

A GPU PARALLELIZED SPECTRAL METHOD FOR ELLIPTIC EQUATIONS

FENG CHEN[†] & JIE SHEN[§]

ABSTRACT. We design and implement the first polynomial-based spectral method on graphic processing units (GPUs). The key to success lies in the seamless integration of the matrix diagonalization technique and new generation CUDA tools. The method is applicable to elliptic equations with general boundary conditions in both 2-D and 3-D cases. We show remarkable speedups of more than 10 times in the 2-D case and more than 30 times in the 3-D case. The GPU-accelerated spectral method is applied to the coupled system of Navier–Stokes equation and Allen–Cahn equation, demonstrating that the new algorithm is particularly suitable for simulations with finer meshes and longer run-times.

1. INTRODUCTION

General-purpose computing on graphics processing units (GPGPU) has drawn much attention recently from the scientific computing community. The new list of top 500 supercomputers shows that more than 10 percent of them are now powered by NVIDIA Tesla GPUs (cf. [4]). Thanks to the high core density and the wide vector width SIMD architecture, using GPGPU can yield performance that is very hard for a conventional CPU to achieve, especially for high and regular throughput workloads (cf. [3]). Therefore, researchers in computational sciences are more and more interested in exploring efficient parallel strategies on GPUs.

There exist many successful GPU implementations of numerical methods for partial differential equations, to name a few, fast multipole methods [10], nodal discontinuous Galerkin methods [12], finite difference methods [6], finite element methods [24, 14, 13], Fourier spectral methods [9, 5], etc. However, to the best of the authors’ knowledge, not much effort, if not at all, are devoted to implementing, on GPUs, non-Fourier based spectral methods for problems with non-periodic boundary conditions. The aim of this paper is to exploit how to efficiently implement spectral methods with GPGPU. In particular, we shall design and implement an efficient matrix diagonalization method with spectral discretizations on the GPU, for solving elliptic and parabolic type equations with non-periodic boundary conditions on a d -dimensional rectangular domain ($d = 2, 3$).

It is well-known that, for separable elliptic equations, one can use the so-called matrix diagonalization technique (cf. [16, 11, 18]) to solve the linear systems from a spectral discretization in $O(N^{d+1})$ operations, where N is the number of points in each dimension. For time-dependent problems, these solvers are called repeatedly. Fortunately, the core part of this matrix diagonalization process is matrix-matrix multiplications, which can be efficiently parallelized on the GPU. More precisely, matrix-matrix multiplications can be efficiently addressed by the new generation CUDA tool, CUBLAS.

Key words and phrases. diagonalization, elliptic equation, spectral collocation method, CUDA, GPU.

The research is partly supported by NSF grant DMS-1217066 and AFOSR grant FA9550-11-1-0328.

[†]Division of Applied Mathematics, Brown University, Providence, RI 02912 (feng_chen.1@brown.edu).

[§]Department of Mathematics, Purdue University, West Lafayette, IN 47907 (shen@math.purdue.edu).

We found in the numerical experiments that the speedup of using GPU vs. CPU goes up as N increases. For example, the speedup reaches a factor of 15 with $N = 2^{12}$ in the 2-D case, and a factor of 37 with $N = 2^9$ in the 3-D case, where N is the number of points in each direction. In addition, computing times of different parts scale consistently with the complexity. In our algorithm, the throughput is regular and the global memory access is completely coalesced, indicating a good performance on the GPU.

To initialize the matrix diagonalization procedure, we need to first compute the eigenvalue and eigenvectors of 1-D problems on the host, then ship these eigenvalues and eigenvectors to the device because they will be used repeatedly in a typical time marching scheme for evolutionary problems.

This paper is organized as follows. Details of the matrix diagonalization method are described in Section 2 using the spectral collocation method as an example. They provide a background for an in-depth discussion in Section 3 for the GPU parallelization of the same algorithm. In Section 4, we present some benchmark results for solving model elliptic equation in both 2-D and 3-D, and for solving 2-D multiphase flows based on the coupled system of Navier–Stokes equation and Allen–Cahn equation.

2. MATRIX DIAGONALIZATION METHOD FOR SPECTRAL DISCRETIZATIONS

In this section, we describe the matrix diagonalization method for spectral discretizations of separable equations. To fix the idea, we consider the following model equation on a rectangular domain $\Omega = (-1, 1)^d$ ($d = 2, 3$):

$$(2.1) \quad \begin{cases} \alpha u - \Delta u = f, & \text{in } \Omega, \\ u|_{\partial\Omega} = 0. \end{cases}$$

2.1. Matrix diagonalization method in 2-D. Let P_N be the space of polynomials of degree less or equal than N , and $X_N = \{u \in P_N : u(\pm 1) = 0\}$. Let $\{x_j = y_j = z_j\}_{j=0}^N$ be the set of Legendre Gauss-Lobatto points with $x_0 = -1$ and $x_N = 1$. The Legendre spectral-collocation method for (2.1) is:

Find $u_N \in X_N \times X_N$ such that

$$(2.2) \quad \alpha u_N(x_i, y_j) - \Delta u_N(x_i, y_j) = f(x_i, y_j), \quad 1 \leq i, j \leq N-1.$$

Let $h_j(x) \in P_N$ be the Lagrange polynomial based on $\{x_k\}_{k=0}^N$, i.e., $h_j(x_i) = \delta_{ij}$. Writing $u_N(x, y) = \sum_{i,j=1}^{N-1} u_N(x_i, y_j) h_i(x) h_j(y)$ in (2.2) and denoting

$$(2.3) \quad \begin{aligned} s_{ij} &= -h_j''(x_i), \quad S = (s_{ij})_{1 \leq i, j \leq N-1}; \\ U &= (u_{ij} = u_N(x_i, y_j))_{1 \leq i, j \leq N-1}, \quad F = (f_{ij} = f(x_i, y_j))_{1 \leq i, j \leq N-1}, \end{aligned}$$

we find that (2.1) is equivalent to

$$(2.4) \quad \alpha U + SU + US^T = F.$$

In order to apply the matrix diagonalization method, we first solve the eigenvalue problem:

$$(2.5) \quad SE = E\Lambda,$$

where $\Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_{N-1})$ are the eigenvalues of S , and the columns of E are the corresponding eigenvectors. It is well-known that $\{\lambda_j\}$ are all real and positive (cf. [7]). Then, setting

$$(2.6) \quad U = EVE^T$$

in (2.4), multiplying the result by E^{-1} from the left, and by E^{-T} from the right, we obtain

$$(2.7) \quad \alpha V + \Lambda V + V\Lambda = G := E^{-1}FE^{-T},$$

which can be written componentwise as

$$(2.8) \quad (\alpha + \lambda_i + \lambda_j)v_{ij} = g_{ij},$$

where g_{ij} is the (i, j) -th element of G . Therefore, the linear system is totally decoupled for each component (i, j) . We summarize below the algorithm for solving (2.1) in 2-D on CPUs.

Algorithm 1

- (1) Pre-compute and store $\{S, E, E^{-1}, \Lambda\}$;
- (2) Compute G in (2.7);
- (3) Solve for V from the decoupled linear system (2.7);
- (4) Calculate U according to (2.6).

2.2. Matrix diagonalization method in 3-D. The above matrix diagonalization can be extended to three-dimensional case in a straightforward fashion. We briefly describe the corresponding steps below.

In the 3-D case, the spectral-collocation method in 3-D is: find $u_N \in X_N \times X_N \times X_N$ such that

$$(2.9) \quad \alpha u_N(x_i, y_j, z_k) - \Delta u_N(x_i, y_j, z_k) = f(x_i, y_j, z_k), \quad 1 \leq i, j, k \leq N - 1.$$

The matrix diagonalization method described in the last section will involve multiplications of eigenmatrix E (and E^{-1}) to the three dimensional unknown array $(u_{lmn} = u(x_l, y_m, z_n))_{1 \leq l, m, n \leq N-1}$, and the inverse E^{-1} to the three dimensional forcing array $(f_{lmn} = u(x_l, y_m, z_n))_{1 \leq l, m, n \leq N-1}$. More precisely, the 3-D correspondence of (2.4) becomes:

$$(2.10) \quad \alpha u_{lmn} + s_{li}u_{imn} + s_{mj}u_{ljn} + s_{nk}u_{lmk} = f_{lmn}, \quad 1 \leq l, m, n \leq N - 1.$$

In the above (and below), the repeated index means that a summation of that index from 1 to $N - 1$.

The transform corresponding to (2.6) becomes

$$(2.11) \quad u_{lnm} = e_{li}e_{mj}e_{nk}v_{ijk}, \quad 1 \leq l, m, n \leq N - 1.$$

Let us denote

$$(2.12) \quad g_{lnm} = e_{li}^{-1}e_{mj}^{-1}e_{nk}^{-1}f_{ijk}, \quad 1 \leq l, m, n \leq N - 1.$$

where e_{ij}^{-1} is the (i, j) -th component of E^{-1} . Then, the equation corresponding to (2.8) is:

$$(2.13) \quad (\alpha + \lambda_i + \lambda_j + \lambda_k)v_{ijk} = g_{ijk}, \quad 1 \leq i, j, k \leq N - 1.$$

In summary, the matrix diagonalization the algorithm for solving (2.1) in 3-D on CPUs is as follows.

Algorithm 2

- (1) Pre-compute and store $\{S, E, E^{-1}, \Lambda\}$;
- (2) Compute $\{g_{lmn}\}$ in (2.12);
- (3) Compute for $\{v_{ijk}\}$ from the decoupled linear system (2.13);
- (4) Compute $\{u_{lmn}\}$ according to (2.11).

Remark 2.1. The eigenvalue decompositions in (2.5) can be pre-computed and stored once for all. Therefore, for both 2-D and 3-D cases, the computational complexity is dominated by the matrix-matrix multiplications in steps (2) and (4), which cost $O(N^{d+1})$ flops.

Remark 2.2. While we have presented the algorithm with the spectral-collocation method, the same algorithm can be applied to spectral-Galerkin methods (cf. [21, 22, 19]) which lead to sparse, for problems with constant-coefficients, and well-conditioned systems.

3. DESIGN AND IMPLEMENTATION OF SPECTRAL METHODS ON THE GPU

We discuss now in detail the design and implementation of the algorithm in the last section on the GPU. There are three basic strategies for optimizing the performance on GPUs (cf. [1, 2]): (i) maximizing memory throughput; (ii) maximizing utilization; and (iii) maximizing instruction throughput. In Section 3.1, we address strategy (i) and describe the overall strategy of the GPU parallelization for our algorithm. In Section 3.2 and 3.3, we focus on strategy (ii), by using both CUBLAS and a user lever kernel function. Since our algorithm only involves additions and multiplications, we do not need to address strategy (iii).

All our experiments were done on an Intel Xeon-E5540 CPU (eight-core threaded with the MKL BLAS library) and on an Nvidia Tesla K20 GPU. Computing times were given in seconds throughout the paper.

To explain essential ideas, we consider the 2-D case first in the first three subsections. Then we consider the 3-D case in Section 3.4.

3.1. Memory Throughput and GPU Parallelization. By reviewing **Algorithm 1**, we observe that data in step (1) need to be pre-computed. We initialize them on the host and then transfer the results to the GPU. An important motivation for accelerating the matrix-diagonalization algorithm for spectral approximations is for solving time-dependent partial differential equations. Therefore, data that do not change in time should be pre-computed and stored on the GPU once and for all, in order to minimize data transfer between the host and the device.

We list all auxiliary arrays (scalars omitted) in Table 1. According to Table 1, we need at least $7q \times q$

variable	size	meaning
x	$(N + 1) \times 1$	Gauss-Lobatto points
$S^{(2)}$	$q \times q$	collocation matrices
E, E^{-1}	$2q \times q$	eigen-vector matrices
F, G, U, V	$4q \times q$	r.h.s. and solution
Λ	$q \times 1$	eigenvalues of $S^{(2)}$

TABLE 1. Auxiliary data that are static in time ($q = N - 1$).

($q = N - 1$) matrices. In addition, it is preferred to allocate another two working matrices to avoid in-place function calls for the ease of implementation. Therefore, the total number of double precision matrices is 9. Current GPUs are still limited by storage. For example, the Nvidia Tesla K20 GPU has a storage of 6 gigabytes. If we take, $N = 2^K$, a simple calculation reveals that we are limited to $K \leq 13$.

Data in Table 1 will be generated on the CPU, and shipped to the GPU with the CUDA runtime API *cudaMemcpy*. We denote the wall time for shipping the memory from the host to the device as HtD.

In Table 2, we list clock times (in seconds) for the initialization and the memory copy. There is only one matrix (the unknown numerical solution) that needs to be copied back from the device to the host (DtH), in order to complete the algorithm. Hence, DtH is much smaller than HtD.

Note that the initialization cost is dominated by the eigenpair computation of the matrix S . This cost can be drastically reduced if one uses the Legendre-Galerkin method where only a sparse matrix (tridiagonal or penta-diagonal) is involved (cf. [21]).

The focus of our method is in parallelizing steps (2)-(4) in **Algorithm 1**. Each of them involves updating information related to the unknown matrix U . We are able to perform these steps on GPU only since every matrix in need has been shipped to the device. This is crucial because lowering the memory transfer between

K	initialization	HtD	DtH
3	3.8e-04	3.3e-04	1.0e-04
4	6.0e-04	3.4e-04	9.9e-05
5	1.2e-03	3.9e-04	1.1e-04
6	5.7e-03	6.1e-04	1.4e-04
7	3.6e-02	1.4e-03	2.8e-04
8	1.9e-01	4.3e-03	7.5e-04
9	1.0e+00	1.6e-02	3.0e-03
10	6.3e+00	5.5e-02	9.6e-03
11	5.1e+01	2.1e-01	3.1e-02
12	4.0e+02	8.1e-01	1.3e-01

TABLE 2. The eigenvalue decomposition (initialization) is done on the host. HtD (DtH) is the wall time of memory copies from the host (device) to the device (host). These operations can be done once for all in most applications.

host and device is the high priority for optimizing CUDA applications. This point has been laid out in the beginning of this section.

3.2. Matrix Multiplications with CUBLAS. The steps (2)-(4) in **Algorithm 1**, especially (2) and (4), dominate the computational complexity and computing times. Thus, it is important to implement them with efficient parallelizing strategies. The key observation is that steps (2) and (4) can be performed with a BLAS operation [25]. In Table 3, we list the three core operations. Noted that step (3) is not a linear operation, and we will deal with it separately in next subsection.

operation	program	step	meaning
p2s	dgemm	(2)	$G \leftarrow E^{-1}FE^{-T}$
kernel	/	(3)	solving for V in (2.7)
s2p	dgemm	(4)	$U \leftarrow EVE^{-1}$

TABLE 3. Core operations in the spectral collocation method: p2s: performing the physical-to-spectral transformation according to (2.7); s2p: performing the spectral-to-physical transformation according to (2.6).

The CUBLAS library comes handy for these numerical linear algebra tasks on GPUs and they are heavily tuned in particular for Nvidia devices. Therefore, it is natural to perform these BLAS operations on GPUs instead of CPUs. In the implementation, users of CUBLAS need to create one or several handles for operations in Table 3. That allows CUDA to collect all BLAS operations involved in the program, and the optimal grids will be arranged automatically by CUBLAS. We emphasize that all the matrix operations in our algorithm are dense. Hence, the global memory access is completely coalesced. In principle, these CUBLAS calls can achieve the highest efficiency thanks to the high core density and the massively thread-parallelization.

3.3. The Grid Arrangement: Maximizing Utilization on the Multiprocessor Level. While the computational complexity (not necessarily computing times) of the step (3) is negligible compared to that of steps

(2) and (4), it still needs to be parallelized once the dynamic data is on the device. Otherwise, the overall speedup can deteriorate by intermediate serial operations or possible memory transfers. In this subsection, we focus on the parallel strategy for step (3), which is the only kernel function that needs to be written by users.

We can immediately see that solving (2.8) is a perfect scenario for the GPU parallelization, because each component of V is decoupled and each index (i, j) can be mapped to an independent thread of the device. The question remains on how to design appropriate grid and block sizes for this particular kernel function. In general, the grid size should be large. Therefore, what we need to determine is the block size. To this end, we carried on an experiment of the CUDA kernel operation using random matrices of size $2^K \times 2^K$. The result is listed in Table 4.

K	Grid A	Grid B	Grid C
3	4.3e-05	4.2e-05	1.9e-05
4	2.0e-05	1.4e-05	2.1e-05
5	2.3e-05	1.3e-05	1.6e-05
6	3.0e-05	1.5e-05	1.6e-05
7	6.9e-05	1.6e-05	1.7e-05
8	2.7e-04	2.6e-05	2.7e-05
9	1.2e-03	4.9e-05	5.1e-05
10	4.9e-03	1.7e-04	1.8e-04
11	1.9e-02	6.7e-04	6.7e-04
12	7.8e-02	2.8e-03	2.8e-03

TABLE 4. Computing times in seconds for different grid configurations in kernel. The block size (number of threads per block) in each of the above three grids is: 1×1 , 16×16 , 32×32 . The grid size is then determined as the quotient of the matrix size and block size.

It is observed that the optimal grid size should match the warp size or half-warp size of the device. On many GPUs, the number would be either 16 or 32. Though Grid A (see Table 4) leads to the largest grid size, it does not achieve the highest efficiency since the latency between different threads becomes significant.

We summarize below the algorithm outline for the spectral collocation method on the GPU.

Algorithm 3

- (1) Pre-computation on the host:
 - (a) Pre-compute and store $\{S, E, E^{-1}, \Lambda\}$ in (2.5);
- (2) Ship the above static variables from host to device;
- (3) Computation on the device:
 - (a) p2s: Compute G in (2.7);
 - (b) kernel: Solve the decoupled linear system (2.7) for V ;
 - (c) s2p: Calculate U according to (2.6);
- (4) Ship the result U from device to host.

3.4. 3-D Case. Essential ideas in the previous three subsections remain the same for the 3-D case. Auxiliary data are initialized on the host and shipped to the device once for all. In terms of the grid arrangement, the optimal grid size still matches the warp size, though a 3-D grid is launched.

The only thing that deserves more explanation is how to perform the matrix-matrix multiplications in (2.11) and (2.12). Vector data in the present algorithm are re-aligned as one dimensional arrays by going through x dimension first, y second, and z third. This is a natural convention that does not cause any trouble in the 2-D case. It is because that (2.7) and (2.6) can be calculated directly by calling CUBLAS_Dgemm and tuning the two operational parameters, CUBLAS_OP_N and CUBLAS_OP_T.

However, this convention brings a problem to the 3-D case, that is, the memory access is not coalesced in the y dimension. Therefore, before performing the matrix-matrix multiplication along the y dimension, the array needs to be re-aligned in order to be accessed continuously. In other words, we perform a matrix transpose before applying e_{mj} in (2.11), to ensure the y -index goes first, and perform another transpose after to reset the order, similarly for (2.12).

For the matrix transpose, two approaches are in consideration: (i) transpose with naive copy, and (ii) transpose with the memory coalesce. Consider the transpose of a matrix A of size $n_1 \times n_2$. If the column major ordering is used, then there is a stride of length of n_1 in the memory access of A^T . Therefore, the first approach is relatively slow for transposing large matrices. We use the second approach that yields much better efficiency when K is large. By dividing the matrix into tiles of size $2^4 \times 2^4$, this technique leads to a remarkable performance on the GPU. The essential idea of the memory coalesce is utilizing the on-cache feature of the shared memory on each block. Readers refer [17] for more details.

We summarize below the 3-D algorithm on the GPU.

Algorithm 4

- (1) Pre-computation on the host:
 - (a) Pre-compute and store $\{S, E, E^{-1}, \Lambda\}$ in (2.5);
- (2) Ship the above static variables from host to device;
- (3) Computation on the device:
 - (a) p2s: Compute G in (2.12);
 - (i) matrix multiplication in x ;
 - (ii) matrix transpose; matrix multiplication in y ; another transpose;
 - (iii) matrix multiplication in z ;
 - (b) kernel: Solve the decoupled linear system (2.13) for V ;
 - (c) s2p: Calculate U according to (2.11), in a similar way as p2s;
- (4) Ship the result U from device to host.

4. NUMERICAL EXPERIMENTS

In the first subsection, we demonstrate the performance of our spectral collocation method on the GPU for some model elliptic equations in 2-D and 3-D. In the second subsection, we apply the algorithm to solve the Navier–Stokes equation and the Allen–Cahn equation through an energy stable scheme which leads to a sequence of Poisson type equations at each time step. The detailed information of the CPU and the GPU in use was described at the beginning of Section 3.

4.1. The Model Equation. As a first test example, we consider the following second-order separable equations with general boundary conditions:

$$\begin{aligned}
 & (\alpha(x) + \beta(y))u - (a(x)u_x)_x - (b(y)u_y)_y = f \quad \text{in } \Omega = (-1, 1)^2; \\
 (4.1) \quad & a_{\pm}u(x, \pm 1) + b_{\pm}u_y(x, \pm 1) = g_{\pm}(x), \quad \forall x \in (-1, 1); \\
 & c_{\pm}u(\pm 1, y) + d_{\pm}u_x(\pm 1, y) = h_{\pm}(y), \quad \forall y \in (-1, 1).
 \end{aligned}$$

We observe first that the algorithm presented in Section 2 can be easily extended to handle the above equation (cf. [19]).

To test our algorithm, we choose the exact solution in (4.1) to be

$$(4.2) \quad u(x, y) = \exp(x) \exp(y),$$

and set the coefficients to be

$$(4.3) \quad \begin{aligned} \alpha(x) &= 2x, & \beta(y) &= 3y, & a(x) &= 3 \sin(x), & b(y) &= 4 \cos(y), \\ a_{\pm} &= 1, & b_{\pm} &= 3, & c_{\pm} &= \pm 1, & d_{\pm} &= \pm 2. \end{aligned}$$

The right hand side functions, $f(x, y)$, $g_{\pm}(x)$, and $h_{\pm}(y)$, are calculated accordingly.

After confirming that our algorithm does lead to spectral accuracy as expected, we list computing times for our algorithm on CPU and GPU respectively in Table 5. We set $N = 2^K$, where N is the highest polynomial degree in each direction. As K increases, the CUDA threads achieve higher parallel efficiency and speedups. In general, the speedup is more than 100 when $K > 9$.

K	CPU	GPU	speedup
3	6.7e-05	1.4e-04	0
4	1.5e-05	1.2e-04	0
5	5.7e-05	1.4e-04	0
6	8.3e-04	1.6e-04	5
7	9.2e-04	2.3e-04	4
8	3.2e-03	3.8e-04	8
9	2.1e-02	1.9e-03	11
10	1.3e-01	8.8e-03	15
11	9.5e-01	6.7e-02	14
12	7.6e+00	5.2e-01	15

TABLE 5. Computing times (in seconds) and speedups for (4.1) and (4.2).

In Table 6, we compare computing times of different parts corresponding to Table 3. For the above problem with general boundary conditions, we also need to form a new right hand side matrix (rhs), which only involves a level-one CUBLAS call. Note that the computing time for rhs and kernel is negligible compared to the other parts.

As a second example, we consider

$$(4.4) \quad 2u - 3u_{xx} - 4u_{yy} - 5u_{zz} = f \quad \text{in } \Omega = (-1, 1)^3,$$

with the homogeneous Dirichlet boundary condition in each dimension. We take the exact solution to be $u = \sin(\pi x) \sin(\pi y) \sin(\pi z)$ with the right hand side $f(x, y, z)$ computed accordingly. The computing times of both CPU and GPU for solving the above equation are listed in Table 7. We observe that the speedups go up as K increases, and are better than those in the 2-D case.

K	rhs	p2s	kernel	s2p
3	2.1e-05	4.1e-05	1.3e-05	4.0e-05
4	1.7e-05	4.6e-05	1.3e-05	4.6e-05
5	1.7e-05	5.5e-05	1.4e-05	5.5e-05
6	1.7e-05	7.1e-05	1.3e-05	7.1e-05
7	1.9e-05	1.0e-04	1.5e-05	1.0e-04
8	2.9e-05	1.7e-04	1.8e-05	1.7e-04
9	1.0e-04	8.8e-04	4.7e-05	8.8e-04
10	3.5e-04	4.2e-03	1.5e-04	4.2e-03
11	1.4e-03	3.3e-02	5.4e-04	3.3e-02
12	5.3e-03	2.6e-01	2.2e-03	2.6e-01

TABLE 6. Computing times of different parts in Table 3 for (4.1) and (4.2).

K	CPU	GPU	speedup
3	7.3e-04	1.4e-04	5
4	9.5e-04	1.6e-04	6
5	1.2e-03	2.2e-04	5
6	1.3e-02	7.7e-04	17
7	1.7e-01	5.4e-03	31
8	2.4e+00	6.6e-02	36
9	3.4e+01	9.1e-01	37

TABLE 7. Computing times (in seconds) and speedups for the 3-D case.

4.2. Incompressible Navier–Stokes Equations. As an example of applications, we apply our GPU accelerated spectral solver to the 2-D incompressible Navier–Stokes equations:

$$(4.5) \quad \begin{cases} \mathbf{u}_t + \mathbf{u} \cdot \nabla \mathbf{u} = \nu \Delta \mathbf{u} - \nabla p + \mathbf{f}, \\ \nabla \cdot \mathbf{u} = 0, \\ \mathbf{u}|_{\partial\Omega} = \mathbf{g}, \quad \mathbf{u}|_{t=0} = \mathbf{u}_0, \end{cases}$$

where $\Omega = (-1, 1)^2$, \mathbf{u} is the unknown velocity field, p is the unknown pressure, \mathbf{f} is a given body force, and \mathbf{g} is the given Dirichlet data.

We discretize the above system in time with the following first-order rotational pressure correction scheme (cf. [8]).

Given $\{\mathbf{u}^n, p^n\}$, we solve first for an intermediate velocity $\tilde{\mathbf{u}}^{n+1}$ from

$$(4.6) \quad \begin{cases} \frac{\tilde{\mathbf{u}}^{n+1} - \mathbf{u}^n}{\delta t} + \mathbf{u}^n \cdot \nabla \mathbf{u}^n = \nu \Delta \tilde{\mathbf{u}}^{n+1} - \nabla p^n + \mathbf{f}^{n+1}, \\ \tilde{\mathbf{u}}^{n+1}|_{\partial\Omega} = \mathbf{g}. \end{cases}$$

Then, we solve $(\mathbf{u}^{n+1}, \psi^{n+1})$ from

$$(4.7) \quad \begin{cases} \frac{\mathbf{u}^{n+1} - \tilde{\mathbf{u}}^{n+1}}{\delta t} + \nabla \psi^{n+1} = \mathbf{0}, \\ \nabla \cdot \mathbf{u}^{n+1} = 0, \\ \mathbf{u}^{n+1} \cdot \mathbf{n}|_{\partial\Omega} = \mathbf{g} \cdot \mathbf{n}. \end{cases}$$

Finally, we update the velocity and pressure as follows:

$$(4.8) \quad \begin{aligned} \mathbf{u}^{n+1} &= \tilde{\mathbf{u}}^{n+1} - \delta t \nabla \psi^{n+1}, \\ p^{n+1} &= p^n + \psi^{n+1} - \nu \nabla \cdot \tilde{\mathbf{u}}^{n+1}. \end{aligned}$$

Note that, by applying the divergence operator, (4.6) can be written equivalently as a Poisson equation for ψ^{n+1} . As a consequence, at each time step, we need to solve three Poisson type equations (for u_1 , u_2 , and ψ), and perform several CUBLAS calls to update \mathbf{u}^{n+1} and p^{n+1} .

As a test example, we consider the driven cavity flow where the top plate of the domain moves towards left with the unit velocity. In Table 8, we compare the running times on the CPU and the GPU respectively. More specifically, we take $\delta t = 10^{-3}$, the final time $T = 10$, and $\nu = 10^{-4}$ that corresponds to a Reynolds number of 20,000. At this Reynolds number, the solution no longer converges to a steady state. We observe that speedups of using the GPU over the CPU for solving Navier-Stokes equations are consistent with those listed in Table 5 for solving an elliptic equation. In particular, with $K = 10$, which means a grid size of 1024×1024 , the CPU takes more than two hours to finish, while the GPU takes less than ten minutes.

K	7	8	9	10	11
CPU	3.05e+01	2.55e+02	1.11e+03	8.01e+03	5.98e+04
GPU	1.50e+01	2.58e+01	1.24e+02	5.71e+02	4.20e+03
speedup	2	10	9	14	14

TABLE 8. Running times for the driven cavity flow with $\delta t = 10^{-3}$, $T = 10$, $\nu = 10^{-4}$.

Next, we consider the following coupled system in the phase-field modeling of multiphase flows (cf. [15]), It consists of the Navier-Stokes equation with the elastic stress,

$$(4.9a) \quad \mathbf{u}_t + \mathbf{u} \cdot \nabla \mathbf{u} = \nu \Delta \mathbf{u} - \nabla p - \lambda \nabla \cdot (\nabla \phi \otimes \nabla \phi), \quad \text{in } \Omega,$$

$$(4.9b) \quad \nabla \cdot \mathbf{u} = 0, \quad \text{in } \Omega,$$

$$(4.9c) \quad \mathbf{u} = \mathbf{g}, \quad \text{on } \partial\Omega,$$

and an advected Allen-Cahn equation with the volume preserving:

$$(4.10a) \quad \phi_t + \mathbf{u} \cdot \nabla \phi = M \lambda \left(\Delta \phi - \frac{\phi^3 - \phi}{\epsilon^2} + \xi(t) \right), \quad \text{in } \Omega,$$

$$(4.10b) \quad \partial_n \phi = 0, \quad \text{on } \partial\Omega,$$

where \mathbf{u} is the flow velocity field, p is the flow pressure, ν is the dimensionless viscosity, ϕ is the phase variable equal to 1 in one material and -1 in another, λ controls the effect of elasticity, M controls the relaxation of the dynamics, ϵ is the interface width between two materials, $\xi(t)$ preserves the volume average of ϕ , and \mathbf{n} is the outward normal of $\partial\Omega$. In Figure 1, the top (bottom) plate moves toward right (left) with 4 units of velocity, which gives the boundary condition of \mathbf{u} , i.e., \mathbf{g} in (4.8c). To solve the above system

numerically, we use a first-order energy stable scheme (cf. [20, 23, 26]), which involves solving four elliptic equations implicitly and performing several numerical differentiations explicitly at each time step. In this example, the running times are nearly five hours on the CPU and less than thirty minutes on the GPU.

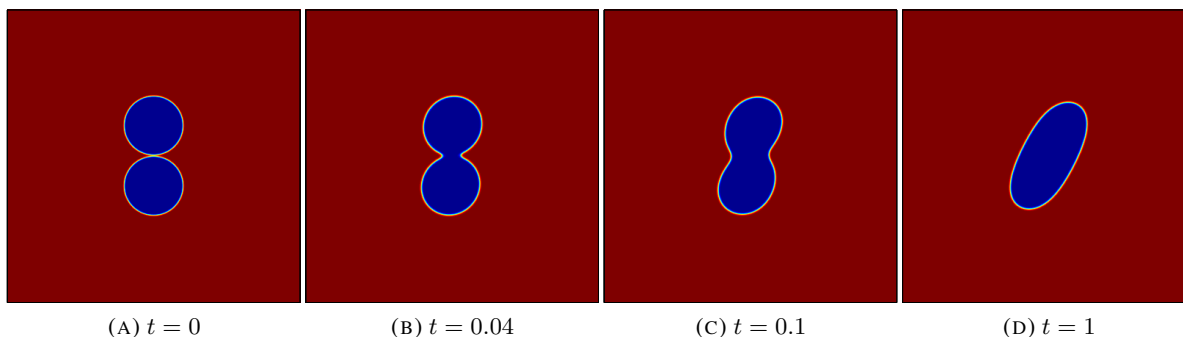


FIGURE 1. The evolution from two kissing drops to one deformed drop is simulated by using the coupled system (4.8)–(4.9) with parameters: $s = 2$, $\nu = 1$, $\epsilon = 5 \times 10^{-3}$, $\lambda = 0.02$, $M = 10^{-1}$. The time step size $\delta t = 10^{-5}$, which means that the simulation runs a total 100000 steps. The problem is discretized on a 512×512 Chebyshev grid. The final drop is tilted due to the presence of the velocity field. The computing time is 1730.95 seconds on the GPU.

In summary, we designed and implemented a GPU parallelized matrix diagonalization method for spectral approximations in two- and three-dimensional rectangular domains, and achieved remarkable speedups. We have applied our algorithm to solving the time dependent Navier–Stokes–Allen–Cahn system and observed consistently a speedup of over two-orders of magnitude. It is clear that the algorithm presented in this paper can be applied to a wide range of time dependent PDEs, and is expected to be capable of delivering real-time simulations for many practical problems.

Acknowledgment. Authors first would like to thank the referees for helpful suggestions and comments. Feng Chen would like to thank Professor Xiaofeng Yang at the University of South Carolina and Yuhang Tang at Brown University for their help. Authors acknowledge the generous facility support from National Institute for Computational Sciences (NICS) at Oak Ridge National Laboratory and the Brown University Center for Computation.

REFERENCES

- [1] CUDA C best practices guide. *NVIDIA Corporation*, 2011.
- [2] CUDA C programming guide. *NVIDIA Corporation*, 2011.
- [3] Cuda community showcase. http://www.nvidia.com/object/cuda_showcase.html.html, July 2012.
- [4] New top500 list. <http://top500.org/list/2012/06/100>, June 2012.
- [5] Heiko Bauke and Christoph H. Keitel. Accelerating the Fourier split operator method via graphics processing units. *Computer Physics Communications*, 182(12):2454–2463, December 2011.
- [6] D. M Dang, C. C. Christara, and K. R Jackson. Parallel implementation on GPUs of ADI finite difference methods for parabolic PDEs with applications to finance. *SSRN eLibrary*, pages 1–30, 2010.
- [7] D. Gottlieb and L. Lustman. The spectrum of the Chebyshev collocation operator for the heat equation. *SIAM J. Numer. Anal.*, 20:909–921, 1983.

- [8] J. L. Guermond, P. Mineev, and J. Shen. An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 195(44):6011C6045, 2006.
- [9] N. A. Gumerov, A. V. Karavaev, A. Surjalal Sharma, X. Shao, and K. D. Papadopoulos. Efficient spectral and pseudospectral algorithms for 3D simulations of whistler-mode waves in a plasma. *Journal of Computational Physics*, 230(7):2605–2619, 2011.
- [10] Nail A. Gumerov and Ramani Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227(18):8290–8313, 2008.
- [11] P. Haldenwang, G. Labrosse, S. Abboudi, and M. Deville. Chebyshev 3-d spectral and 2-d pseudospectral solvers for the Helmholtz equation. *J. Comput. Phys.*, 55:115–128, 1984.
- [12] 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, November 2009.
- [13] M. G. Knepley and A. R. Terrel. Finite element integration on GPUs. *Arxiv preprint arXiv:1103.0066*, 2011.
- [14] D. Komatitsch, G. Erlebacher, D. Goddeke, and D. Micha. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 229(20):7692–7714, 2010.
- [15] C. Liu and J. Shen. A phase field model for the mixture of two incompressible fluids and its approximation by a fourier-spectral method. *Physica D: Nonlinear Phenomena*, 179(3-4):211228, 2003.
- [16] R. E. Lynch, J. R. Rice, and D. H. Thomas. Direct solution of partial differential equations by tensor product methods. *Numer. Math.*, 6:185–199, 1964.
- [17] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in CUDA. *NVIDIA CUDA SDK Application Note*, 2009.
- [18] J. Shen and T. Tang. *Spectral and high-order methods with applications*. Science Press, 2006.
- [19] J. Shen, T. Tang, and L. L. Wang. *Spectral Methods: Algorithms, Analysis and Applications, volume 41 of*. Springer Series in Computational Mathematics, 2011.
- [20] J. Shen and X. Yang. Numerical approximations of allen-cahn and cahn-hilliard equations. *Discrete and Continuous Dynamical Systems, Series A*, 28:16691691, 2010.
- [21] Jie Shen. Efficient spectral-Galerkin method I. direct solvers for second- and fourth-order equations by using Legendre polynomials. *SIAM J. Sci. Comput.*, 15:1489–1505, 1994.
- [22] Jie Shen. Efficient spectral-Galerkin method II. direct solvers for second- and fourth-order equations by using Chebyshev polynomials. *SIAM J. Sci. Comput.*, 16:74–87, 1995.
- [23] Jie Shen and Xiaofeng Yang. A phase-field model and its numerical approximation for two-phase incompressible flows with different densities and viscosities. *SIAM Journal on Scientific Computing*, 32(3):1159, 2010.
- [24] Toru Takahashi and Tsuyoshi Hamada. GPU-accelerated boundary element method for Helmholtz’ equation in three dimensions. *International Journal for Numerical Methods in Engineering*, 80(10):1295–1321, July 2009.
- [25] Wikipedia contributors. Basic linear algebra subprograms, April 2012. Page Version ID: 478681520.
- [26] X. Yang, J. J. Feng, C. Liu, and J. Shen. Numerical simulations of jet pinching-off and drop formation using an energetic variational phase-field method. *Journal of Computational Physics*, 218(1):417428, 2006.