1,$\cdots$,N_z) denotes all the quadrature points.

For a 3D array U, we define a page as the matrix obtained by fixing the last index of U. Namely, $U(:,:,k)$ for any fixed k is a page of U. For a matrix $U(:,:,k)$ of size $N_x \times N_y$, recall that $vec(U(:,:,k))$ is a column vector of size $N_x N_y$. We define $\hat{U}$ as the following matrix of size $N_x N_y \times N_z$ obtained by reshaping U:

$$\hat{U} = \big[vec(U(:,:,1)) \quad vec(U(:,:,2)) \quad \cdots \quad vec(U(:,:,N_z))\big].$$

Then we define $vec(U)$ as the vector of size $N_x N_y N_z \times 1$ by reshaping $\hat{U}$ in a column by column order.

With the notation above, it is straightforward to verify that

$$(A_3^T \otimes A_2^T \otimes A_1) vec(U) = vec((A_2^T \otimes A_1)\hat{U} A_3). \tag{2.11}$$

Next, we consider how to implement the matrix vector multiplication in (2.11) without reshaping the 3D arrays. Let Y be a 3D array of size $N_x \times N_y \times N_z$ defined by

$$vec(Y) = (A_3^T \otimes A_2^T \otimes A_1) vec(U). \tag{2.12}$$

With the simple property (2.11), in our numerical tests, we find that the following simple implementation of (2.12) in MATLAB 2023 is efficient using two functions *tensorprod* and *pagemtimes*:

```
% Computing a 3D array Y of the same size as U defined above
Y = tensorprod(U,A3,3,1);
Y = pagemtimes(Y,A2);
Y = squeeze(tensorprod(A1,Y,2,1));
```

For the three-dimensional case, for simplicity, we consider Eq. (2.1) with $\alpha = 0$. With similar notation as in the two-dimensional case, the matrix form of the Q^k spectral-element method (2.2) can be given as

$$\big(M_z \otimes M_y \otimes S_x + M_z \otimes S_y \otimes M_x + S_z \otimes M_y \otimes M_x\big)\, vec(U) = \big(M_z \otimes M_y \otimes M_x\big)\, vec(F),$$

or equivalently,

$$\big(I_z \otimes I_y \otimes H_x + I_z \otimes H_y \otimes I_x + H_z \otimes I_y \otimes I_x\big)\, vec(U) = vec(F), \tag{2.13}$$

where U is a 3D array with (i,j,k)-th entry denoting the point value $u_h(x_i,y_j,z_k)$, and F is a 3D array with (i,j,k)-th entry denoting the point value $f(x_i,y_j,z_k)$.

With the eigenvalue decomposition $H = M^{-1}S = T\Lambda T^{-1}$, similar to the derivation of (2.9), the equation (2.13) is equivalent to

$$(T_z \otimes T_y \otimes T_x)\big(I_z \otimes I_y \otimes \Lambda_x + I_z \otimes \Lambda_y \otimes I_x + \Lambda_z \otimes I_y \otimes I_x\big)(T_z^{-1} \otimes T_y^{-1} \otimes T_x^{-1}) vec(U) = vec(F). \tag{2.14}$$

Define a 3D array $\Lambda 3D$ with its (i,j,k)-th entry being equal to $(\Lambda_x)_{i,i} + (\Lambda_y)_{j,j} + (\Lambda_z)_{k,k}$, then (2.14) can be implemented efficiently as the following (Table 1) in MATLAB.

Table 1: The MATLAB 2023 script of implementing (2.14) on both CPU and GPU.

```
% Simple and efficient implementation of (14)
% TInv denotes the inverse matrix of T
U = tensorprod(F,TzInv',3,1);
U = pagemtimes(U,TyInv');
U = squeeze(tensorprod(TxInv,U,2,1));
U = U./Lambda3D;
U = tensorprod(U,Tz',3,1);
U = pagemtimes(U,Ty');
U = squeeze(tensorprod(Tx,U,2,1));
```

3 Numerical tests

In this section, we report the performance of the simple MATLAB implementation in Table 1. In particular, a demonstration code is provided in the Appendix. The performance and speed-up are of course dependent on the hardwares. We test our code on the following three devices:

1. CPU: Intel i7-12700 2.10 GHz (12-core) with 16G memory;

2. GPU: Quadro RTX 8000 (48G memory);

3. GPU: Nvidia A100 (80G memory).

In MATLAB 2023, for computation on either CPU or GPU, the code for implementing (2.14) is the same as in Table 1. On the other hand, matrices like T_x, T_y, T_z and arrays like F and $\Lambda 3D$ must be loaded to GPU memory before performing the GPU computation, see the full code in the Appendix. We define the process of loading matrices and arrays $T_x, T_y, T_z, F, \Lambda 3D$ as the offline step since it is preparatory, and undertaken only once, regardless of how many times the Laplacian needs to be inverted. We define the step in Table 1 as the online computation step. **All the computational time reported in this section are online computational time, i.e., we do not count the offline preparational time.**

3.1 Accuracy tests

We list a few accuracy tests to show that the scheme implemented is indeed high order accurate. In particular, the Q^k ($k \geq 2$) spectral-element method is $(k+2)$-th order accurate for smooth solutions when measuring the ℓ^2 error in function values for solving second order PDEs, which has been rigorously proven recently in [13, 14].

Table 2: Accuracy tests for discrete Laplacian for a 3D problem with Dirichlet boundary conditions and a 3D problem with Neumann boundary conditions.

Q^5 spectral-element method (SEM)						
FEM Mesh	Dirichlet boundary			Neumann boundary		
	Total DoFs	ℓ^2 error	order	Total DoFs	ℓ^2 error	order
2^3	9^3	2.27E-1	-	11^3	4.76E-1	-
4^3	19^3	3.91E-3	5.86	21^3	5.49E-3	6.44
8^3	39^3	4.12E-5	6.57	41^3	4.32E-5	6.99
16^3	79^3	3.34E-7	6.95	81^3	3.42E-7	6.98
32^3	159^3	2.63E-9	6.99	161^3	2.67E-9	7.00
Q^6 spectral-element method (SEM)						
FEM Mesh	Dirichlet boundary			Neumann boundary		
	Total DoFs	ℓ^2 error	order	Total DoFs	ℓ^2 error	order
2^3	11^3	9.68E-2	-	13^3	1.18E-1	-
4^3	23^3	6.05E-4	7.32	25^3	8.42E-4	7.13
8^3	47^3	3.11E-6	7.60	49^3	3.24E-6	8.02
16^3	95^3	1.26E-8	7.95	97^3	1.28E-8	7.98
32^3	191^3	4.96E-11	7.98	193^3	5.09E-11	7.98

We consider the Poisson type equation (2.1) with $\alpha=1$ in domain $\Omega=[-1,1]^3$. For Dirichlet boundary conditions, we test a smooth exact solution

$$u_D^*(\boldsymbol{x})=\sin(\pi x)\sin(2\pi y)\sin(3\pi z)+(x-x^3)(y^2-y^4)(1-z^2).$$

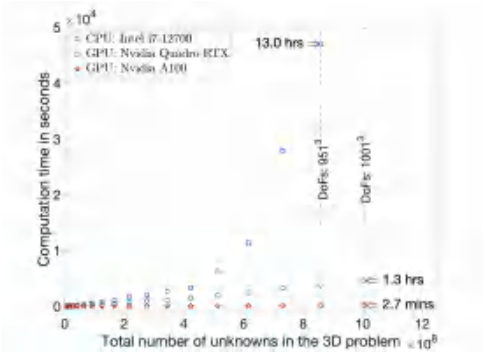
For Neumann boundary conditions, we test a smooth exact solution

$$u_N^*(\boldsymbol{x})=\cos(\pi x)\cos(2\pi y)\cos(3\pi z)+(1-x^2)^3(1-y^2)^2(1-z^2)^4.$$

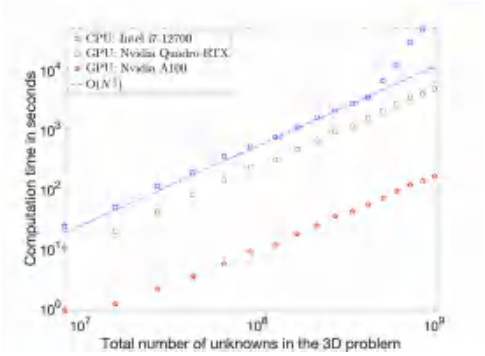
The results of Q^5 and Q^6 spectral-element methods are listed in Table 2.

3.2 GPU acceleration for solving a Poisson type equation

In this subsection, we list the online computational time comparison for solving $\alpha u-\Delta u=f$ on $\Omega=[-1,1]^3$ with $\alpha=1$ and Neumann boundary conditions, by using the Q^5 spectral-element method. To obtain a more accurate estimate of the online computational time, we count the online computation time for solving the Poisson equation 200 times. The results of online computational time, depicted in Fig. 2 and Table 3, demonstrate a speed-up factor of at least 60 for sufficiently large problems when comparing Nvidia A100 to Intel i7-12700. In particular, we observe that on the A100, solving a Poisson type equation (2.1) with a total degree of freedoms (DoFs) equal to 1001^3, takes approximately only 0.8 second.



(a) Comparison on three devices.



(b) Semilogx plot shows the complexity on all devices are $\mathcal{O}(N^{\frac{4}{3}})$ for problems with proper sizes.

Figure 2: Online computation time of Q^5 spectral-element method solving a 3D Poisson equation two hundred times. On the A100, it takes approximately only 0.8 second when solving one Poisson equation for the total number of DoFs being 1001^3.

Table 3: Online computation time of solving a 3D Poisson equation two hundred times on three devices: the time unit is second, and the speed-up is GPU versus CPU.

Total DoFs	Intel i7-12700	NVIDIA Quadro		NVIDIA A100	
	CPU time	GPU time	speed-up	GPU time	speed-up
201^3	2.29E1	1.03E1	2.23	9.00E-1	25.47
251^3	4.86E1	1.91E1	2.54	1.15E0	42.14
301^3	1.08E2	4.10E1	2.65	2.05E0	52.94
351^3	1.81E2	7.82E1	2.32	3.31E0	54.77
401^3	3.36E2	1.36E2	2.47	5.41E0	62.12
451^3	4.98E2	2.25E2	2.22	8.52E0	58.49
501^3	7.13E2	2.96E2	2.41	1.11E1	64.09
551^3	1.05E2	4.46E2	2.35	1.71E1	61.19
601^3	1.57E3	6.40E2	2.46	2.35E1	67.09
651^3	2.05E3	9.09E2	2.25	3.37E1	60.68
701^3	2.63E3	1.11E3	2.37	4.07E1	64.63
751^3	3.30E3	1.48E3	2.23	5.31E1	62.19
801^3	6.29E3	1.97E3	3.20	6.86E1	91.64
851^3	1.13E4	2.55E3	4.45	8.97E1	126.36
901^3	2.79E4	3.27E3	8.53	1.14E2	244.19
951^3	4.69E4	3.77E3	12.45	1.34E2	349.16

For completeness, in Table 4, we also include the offline preparation time which includes the time for generating arrays and loading arrays to GPU memory.

Table 4: Offline preparation time in MATLAB on GPU for solving a 3D Poisson equation: the time unit is second.

Total DoFs	200^3	250^3	300^3	350^3	400^3	450^3
Quadro	1.59E0	1.60E0	1.64E0	1.72E0	1.77E0	1.85E0
A100	3.01E-1	3.19E-1	3.41E-1	3.85E-1	4.36E-1	4.93E-1
Total DoFs	500^3	550^3	600^3	650^3	700^3	750^3
Quadro	1.92E0	2.01E0	2.16E0	2.42E0	2.46E0	2.49E0
A100	5.57E-1	6.30E-1	7.07E-1	8.41E-1	9.61E-1	1.12E0
Total DoFs	800^3	850^3	900^3	950^3	1000^3	1050^3
Quadro	2.52E0	2.82E0	3.08E0	3.40E0	3.70E0	4.37E0
A100	1.24E0	1.41E0	1.60E0	1.83E0	2.04E0	2.29E0

3.3 GPU acceleration for solving a Schrödinger equation

For a Problem with general variable coefficients, the tensor product structure of the eigenvectors no longer holds. Then, an efficient method for solving such problems is to use a preconditioned conjugate gradient method with the inverse of Poisson type equation as a preconditioner. As an example, we consider the following equation

$$\alpha u-\Delta u+V(\boldsymbol{x})u=f, \tag{3.1}$$

on $\Omega=[-16,16]^3$ with $\alpha=1$,

$$V(\boldsymbol{x})=\beta\sin\Big(\frac{\pi}{4}x\Big)^2\sin\Big(\frac{\pi}{4}y\Big)^2\sin\Big(\frac{\pi}{4}z\Big)^2,\quad \beta>0, \tag{3.2}$$

and an exact solution

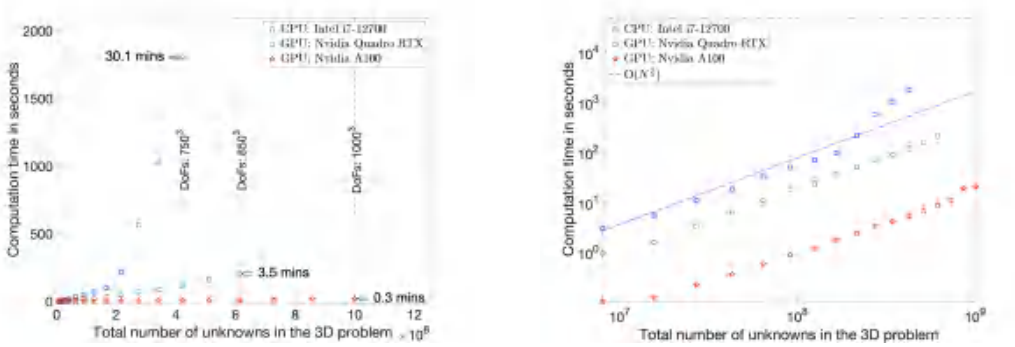
$$u(\boldsymbol{x})=\cos\Big(\frac{\pi}{16}x\Big)\cos\Big(\frac{\pi}{16}y\Big)\cos\Big(\frac{\pi}{16}z\Big). \tag{3.3}$$

Eq. (3.1) is sometimes referred to as a Schrödinger equation, which emerges in solving more complicated problems originated from the nonlinear Schrödinger equation, e.g., the Gross-Pitaevskii equation [5]. The boundary conditions can be either periodic or homogeneous Neumann.

Note that $0\le V(x)\le\beta$. We use $((\alpha+\frac{1}{2}\beta)I-\Delta)^{-1}$ as a preconditioner in the preconditioned conjugate gradient (PCG) method inverting the operator $\alpha I-\Delta+V(\mathbf{x})$ with periodic boundary conditions in the Q^5 spectral-element method, where $((\alpha+\frac{1}{2}\beta)I-\Delta)^{-1}$ is implemented in the same way as in Table 1. **We emphasize that eigenvectors can not be implemented by fast Fourier transform (FFT) for high order schemes with periodic boundary conditions, because the stiffness matrix S for Q^k SEM is a circulant matrix only when $k=1$, i.e., FFT can be used to invert Laplacian only for second order accurate schemes.**

Table 5: Number of PCG iterations needed for PCG with $((\alpha+\frac{1}{2}\beta)I-\Delta)^{-1}$ as the preconditioner to converge for solving a Schrödinger equation by the Q^5 SEM on different meshes with different $\beta=1,10,100,200,\cdots,10000$.

Total	Number of PCG iterations									
DoFs	$\beta=1$	10	100	200	400	800	1000	2000	4000	10000
250^3	10	35	85	112	149	191	214	288	388	535
350^3	10	32	81	108	143	184	202	265	348	522
450^3	10	30	80	105	142	181	191	252	333	467
550^3	10	28	76	104	129	174	183	239	321	448
650^3	10	27	77	100	135	169	182	248	327	465
750^3	10	27	76	100	130	173	192	233	320	456
850^3	10	27	76	100	134	173	188	233	305	430
950^3	10	25	72	98	128	165	180	234	305	428
1000^3	10	25	72	101	134	172	188	243	305	433



(a) Comparison on three devices.

(b) Semilogx plot shows the complexity on all devices are $\mathcal{O}(N^{\frac{4}{3}})$ for problems of proper sizes.

Figure 3: Online computational time of Q^5 SEM for a Schrödinger equation with $\beta=1$ in (3.2), solved by PCG with $(I-\Delta)^{-1}$ as the preconditioner.

Obviously, the performance of such a simple method depends on the condition number of the operator $\alpha I-\Delta+V(\mathbf{x})$, which is affected by the choice of $V(\mathbf{x})$. By choosing different β in (3.2), the performance of PCG, e.g., the number of PCG iterations needed for the PCG iteration residue to reach round-off errors, would vary. We first list the performance of PCG for the Q^5 spectral-element method on different meshes for different β in Table 5. We can observe that the performance only depends on $V(\boldsymbol{x})$ for a fine enough mesh.

The online computational time of using PCG for the Q^5 spectral-element method solving one Schrödinger equation with $\beta=1$ in (3.2) is listed in both Fig. 3 and Table 6. We can

Table 6: The online computational time (unit is second) of using PCG for the Q^5 SEM solving one Schrödinger equation with $\beta=1$ in (3.2).

Total DoFs	200^3	250^3	300^3	350^3	400^3	450^3
Intel i7-12700	2.99E0	5.50E0	1.09E1	1.81E1	3.27E1	4.83E1
Nvidia Quadro	9.48E-1	1.58E0	3.32E0	6.30E0	1.10E1	1.80E1
Nvidia A100	1.04E-1	1.23E-1	2.21E-1	3.61E-1	5.75E-1	8.86E-1
Total DoFs	500^3	550^3	600^3	650^3	700^3	750^3
Intel i7-12700	6.90E1	9.83E1	2.16E2	5.63E2	1.03E3	1.80E3
Nvidia Quadro	2.38E1	3.55E1	5.11E1	7.23E1	8.78E1	1.18E2
Nvidia A100	1.18E0	1.75E0	2.38E0	3.35E0	4.10E0	5.30E0
Total DoFs	800^3	850^3	900^3	950^3	1000^3	
Nvidia Quadro	1.57E2	2.09E2	-	-	-	
Nvidia A100	6.79E0	8.71E0	1.11E1	1.88E1	2.04E1	

observe a satisfying speed-up. With 10 PCG iterations, it costs about 20 seconds on A100 for inverting a 3D Schrödinger operator for a total number of DoFs as large as 1000^3.

3.4 Robustness of the implementation for very high order elements

For very high order elements, it is important to have a robust procedure for finding the eigenvalue decomposition of the matrix H. We test the implementation in Remark 2.3 for the Q^{20} spectral-element method solving the Schrödinger equation. The error in Table 7 and the online computational time in Table 8 validate the robustness of the implementation. In other words, even for Q^{20} element, the numerical computation of eigenvalue decomposition in Remark 2.3 is still accurate.

Table 7: The ℓ^∞ error for Q^{20} SEM solving one Schrödinger equation with different β in (3.2).

Total DoFs	ℓ^∞ error		
	$\beta=1$	$\beta=10$	$\beta=100$
500^3	1.89E-13	1.62E-13	1.29E-13
800^3	4.86E-13	4.09E-13	2.97E-13
1000^3	6.28E-13	5.23E-13	3.76E-13

3.5 Comparison with FFT on GPU

It is also interesting to compare the implementation in Table 1 with the performance of fast Fourier transform (FFT) on GPU. In order to do so, we consider solving the Poisson type equation(2.1) and the Schrödinger equation (3.9) with periodic boundary conditions

Table 8: Online computational time in seconds for Q^{20} SEM solving one Schrödinger equation with different β in (3.2).

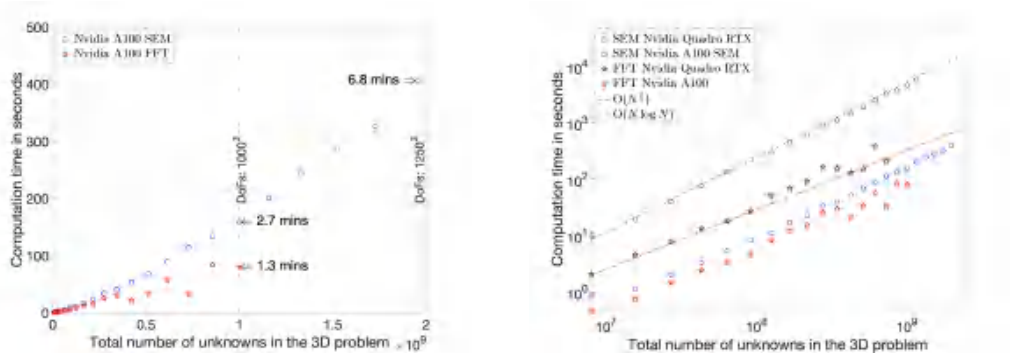
Total DoFs	$\beta=1$		$\beta=10$		$\beta=100$	
	Time	# PCG	Time	# PCG	Time	# PCG
500^3	1.23E0	10	2.36E0	23	6.47E0	68
800^3	7.13E0	11	3.15E1	56	7.84E1	142
1000^3	2.04E1	10	4.40E1	23	1.30E2	70

using second order finite difference, or equivalently the Q^1 spectral-element method, for which the discrete Laplacian can be diagonalized by FFT, e.g., the eigenvector matrices T^{-1} in (2.14) is the discrete Fourier transform matrix. In other words, for Q^1 element with periodic boundary, the implementation in Table 1 can be replaced by the following implementation (Table 9) via FFT in MATLAB:

Table 9: The FFT implementation of a second order scheme for the Poisson equation with periodic boundary conditions.

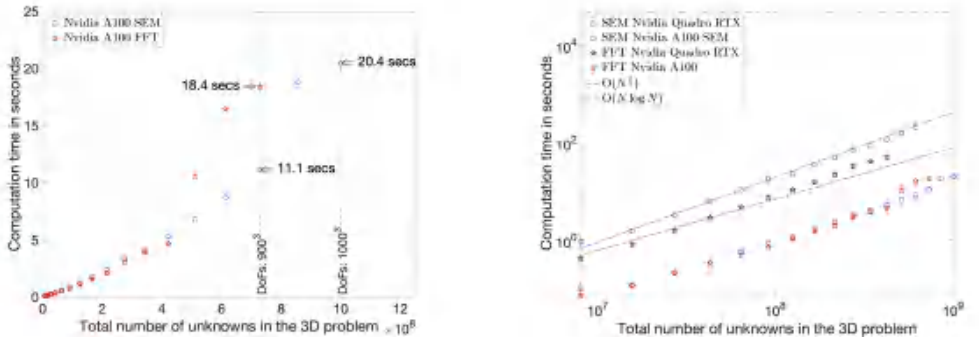
```
U = fftn(F);
U = U./Lambda3D;
U = real(ifftn(U));
```

We will refer to such an implementation for a second order scheme as FFT in Fig. 4 and Fig. 5. On the other hand, even if the Poisson equation has periodic boundary conditions, the matrices T and T^{-1} for high order elements cannot be implemented by FFT.



(a) Comparison on one device. (b) Semilogx plot shows the complexity.

Figure 4: Comparison between Q^5 SEM implemented in Table 1 and a second order scheme implemented by FFT in Table 9, for solving a Poisson equation 200 times. On A100, the FFT implementation cannot solve a problem of size 1050^3 in MATLAB 2023, due to the larger memory cost of FFT.



(a) Comparison on one device. (b) Semilogx plot shows the complexity.

Figure 5: Comparison between Q^5 SEM implemented in Table 1 and a second order scheme implemented by FFT in Table 9 for solving a Schrödinger equation by PCG. On A100, the FFT implementation cannot solve a problem of size 950^3 in MATLAB 2023, due to the larger memory cost of FFT.

We simply refer to the implementation in Table 1 for high order elements as SEM in Fig. 4 and Fig. 5. The detailed comparison is listed in Fig. 4 and Fig. 5, as well as Table 10 and Table 11. We can observe that FFT is faster as expected, on most meshes. However, the memory cost of performing FFT is more demanding, especially on finer meshes. For the Schrödinger problem, the performance of FFT deteriorates on finest meshes.

3.6 A Cahn–Hilliard equation

We consider solving the Cahn–Hilliard equation [1], which is not only a fourth-order equation in space, but also incorporates a time derivative. Consider a domain $\Omega=[-1,1]^3$ with its boundary denoted as $\partial\Omega$. Within this domain, the Cahn–Hilliard equation with simple boundary conditions is given by

$$\begin{cases} \phi_t = m\Delta\left(-\epsilon\Delta\phi+\frac{1}{\epsilon}F'(\phi)\right) & \text{in } \Omega, \\ \partial_{\boldsymbol{n}}\phi=0, \quad \partial_{\boldsymbol{n}}\Delta\phi=0 & \text{on } \partial\Omega, \end{cases} \tag{3.4}$$

where ϕ is a phase function with a thin, smooth transitional layer, whose thickness is proportional to the parameter ϵ, m is the mobility constant, and $F(\phi)=\frac{1}{4}(\phi^2-1)^2$ is a double-well form function.

Due to the simplicity of the boundary conditions, we can avoid solving a fourth-order equation directly by reformulating (3.4) as a system of second-order equations after introducing the chemical potential μ, which can be expressed as the variational derivative of the energy functional:

$$E(\phi)=\int_\Omega \frac{\epsilon}{2}|\nabla\phi|^2+\frac{1}{\epsilon}F(\phi)d\boldsymbol{x}. \tag{3.5}$$

Table 10: Online computational time comparison between Q^5 SEM implemented in Table 1 and a second order scheme implemented by FFT in Table 9, for solving a Poisson equation 200 times. On A100, the FFT implementation cannot solve a problem of size 1050^3 in MATLAB 2023, due to the larger memory cost of FFT. Unit is in seconds.

Total DoFs	200^3	250^3	300^3	350^3	400^3	450^3
Quadro (SEM)	1.03E1	1.91E1	4.10E1	7.82E1	1.36E2	2.25E2
Quadro (FFT)	2.06E0	4.61E0	7.82E0	1.33E1	1.83E1	2.70E1
A100 (SEM)	9.00E-1	1.15E0	2.05E0	3.31E0	5.41E0	8.52E0
A100 (FFT)	4.58E-1	7.50E-1	1.49E0	2.44E0	3.37E0	4.52E0
Total DoFs	500^3	550^3	600^3	650^3	700^3	750^3
Quadro (SEM)	2.96E2	4.46E2	6.40E2	9.09E2	1.11E3	1.48E3
Quadro (FFT)	5.19E1	6.89E1	9.12E1	1.64E2	1.57E2	1.31E2
A100 (SEM)	1.11E1	1.71E1	2.35E1	3.37E1	4.07E1	5.31E1
A100 (FFT)	8.22E0	1.21E1	1.56E1	2.55E1	3.02E1	2.17E1
Total DoFs	800^3	850^3	900^3	950^3	1000^3	1050^3
Quadro (SEM)	1.97E3	2.55E3	3.27E3	3.77E3	4.656E3	5.79E3
Quadro (FFT)	1.52E2	3.82E2	2.13E2	-	-	-
A100 (SEM)	6.86E1	8.97E1	1.14E2	1.34E2	1.59E2	2.01E2
A100 (FFT)	3.38E1	5.76E1	3.32E1	8.43E1	7.97E1	-
Total DoFs	1100^3	1150^3	1200^3	1250^3	1300^3	1350^3
A100 (SEM)	2.46E2	2.85E2	3.27E2	4.06E2	-	-

Then, the system can be derived as

$$\begin{cases} \phi_t - m\Delta\mu = 0 & \text{in } \Omega, \\ \mu = -\epsilon\Delta\phi + \frac{1}{\epsilon}F'(\phi) & \text{in } \Omega, \\ \partial_{\boldsymbol{n}}\phi = 0, \quad \partial_{\boldsymbol{n}}\mu = 0 & \text{on } \partial\Omega. \end{cases} \tag{3.6}$$

For the space discretization, we use Q^5 spectral-element method. For time discretization, we implement the second order backward differentiation formula (BDF-2) to the system (3.6):

$$\begin{cases} \frac{a\phi_{n+1}-\hat{\phi}_n}{\delta t} - m\Delta\mu_{n+1} = 0, \\ \mu_{n+1} = -\epsilon\Delta\phi_{n+1} + \frac{1}{\epsilon}F'(\bar{\phi}_n), \end{cases} \tag{3.7}$$

where $a = \frac{3}{2}$, $\hat{\phi}_n = 2\phi_n - \frac{1}{2}\phi_{n-1}$, and $\bar{\phi}_n = 2\phi_n - \phi_{n-1}$. To solve this linear system, we can write it as

$$\begin{bmatrix} \alpha I & -m\delta t\Delta \\ \epsilon\Delta & I \end{bmatrix} \begin{bmatrix} \phi \\ \mu \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, \tag{3.8}$$

Table 11: Online computational time comparison between Q^5 SEM implemented in Table 1 and a second order scheme implemented by FFT in Table 9 for solving a Schrödinger equation by PCG. On A100, the FFT implementation cannot solve a problem of size 950^3 in MATLAB 2023, due to the larger memory cost of complex numbers. Unit is in seconds.

Total DoFs	200^3	250^3	300^3	350^3	400^3	450^3
Quadro (SEM)	9.48E-1	1.58E0	3.32E0	6.30E0	1.10E1	1.80E1
Quadro (FFT)	4.25E-1	8.22E-1	1.60E0	2.91E0	4.76E0	7.60E0
A100 (SEM)	1.04E-1	1.23E-1	2.21E-1	3.61E-1	5.75E-1	8.86E-1
A100 (FFT)	7.55E-2	1.19E-1	2.13E-1	3.41E-1	5.02E-1	7.17E-1
Total DoFs	500^3	550^3	600^3	650^3	700^3	750^3
Quadro (SEM)	2.38E1	3.55E1	5.11E1	7.23E1	8.78E1	1.18E2
Quadro (FFT)	1.09E1	1.58E1	2.24E1	3.36E1	4.16E1	5.14E1
A100 (SEM)	1.18E0	1.75E0	2.38E0	3.35E0	4.10E0	5.30E0
A100 (FFT)	1.09E0	1.57E0	2.05E0	3.04E0	3.89E0	4.66E0
Total DoFs	800^3	850^3	900^3	950^3	1000^3	1050^3
Quadro (SEM)	1.57E2	2.09E2	-	-	-	-
Quadro (FFT)	-	-	-	-	-	-
A100 (SEM)	6.79E0	8.71E0	1.11E1	1.88E1	2.04E1	-
A100 (FFT)	1.05E1	1.64E1	1.84E1	-	-	-

and its solution is given by

$$\begin{bmatrix}\phi\\ \mu\end{bmatrix}=\begin{bmatrix}\mathcal{D}I & \mathcal{D}(m\delta t\Delta)\\ \mathcal{D}(-\epsilon\Delta) & \alpha\mathcal{D}I\end{bmatrix}\begin{bmatrix}f_1\\ f_2\end{bmatrix}=\begin{bmatrix}\mathcal{D}(f_1+m\delta t\Delta f_2)\\ \mathcal{D}(-\epsilon\Delta f_1+\alpha f_2)\end{bmatrix}, \tag{3.9}$$

where $\mathcal{D}=(\alpha I+m\delta t\epsilon\Delta^2)^{-1}$.

Notice that μ and ϕ are already decoupled in (3.9). Thus for implementing the scheme (3.7), we only need to compute ϕ without computing μ:

$$\phi_{n+1}=\mathcal{D}\hat{\phi}_n+m\delta t\frac{1}{\epsilon}\mathcal{D}\Delta F'(\bar{\phi}_n), \tag{3.10}$$

where both $\mathcal{D}=(\alpha I+m\delta t\epsilon\Delta^2)^{-1}$ and $\mathcal{D}\Delta=(\alpha I+m\delta t\epsilon\Delta^2)^{-1}\Delta$ can be implemented in the same way as shown in Table 1.

Since $(\alpha I+m\delta t\epsilon\Delta^2)^{-1}$ and $(\alpha I+m\delta t\epsilon\Delta^2)^{-1}\Delta$ share the same eigenvectors, the implementation of (3.10) costs slightly less than solving the Poisson type equation twice. In Table 3, we observe that, the average online computational time of inverting Laplacian once is approximately 0.8 second for the number DoFs being 1001^3. For the same mesh and same DoFs, each time step (3.10) of solving the Cahn–Hilliard equation costs about 1.27 seconds in Table 12.

3.6.1 Accuracy test

We first use a manufactured analytical solution of the Cahn–Hilliard equation to validate the convergence rate of the BDF-2 scheme (3.7). This solution is in the domain $\Omega=[-1,1]^3$ with $\epsilon=0.2$, $m=0.01$:

$$\phi^*(\boldsymbol{x})=\cos(\pi x)\cos(\pi y)\cos(\pi z)\exp(t), \tag{3.11}$$

and the corresponding forcing term can be obtained from Eq. (3.6). We fix the number of basis function as $N_x=N_y=N_z=51$ in Q^5 SEM so that the spatial error is negligible compared with the time discretization error. Fig. 6 shows that the scheme (3.7) achieves the expected second order time accuracy.

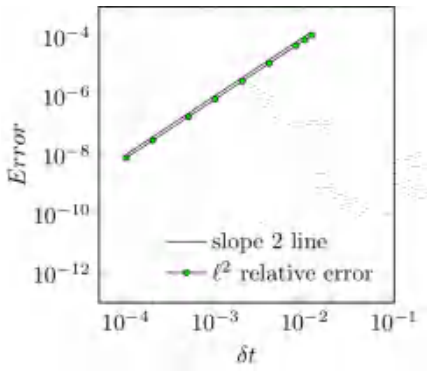


Figure 6: The ℓ^2 relative error of BDF2 scheme (3.10) for the Cahn–Hilliard equation.

3.6.2 Coalescence of two drops

We now study the coalescence of two droplets, as described by the Cahn–Hilliard equation, within the computational domain $\Omega=[-1,1]^3$. Drawing from parameter settings in [2], we select $\epsilon=0.02$, the mobility constant $m=0.02$, and the time step size $\delta t=0.001$ with an end time $T=10$. For stable computation, we use the same simple stabilization method and stabilization parameter as in [2]. Initially, at time $t=0$, the domain is occupied by two neighboring spherical regions of the first material, while the second material fills the remaining space. As time progresses under the Cahn–Hilliard dynamics, these two spherical regions coalesce to form a singular droplet. More specifically, the initial condition for the phase function is given by

$$\phi_0(\boldsymbol{x})=1-\tanh\frac{|\boldsymbol{x}-\boldsymbol{x}_1|-R}{\sqrt{2}\epsilon}-\tanh\frac{|\boldsymbol{x}-\boldsymbol{x}_2|-R}{\sqrt{2}\epsilon}, \tag{3.12}$$

where $\boldsymbol{x}_1=(x_1,y_1,z_1)=(0,0,0.37)$ and $\boldsymbol{x}_2=(x_2,y_2,z_2)=(0,0,-0.37)$ are the centers of the initial spherical regions of the first material, and $R=0.35$ is the radius of these spheres.

Owing to the mass conservation and energy dissipation of the system (3.6), the energy $E(\phi)$ first decreases before stabilizing at a constant value, as shown in Fig. 7. The

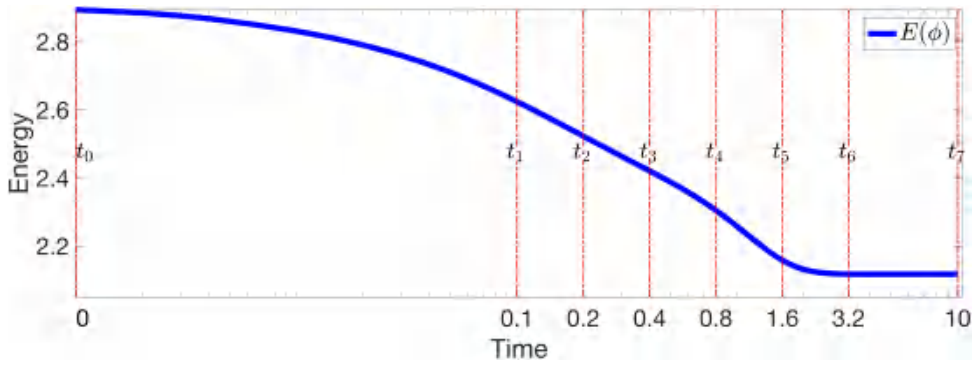


Figure 7: Semilogx plot shows the temporal evolution of energy $E(\phi)$.

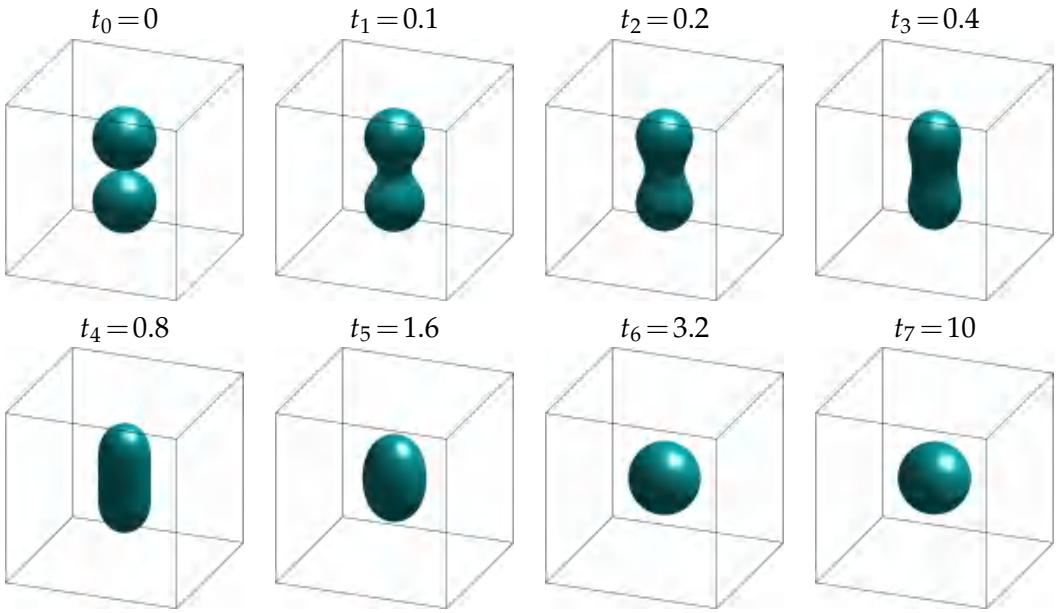


Figure 8: Snapshots of the zero-isocontour of the phase function ϕ show the coalescence of two drops at different time instants as indicated. See Table 12 for the computational time.

coalescence dynamics of the two droplets is illustrated through a series of temporal snapshots in Fig. 8. These snapshots capture the evolving interfaces between the materials, visualized by the level set of $\phi=0$. In Table 12, we enumerate the online computational costs associated with various total DoFs. As explained above, the online computational time at each time step is less than solving two Poisson equations.

Table 12: Online computational time in seconds for Q^5 SEM in the BDF2 scheme (3.10) solving the Cahn–Hilliard equation with 10,000 time steps for computing the solution at $T=10$ on Nivida A100. The *Total time* represents the online computational time for 10,000 time steps, and the *Time for each time step* is average online computational time per time step.

Total DoFs	501^3	551^3	601^3	651^3	701^3	751^3
Total time	8.46E2	1.29E3	1.79E3	2.53E3	3.10E3	4.02E4
Time for each time step	8.46E-2	1.29E-1	1.79E-1	2.53E-1	3.10E-1	4.02E-1
Total DoFs	801^3	851^3	901^3	951^3	1001^3	1051^3
Total time	5.41E3	6.79E3	8.84E3	1.03E4	1.27E4	-
Time for each time step	5.41E-1	6.79E-1	8.84E-1	1.03E0	1.27E0	-

3.7 Comparison with implementation in Python

For implementing (2.14) on both CPU and GPU, similar to the implementation in MATLAB shown in Table 1, (2.14) can be efficiently implemented using the function *jax.numpy.einsum* in the Python package JAX as shown in Table 13.

Table 13: The Python script of implementing (14) on both CPU and GPU where `jnp` means `jax.numpy`.

```
u = jnp.einsum('ijk,kl->ijl',f,invTz.transpose())
u = jnp.einsum('ijk,jl->ilk',u,invTy.transpose())
u = jnp.einsum('li,ijk->ljk',invTx,u)
u = u/Eig3D
u = jnp.einsum('ijk,kl->ijl',u,Tz.transpose())
u = jnp.einsum('ijk,jl->ilk',u,Ty.transpose())
u = jnp.einsum('li,ijk->ljk',Tx,u)
```

Since both Python and MATLAB allow similar simple implementations of (2.14) on GPU, it is interesting to compare them. We compare the performance of MATLAB with Python under double precision, as well as single precision, which often depends on specific hardware and their driver versions.

In Table 14, we list the online computational time comparison of similar implementations in MATLAB and Python on A100 for solving a 3D Poisson equation 200 times. As we can see in Table 14, for double precision computation and problems with size smaller than 1000^3, there is no significant difference in the online computational time between MATLAB and Python on GPUs. However, on A100 with 80G memory, MATLAB allows a problem size as large as 1250^3, for which Python can handle only with single precision computation. It is noteworthy that the performance of single precision computation in Python can be significantly affected by the use of the TF32 (TensorFloat-32) format on NVIDIA GPUs. TF32 is a lower-precision format that requires less memory bandwidth and compute resources compared to the traditional FP32 (Float-32) format, leading to faster computation at the cost of some loss in precision.

Table 14: Online computational time of single precision and double precision on one Nvidia A100 80G GPU card, for Q^5 SEM for solving a 3D Poisson equation 200 times. The time unit is second. For double precision computation in Python on A100, an out-of-memory error will emerge for problems with size larger than 950^3.

Total DoFs	Python(JAX)			MATLAB	
	Single(TF32)	Single(FP32)	Double	Single	Double
200^3	4.80E-1	6.88E-1	6.70E-1	4.72E-1	5.20E-1
250^3	5.85E-1	9.92E-1	1.17E0	7.32E-1	9.11E-1
300^3	7.43E-1	1.67E0	2.10E0	1.54E0	1.80E0
350^3	1.39E0	2.80E0	3.47E0	2.56E0	3.04E0
400^3	1.51E0	4.56E0	5.47E0	4.63E0	5.07E0
450^3	2.89E0	7.24E0	8.74E0	7.05E0	8.09E0
500^3	2.92E0	9.48E0	1.20E1	8.92E0	1.07E1
550^3	6.07E0	1.41E1	1.82E1	1.45E1	1.65E1
600^3	5.41E0	1.99E1	2.40E1	1.97E1	2.28E1
650^3	1.02E1	2.80E1	3.35E1	3.02E1	3.32E1
700^3	9.76E0	3.41E1	4.24E1	3.55E1	4.04E1
750^3	1.66E1	4.48E1	5.66E1	4.57E1	5.22E1
800^3	1.45E1	6.12E1	7.14E1	6.32E1	6.89E1
850^3	2.66E1	7.92E1	9.28E1	8.01E1	9.05E1
900^3	2.37E1	1.00E2	1.15E2	1.05E2	1.15E2
950^3	4.10E1	1.17E2	1.37E2	1.20E2	1.35E2
1000^3	3.17E1	1.40E2	-	1.40E2	1.60E2
1050^3	6.12E1	1.81E2	-	1.87E2	2.04E2
1100^3	4.77E1	2.13E2	-	2.16E2	2.51E2
1150^3	7.96E1	2.37E2	-	2.39E2	2.91E2
1200^3	6.30E1	2.94E2	-	2.95E2	3.34E2
1250^3	1.13E2	3.36E2	-	3.45E2	4.13E2

As shown in Table 14, for larger problems such as one billion DoFs, the fastest implementation on GPU is Python with single precision computation, which might be suitable for some practical simulations. When using the default TF32 setting on A100, the accuracy of the Q^5 spectral-element method deteriorates for larger problem sizes, with the order of accuracy dropping below the expected value. However, by setting the environment variable `NVIDIA_TF32_OVERRIDE=0` to disable TF32 precision for certain operations, such as matrix multiplication and `jax.numpy.einsum`, the accuracy can be improved, and the expected order of accuracy is maintained. To investigate the impact of TF32 on the accuracy of single precision computation in Python, we conducted additional tests with and without the TF32 format. Table 15 presents the accuracy results for the Q^5 spectral-element method under single precision on A100 with both TF32 and FP32 format. In

Table 15: Accuracy tests under TF32 and FP32 single precision in Python on Nvidia GPU A100 for the 3D Poisson equation (2.1) with $\alpha=1$. The actual accuracy of single precision computation depends very much on the hardware and version of hardware drivers. For Python, we implement the code under the environment JAX version 0.4.19 for Nvidia GPU A100, with Driver Version 535.86.10 and CUDA Version 12.2. See Table 2 for the results of MATLAB with double precision on Nvidia GPU A100.

Q^5 spectral-element method (TF32 single precision)						
FEM Mesh	Dirichlet boundary			Neumann boundary		
	Total DoFs	ℓ^2 error	order	Total DoFs	ℓ^2 error	order
2^3	9^3	2.27E-1	-	11^3	4.82E-1	-
4^3	19^3	3.92E-3	5.86	21^3	6.60E-3	6.19
8^3	39^3	4.12E-5	6.57	41^3	4.32E-5	7.26
16^3	79^3	1.44E-3	-5.13	81^3	2.46E-3	-5.83
32^3	159^3	1.95E-3	-0.44	161^3	2.73E-3	-0.15
Q^5 spectral-element method (FP32 single precision)						
FEM Mesh	Dirichlet boundary			Neumann boundary		
	Total DoFs	ℓ^2 error	order	Total DoFs	ℓ^2 error	order
2^3	9^3	2.27E-1	-	11^3	4.76E-1	-
4^3	19^3	3.91E-3	5.86	21^3	5.49E-3	6.44
8^3	39^3	4.11E-5	6.57	41^3	4.32E-5	6.99
16^3	79^3	1.67E-6	4.62	81^3	1.63E-6	4.72
32^3	159^3	1.34E-6	0.32	161^3	1.95E-6	-0.26

general, the implementation for high order SEM with TF32 single precision is not robust on A100, e.g., computation with SEM for the problem in Fig. 8 might blow up.

On the other hand, the second order finite difference implemented in Python Jax with TF32 single precision computation is robust as suggested by Table 16. For periodic boundary conditions, the eigenvectors of second order finite difference (i.e., Q^1 spectral-element method) can be implemented by FFT as shown in Table 17. As a demonstration, we include the computation results for the Cahn-Hilliard equation of Python in TF32 single precision on A100 in Fig. 9, which is comparable to the double precision results on A100 in Fig. 8.

4 Concluding remarks

In this paper, we have discussed a simple MATLAB 2023 implementation for accelerating high order methods on GPUs. For large enough 3D problems, a speed-up of at least 60 can be achieved on Nvidia A100. In particular, solving a 3D Poisson type equation with one billion DoFs costs only 0.8 second for Q^k spectral-element method. As examples of applications, we applied this fast solver to solve a linear (time-independent) Schrödinger equation and a nonlinear (time-dependent) Cahn-Hilliard equation in three-dimension.

Table 16: Accuracy tests for second order finite difference with periodic boundary (FFT implementation) in Python on Nvidia GPU A100 for the 3D Poisson equation (2.1) with $u^*=\sin(2\pi x)\sin(3\pi y)\sin(4\pi z)$ and $\alpha=1$.

FFT implementation on A100 for periodic boundary				
Total DoFs	TF32 Single precision		Double precision	
	ℓ^2 error	order	ℓ^2 error	order
10^3	5.00E-1	-	5.00E-1	-
20^3	1.05E-1	2.25	1.05E-1	2.25
40^3	2.53E-2	2.06	2.53E-2	2.06
80^3	6.26E-3	2.01	6.26E-3	2.01
160^3	1.56E-3	2.01	1.56E-3	2.00
320^3	3.88E-4	2.00	3.90E-4	2.00
640^3	8.57E-5	2.18	9.75E-5	2.00
900^3	5.64E-5	1.23	4.93E-5	2.00
1200^3	9.32E-5	-1.75	-	-

Table 17: The Python script for FFT implementation of a second order (i.e., Q^1 spectral-element method) for the Poisson equation with periodic boundary conditions on both CPU and GPU where `jnp` means `jax.numpy`.

```
    u = jnp.fft.fftn(f)/Eig3D
    if alpha == 0:
        u[0,0,0] = 0.
    u = jnp.real(jnp.fft.ifftn(u))
```

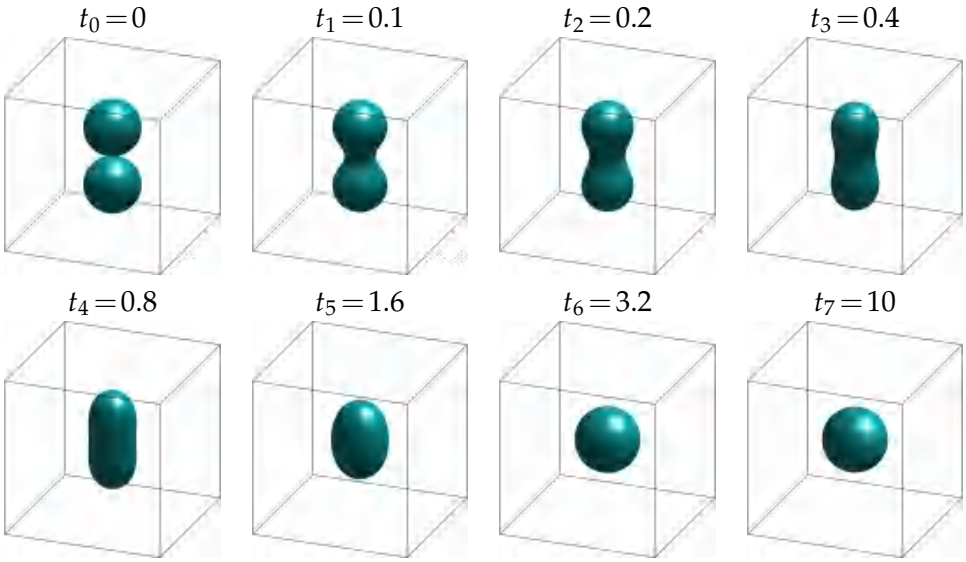


Figure 9: Snapshots of the same problem in Fig. 8, implemented by Python in TF32 single precision on A100 for second order finite difference (i.e., Q^1 spectral-element method) by FFT with total DoFs 800^3.

We expect the proposed simple implementation to have the same performance for any problem with similar tensor product structure, e.g., exponential time differencing and spectral fractional Laplacians.

Data availability statements

The authors declare that the data supporting the findings of this study are available within the paper and its supplementary information files.

Acknowledgments

J. Shen's research was supported in part by NSFC 12371409, and X. Zhang's research was supported by NSF DMS-220815. The authors declare they have no financial interests.

A Appendix: MATLAB scripts for a 3D Poisson equation

We provide a demonstration in MATLAB 2023 for Q^k spectral-element method solving a Poisson equation in three dimensions, which involves three MATLAB scripts:

1. *Poisson3D.m* for solving the Poisson equation on either CPU or GPU;
2. *SEGenerator1D.m* for generating stiffness and mass matrices in spectral element method;
3. *LegendreD.m* for Legendre and Jocaboi polynomials from [9].

Readers can easily reproduce the results in Section 3.1 and Section 3.2 using these three MATLAB scripts.

Poisson3D.m:

```
% Solving the Poisson equation by Q5 SEM with Neumann b.c.
if gpuDeviceCount('available')<1; Param.device='cpu';
else;Param.device='gpu'; Param.deviceID=1;end % ID=1,2,3,...
Np=5; Param.Np=Np; % polynomial degree Q5
Ncellx=40; Ncelly=40; Ncellz=40; % finite element cell number
% total number of unknowns in each direction
nx=Ncellx*Np+1; ny=Ncelly*Np+1; nz=Ncellz*Np+1;
% the domain is [-Lx, Lx]*[-Ly, Ly]*[-Lz, Lz]
Lx=1; Ly=1; Lz=1; cx=pi; cy=2*pi; cz=3*pi; alpha=1;
Param.Ncellx=Ncellx; Param.Ncelly=Ncelly; Param.Ncellz=Ncellz;
Param.nx = nx; Param.ny = ny; Param.nz = nz;
fprintf('3D Poisson with total DoFs %d by %d by %d \n',nx,ny,nz);
fprintf('Laplacian is Q%d spectral element method \n', Np);
[x,ex,Tx,eigx]=SEGenerator1D('x',Lx,Param);
[y,ey,Ty,eigy]=SEGenerator1D('y',Ly,Param);
[z,ez,Tz,eigz]=SEGenerator1D('z',Lz,Param);
% a smooth solution
u1x=cos(cx*x); u2x=power(1-power(x,2),3); du2x=30*power(x,4)-36*power(x,2)+6;
u1y=cos(cy*y); u2y=power(1-power(y,2),2); du2y=4-12*power(y,2);
u1z=cos(cz*z); u2z=power(1-power(z,2),4);
du2z=(8-56*power(z,2)).*power(1-power(z,2),2);
uexact=squeeze(tensorprod(u1x*u1y',u1z)+tensorprod(u2x*u2y',u2z));
f=(cx*cx+cy*cy+cz*cz)*squeeze(tensorprod(u1x*u1y',u1z))+...
  squeeze(tensorprod(du2x*u2y',u2z)+tensorprod(u2x*du2y',u2z)+...
  tensorprod(u2x*u2y',du2z))+alpha*uexact;
TxInv=pinv(Tx); TyInv=pinv(Ty); TzInv=pinv(Tz);
if strcmp(Param.device,'gpu');Device=gpuDevice(Param.deviceID);
   fprintf('GPU computation: starting to load matrices/data \n');
   Tx=gpuArray(Tx); Ty=gpuArray(Ty); Tz=gpuArray(Tz);
   eigx=gpuArray(eigx);eigy=gpuArray(eigy);eigz=gpuArray(eigz);
   ex=gpuArray(ex); ey=gpuArray(ey); ez=gpuArray(ez); f=gpuArray(f);
   TxInv=gpuArray(TxInv);TyInv=gpuArray(TyInv);TzInv=gpuArray(TzInv);
end
Lambda3D=squeeze(tensorprod(eigx,ey*ez')+tensorprod(ex,eigy*ez')...
        +tensorprod(ex,ey*eigz'));
if strcmp(Param.device,'gpu'); wait(Device);
    fprintf('GPU loading finished and computing started \n');
end
tic; % online computation
u = tensorprod(f,TzInv',3,1); u = pagemtimes(u,TyInv');
u = squeeze(tensorprod(TxInv,u,2,1)); u = u./(Lambda3D + alpha);
u = tensorprod(u,Tz',3,1); u = pagemtimes(u,Ty');
u = squeeze(tensorprod(Tx,u,2,1));
if strcmp(Param.device,'gpu');wait(Device);end;time=toc;err=u-uexact;
fprintf('The ell infinity norm error is %d \n',norm(err(:),inf));
if strcmp(Param.device,'gpu')
   fprintf('The online GPU computation time is %d \n', time);
else
   fprintf('The online CPU computation time is %d \n', time);
end
```

SEGenerator1D.m:

```
function [varargout] = SEGenerator1D(direction,L,Param)
% generate 1D spectral element with Neumann B.C.
switch direction
    case 'x'
        N=Param.Np; Ncell=Param.Ncellx; n=Param.nx;
    case 'y'
        N=Param.Np; Ncell=Param.Ncelly; n=Param.ny;
    case 'z'
        N=Param.Np; Ncell=Param.Ncellz; n=Param.nz;
end
% generate the mesh with Ncell intervals with domain [Left, Right]
[D,r,w] = LegendreD(N); Left = -L; Right = L;
Length = Right - Left; dx = Length/Ncell;
for j = 1:Ncell
    cellLeft = Left+dx*(j-1);
    localPoints = cellLeft+dx/2+r*dx/2;
    if (j==1)
    x = localPoints;
    else
    x = [x;localPoints(2:end)];
    end
end
SLocal = D'*diag(w)*D; % local stiffness matrix for each element
S=[]; M=[];
for j = 1:Ncell % global stiffness and lumped mass matrices
    S = blkdiag(SLocal,S); M = blkdiag(diag(w),M);
end
% Next step: "glue" the cells
Np = N+1; % number of points in each cell
Glue = sparse(zeros(Ncell*Np-Ncell+1, Ncell*Np));
for j = 1:Ncell
    rowStart=(j-1)*Np+2-j; rowEnd=rowStart+Np-1;
    colStart=(j-1)*Np+1; colEnd=colStart+Np-1;
    Glue(rowStart:rowEnd,colStart:colEnd)=speye(Np);
end
S=Glue*S*Glue'; M=Glue*M*Glue'; H=diag(1./diag(M))*S;
ex=ones(n,1); MHalfInv=diag(1./sqrt(diag(M)));
S1=MHalfInv*S*MHalfInv; S1=(S1+S1')/2;
[U,d]=eig(S1,'vector'); [lambda,indexSort]=sort(d);
T=U(:,indexSort); h=dx/2; lambda=lambda/(h*h);
S1=sparse(S1/(h*h)); M=sparse(M);
% after this step, T is the eigenvector of H
T = MHalfInv*T; H=full(H/(h*h)); S=S/h; M=full(M*h);
varargout{1}=x; varargout{2}=ex; varargout{3}=T; varargout{4}=lambda;
end
```

LegendreD.m:

```
function [D,r,w] = LegendreD(N)
    Np = N+1; r = JacobiGL(0,0,N);
    w = (2*N+1)/(N*N+N)./power(JacobiP(r,0,0,N),2);
    Distance = r*ones(1,N+1)-ones(N+1,1)*r'+eye(N+1);
    omega = prod(Distance,2); D = diag(omega)*(1./Distance)*diag(1./omega);
    D(1:Np+1:end) = 0; D(1:Np+1:end) = -sum(D,2);
end
function [x] = JacobiGL(alpha,beta,N)
    x = zeros(N+1,1);
    if (N==1); x(1)=-1.0; x(2)=1.0; return; end
    [xint,temp] = JacobiGQ(alpha+1,beta+1,N-2);
    x = [-1, xint', 1]'; return;
end
function [x,w] = JacobiGQ(alpha,beta,N)
    if (N==0)
        x(1) = -(alpha-beta)/(alpha+beta+2); w(1) = 2; return;
    end
    h1 = 2*(0:N)+alpha+beta;
    J = diag(-1/2*(alpha*alpha-beta*beta)./(h1+2)./h1) + ...
        diag(2./(h1(1:N)+2).*sqrt((1:N).*((1:N)+alpha+beta).*...
        ((1:N)+alpha).*((1:N)+beta)./(h1(1:N)+1)./(h1(1:N)+3)),1);
    if (alpha+beta<10*eps); J(1,1)=0.0; end
    J = J + J'; [V,D] = eig(J); x = diag(D);
    w = power(V(1,:)',2)*power(2,alpha+beta+1)/(alpha+beta+1)*...
        gamma(alpha+1)*gamma(beta+1)/gamma(alpha+beta+1);
end
function [P] = JacobiP(x,alpha,beta,N)
    xp = x; dims = size(xp);
    if (dims(2)==1); xp = xp'; end
    PL = zeros(N+1,length(xp));
    gamma0 = power(2,alpha+beta+1)/(alpha+beta+1)*gamma(alpha+1)*...
        gamma(beta+1)/gamma(alpha+beta+1);
    PL(1,:) = 1.0/sqrt(gamma0);
    if (N==0); P = PL'; return; end
    gamma1 = (alpha+1)*(beta+1)/(alpha+beta+3)*gamma0;
    PL(2,:) = ((alpha+beta+2)*xp/2 + (alpha-beta)/2)/sqrt(gamma1);
    if (N==1); P = PL(N+1,:)'; return; end
    aold = 2/(2+alpha+beta)*sqrt((alpha+1)*(beta+1)/(alpha+beta+3));
    for i = 1:N-1
      h1 = 2*i+alpha+beta;
      anew = 2/(h1+2)*sqrt( (i+1)*(i+1+alpha+beta)*(i+1+alpha)*...
          (i+1+beta)/(h1+1)/(h1+3));
      bnew = - (alpha*alpha-beta*beta)/h1/(h1+2);
      PL(i+2,:) = 1/anew*( -aold*PL(i,:) + (xp-bnew).*PL(i+1,:));
      aold = anew;
    end
    P = PL(N+1,:)';
end
```

References

[1] John W Cahn and John E Hilliard. Free energy of a nonuniform system. I. Interfacial free energy. *The Journal of Chemical Physics*, 28(2):258–267, 1958.

[2] Feng Chen and Jie Shen. Efficient spectral-Galerkin methods for systems of coupled second-order equations and their applications. *Journal of Computational Physics*, 231(15):5016–5028, 2012.

[3] Feng Chen and Jie Shen. A GPU parallelized spectral method for elliptic equations in rectangular domains. *Journal of Computational Physics*, 250:555–564, 2013.

[4] Sheng Chen and Jie Shen. An efficient and accurate numerical method for the spectral fractional Laplacian equation. *Journal of Scientific Computing*, 82(1):17, 2020.

[5] Ziang Chen, Jianfeng Lu, Yulong Lu, and Xiangxiong Zhang. On the convergence of Sobolev gradient flow for the Gross–Pitaevskii eigenvalue problem. *SIAM Journal on Numerical Analysis*, 62(2):667–691, 2024.

[6] Qiang Du, Lili Ju, Xiao Li, and Zhonghua Qiao. Maximum principle preserving exponential time differencing schemes for the nonlocal Allen–Cahn equation. *SIAM Journal on Numerical Analysis*, 57(2):875–898, 2019.

[7] Jean-Luc Guermond, Peter Minev, and Jie Shen. An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 195(44-47):6011–6045, 2006.

[8] Dale B Haidvogel and Thomas Zang. The accurate solution of Poisson's equation by expansion in Chebyshev polynomials. *Journal of Computational Physics*, 30(2):167–180, 1979.

[9] Jan S Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer Science & Business Media, 2007.

[10] Jingwei Hu and Xiangxiong Zhang. Positivity-preserving and energy-dissipative finite difference schemes for the Fokker–Planck and Keller–Segel equations. *IMA Journal of Numerical Analysis*, 43(3):1450–1484, 2023.

[11] A. Klckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

[12] Yuen-Yick Kwan and Jie Shen. An efficient direct parallel spectral-element solver for separable elliptic problems. *Journal of Computational Physics*, 225(2):1721–1735, 2007.

[13] Hao Li, Daniel Appelö, and Xiangxiong Zhang. Accuracy of Spectral Element Method for Wave, Parabolic, and Schrödinger Equations. *SIAM Journal on Numerical Analysis*, 60(1):339–363, 2022.

[14] Hao Li and Xiangxiong Zhang. Superconvergence of high order finite difference schemes based on variational formulation for elliptic equations. *Journal of Scientific Computing*, 82(2):36, 2020.

[15] Re Lynch, John R Rice, and Donald H Thomas. Tensor product analysis of partial difference equations. *Bull. Amer. Math. Soc.*, 70, 1964.

[16] Yvon Maday and Einar M Rønquist. Optimal error analysis of spectral methods with emphasis on non-constant coefficients and deformed geometries. *Computer Methods in Applied Mechanics and Engineering*, 80(1-3):91–115, 1990.

[17] Anthony T Patera. Fast direct poisson solvers for high-order finite element discretizations in rectangularly decomposable domains. *Journal of Computational Physics*, 65(2):474–480, 1986.

[18] Jie Shen. Efficient spectral-Galerkin method I. Direct solvers of second-and fourth-order equations using Legendre polynomials. *SIAM Journal on Scientific Computing*, 15(6):1489–1505, 1994.

[19] Jie Shen, Jie Xu, and Jiang Yang. A new class of efficient and robust energy stable schemes for gradient flows. *SIAM Review*, 61(3):474–506, 2019.
[20] Jie Shen and Xiangxiong Zhang. Discrete maximum principle of a high order finite difference scheme for a generalized Allen–Cahn equation. *Communications in Mathematical Sciences*, 20(5):1409–1436, 2022.