ATLANTIS PRESS

Advances in Engineering Research, volume 138

2nd International Conference on Machinery, Electronics and Control Simulation (MECS 2017)

GPU-accelerated Exponential Time Difference Method for Parabolic Equations with Stiffness

Xueyun Xie[a, *], Liyong Zhu[b]

Mathematics and Systems Science, Beihang University, Beijing 100191, China

[a]xxyliulei@buaa.edu.cn, [b]liyongzhu@buaa.edu.cn

Keywords: GPU acceleration technique, parabolic equations with stiffness, Runge-Kutta exponential time difference method, cuFFT.

Abstract: In this work, we present a Graphics Processing Unit (GPU) accelerated Runge-Kutta exponential time difference (RKETD) method for parabolic equations with stiffness based on CUDA (Computed Unified Device Architecture). In the proposed method, the cuFFT library developed by NVIDIA is employed to compute discrete Fast Fourier Transforms (FFTs) which is the key part in the RKETD method on GPU. Several CUDA code optimization skills are used to improve the speedup performance. The comparison of the numerical results of CUDA-implemented RKETD method on GPU and original RKETD method on CPU demonstrates that the former can obtain good speedup performance. Numerical experiments demonstrate effectiveness and accuracy of the GPU acceleration for typical nonlinear parabolic problem with stiffness.

1. Introduction

Recent years, the fast compact exponential time difference method [9, 10, 13, 5] have compiled much numerical techniques such as matrix diagonal decomposition, linear operator splitting, Fast Fourier Fransformation and so on. At the time of maintaining the high precision and stability, it effectively reduces the computational complexity of the algorithm and reduces the computational cost that the complicated parabolic equations with stiffness can be solved effciently. Although the development of these algorithms by means of discrete Fast Fourier Transform to reduce the computational complexity, they still can't meet the needs of solving large-scale parabolic problem with stiffness. Especially for three dimensional problem, due to the sharp increase in computation, the calculation time is far beyond the acceptable range. So it is required for further development of accelerated techniques to this method to achieve efficient solutions to large-scale parabolic problem with stiffness.

On the other hand, since the first GPU (Graphics Processing Unit) based on CUDA (Computed Unified Device Architecture) was released by NVIDIA in 2006, GPU acceleration technology has been rapidly developed and widely used [9, 10, 13, 5] in general computing. At the moment, the computing performance of high-end GPU has been reached to a trillion times per second. It is equivalent to a high performance computing cluster system, which performs far more than the performance of the mainstream CPU. The strong parallel acceleration capability of GPU provides a new way to further speed up.

In this paper, we will develop a fast exponential time difference method based on GPU for solving parabolic problems with stiffness. The compact exponential time difference format proposed in the document [9, 10, 13, 5] is an explicit iteration format. One of the characteristics of the algorithm is that each time layer needs to do the same operations for different data. Such single instruction multi-data algorithm is very good for parallel acceleration using GPUs. In addition, the core of the algorithm is the calculation of discrete Fast Fourier Transforms. And NVIDIA has developed a very sophisticated library-cuFFT [4] for computing discrete Fast Fourier Transforms which makes the acceleration algorithm easier to implement. This paper applies the GPU acceleration technology based on CUDA to the fast compact exponential time difference method. It realizes the acceleration of two-dimensional and three-dimensional nonlinear parabolic equations with stiffness. And the

computational efficiency was increased to 8 and 13 times. Finally, numerical examples verified the accuracy and validity of the accelerated algorithm.

The structure of this article is as follows: in the second part, we give the Runge-Kutta exponential time difference method based on the discrete Fast Fourier Transform in the literature; In the third part, the algorithm is presented based on the GPU parallel acceleration technique of CUDA and its CUDA programming optimization techniques; In the fourth part, the accuracy, validity and better acceleration performance of the proposed algorithm are verified by numerical examples of typical nonlinear parabolic problems; Finally, the paper summarizes and points out future research directions.

2. The Runge-Kutta exponential time difference method based on the Fast Fourier Transform

Consider the initial value of the parabolic equation:

$$\begin{cases} \dfrac{\partial u}{\partial t} = D\Delta u - f(u), & \mathbf{x} \in \Omega, t \in (t_0, t_0 + T], \\ u|_{t_0} = u_0, & \mathbf{x} \in \Omega. \end{cases} \tag{1}$$

where D is the diffusion coefficient and $f(u)$ is the non-linear term. $\Omega = \{x_b < x < x_e, y_b < y < y_e\}$ is a rectangular open area in R^d, d is the spatial dimensions and T>0. Here we consider the following periodic boundary conditions:

$$\begin{cases} u(t, \mathbf{x}_b) = u(t, \mathbf{x}_e), & \mathbf{x} \in [\mathbf{x}_b, \mathbf{x}_e] \in R^d, \\ \dfrac{\partial u}{\partial \mathbf{x}}(t, \mathbf{x}_b) = \dfrac{\partial u}{\partial \mathbf{x}}(t, \mathbf{x}_e), & \mathbf{x} \in [\mathbf{x}_b, \mathbf{x}_e] \in R^d. \end{cases} \tag{2}$$

Here we point out that the acceleration algorithm can also applie to the Dirichlet boundary conditions and Neumann boundary conditions. Now the paper presents the exponential time difference method based on the discrete fast fourier transform in the literature [10].

2.1 The exponential time difference method based on the discrete Fast Fourier Transform in two dimension

For the given area $\Omega = \{x_b < x < x_e, y_b < y < y_e\}$, set $(x_i, y_j) = (x_b + ih_x, y_b + jh_y)$. Where $0 \le i \le N_x$, $0 \le j \le N_y$, $h_x = (x_e - x_b)/N_x$ and $h_y = (y_e - y_b)/N_y$. Now define the following matrix:

$$\mathbf{G}_{P\times P} = \begin{pmatrix} -2 & 1 & 0 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 \\ & & \ddots & \ddots & \ddots & & \\ 0 & 0 & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 0 & 1 & -2 \end{pmatrix}_{P\times P}.$$

Set $\mathbf{A} = \dfrac{D}{h_x^2}\mathbf{G}_{N_x\times N_x}$ and $\mathbf{B} = \dfrac{D}{h_y^2}\mathbf{G}_{N_y\times N_y}$. Next introduce the following two special operators $\oplus$ and $\otimes$:

$$(\mathbf{A}\oplus\mathbf{U})_{i,j} = \sum_{l=1}^{N_x}(\mathbf{A})_{i,l}u_{l,j}, \qquad (\mathbf{B}\otimes\mathbf{U})_{i,j} = \sum_{l=1}^{N_y}(\mathbf{B})_{j,l}u_{i,l}.$$

It is easy to see that they have the following properties:

$$\mathbf{A}\oplus\mathbf{U} = \mathbf{A}\mathbf{U}, \quad \mathbf{B}\otimes\mathbf{U} = \mathbf{U}\mathbf{B}^T.$$

Assume that the numerical solution of the problem is $\mathbf{U} = (u_{i-1,j-1})_{N_x\times N_y}$ $(i = 1,\ldots,N_x, j = 1,\ldots,N_y)$. As for the Laplace operator in equation, we take the standard second-order center difference format and discrete it. Then we can get the following compact semi-discrete format[10]:

$$\frac{d\mathbf{U}}{dt} = \mathbf{A}\oplus\mathbf{U} + \mathbf{B}\otimes\mathbf{U} - \kappa\mathbf{U} - \mathcal{F}(\mathbf{U}). \tag{4}$$

Where $\mathcal{F}(\mathbf{U}) = (f(u_{i,j}))_{N_x \times N_y} - \kappa \mathbf{U}$ and $\kappa > 0$ is the linear operator splitting parameter which is introduced to increase the stability of the numerical format[6].

Considering that $\mathbf{A}$ and $\mathbf{B}$ are diagonalizable and have the following eigenvalue decompositions:

$$\mathbf{A} = \mathbf{P}_x \widetilde{\mathbf{D}}_x \mathbf{P}_x^{-1}, \quad \mathbf{B} = \mathbf{P}_y \widetilde{\mathbf{D}}_y \mathbf{P}_y^{-1}.$$

Where $\mathbf{P}_x$ and $\mathbf{P}_y$ are orthogonal matrices, and

$$\widetilde{\mathbf{D}}_x = diag[d_1^x, d_2^x, \ldots, d_{N_x}^x], \quad \widetilde{\mathbf{D}}_y = diag[d_1^y, d_2^y, \ldots, d_{N_y}^y]$$

With

$$d_k^x = -\frac{4D}{h_x^2} sin^2(\frac{(k-1)\pi}{N_x}), \quad k = 1, 2, \cdots, N_x, \quad (\mathbf{P}_x)_{k,j} = exp(-i\frac{2\pi(k-1)(j-1)}{N_x}), \quad k, j = 1, 2, \cdots, N_x,$$

$$d_k^y = -\frac{4D}{h_y^2} sin^2(\frac{(k-1)\pi}{N_y}), \quad k = 1, 2, \cdots, N_y, \quad (\mathbf{P}_y)_{k,j} = exp(-i\frac{2\pi(k-1)(j-1)}{N_y}), \quad k, j = 1, 2, \cdots, N_y.$$

Introduce the matrix $\mathbf{H} = (h_{i,j})_{N_x \times N_y}$ with the element $h_{i,j}$ defined by

$$h_{i,j} = d_i^x + d_j^y - \kappa. \tag{5}$$

As shown in [6], we define an operation "(e^*)" by taking exponentials of an array element by element as

$$(e^*)^{\mathbf{H}} = (e^{h_{i,j}})_{N_x \times N_y},$$

And another operator "$\odot$" for element by element multiplication between two arrays of same sizes
as

$$(\mathbf{M} \odot \mathbf{L})_{i,j} = (\mathbf{L} \odot \mathbf{M})_{i,j} = (m_{i,j} l_{i,j}),$$

Where $\mathbf{M} = (m_{i,j}), \mathbf{L} = (l_{i,j})$. Then, as in [6], a variation of constant formula by exponential time integration for (4) leads to

$$\mathbf{U}_{n+1} = \mathbf{P}_y \otimes \mathbf{P}_x \oplus \left((e^*)^{\mathbf{H}\Delta t} \odot \mathbf{P}_y^{-1} \otimes \mathbf{P}_x^{-1} \oplus \mathbf{U}_n - \int_0^{\Delta t} (e^*)^{\mathbf{H}(\Delta t - \tau)} \odot \mathbf{P}_y^{-1} \otimes \mathbf{P}_x^{-1} \oplus \mathcal{F}(\mathbf{U}(t_n + \tau)) d\tau \right). \tag{6}$$

Set

$$\mathbf{F}(t_n + \tau, \mathbf{U}(t_n + \tau)) = (f_{i,j})_{N_x \times N_y} = \mathbf{P}_y^{-1} \otimes \mathbf{P}_x^{-1} \oplus \mathcal{F}(\mathbf{U}(t_n + \tau))$$

And

$$\mathbf{Q}^{\mathbf{F}} = (q_{i,j}^{\mathbf{F}})_{(N_x-1)\times(N_y-1)} = \int_0^{\Delta t} \mathbf{F}(t_n + \tau, \mathbf{U}(t_n + \tau)) \odot (e^*)^{\mathbf{H}(\Delta t - \tau)} d\tau.$$

Then (6) can be written by

$$\mathbf{U}_{n+1} = \mathbf{P}_y \otimes \mathbf{P}_x \oplus \left((e^*)^{\mathbf{H}\Delta t} \odot \mathbf{P}_y^{-1} \otimes \mathbf{P}_x^{-1} \oplus \mathbf{U}_n - \mathbf{Q}^{\mathbf{F}} \right). \tag{7}$$

As for the $\mathbf{Q}^{\mathbf{F}}$ in (7), different methods of exponential time difference can be derived by using different processing methods [6, 10]. And the document [10] gives the first order to the fourth-order Runge-Kutta exponential time difference format.

Notice that for any matrix $\mathbf{V}$ of the size for $N_x \times N_y$, $\mathbf{P}_x \oplus \mathbf{V}$ is equivalent to doing a discrete Fourier Transform (DFT) of each column of the matrix $\mathbf{V}$. Similarly, $\mathbf{P}_y \otimes \mathbf{V}$ is equivalent to doing an inverse discrete Fourier Transform (iDFT) for each column of the matrix $\mathbf{V}$. So

$$\mathbf{P}_x \oplus \mathbf{V} = DFT_{N_x}(\mathbf{V}), \quad \mathbf{P}_x^{-1} \oplus \mathbf{V} = iDFT_{N_x}(\mathbf{V}).$$

Similarly,

$$\mathbf{P}_y \otimes \mathbf{V} = DFT_{N_y}(\mathbf{V}^T)^T, \quad \mathbf{P}_y^{-1} \otimes \mathbf{V} = iDFT_{N_y}(\mathbf{V}^T)^T.$$

In this way, by means of fast discrete Fourier Transform, the computational complexity of the compact Runge-Kutta time difference method in two dimension can be reduced from $O(N_x N_y N)$ to $O(N_x N_y \log_2 N)$, where $N = \max\{N_x, N_y\}$.

2.2 The exponential time difference method based on the discrete Fast Fourier Transform in three dimension

For three dimensions, the area of solution is $\Omega = \{x_b \le x \le x_e, y_b \le y \le y_e, z_b \le z \le z_e\}$. Set the points that $(x_i, y_j, z_l) = (x_b + ih_x, y_b + jh_y, z_b + lh_z)$, where $0 \le i \le N_x$, $0 \le j \le N_y$, $0 \le l \le N_z$ and let $hz = (z_e - z_b)/N_z$. Assume the solution of the format is that $\mathbf{U} = (u_{i-1,j-1,l-1})_{N_x \times N_y \times N_z}$, where $i = 1, \cdots, N_x$, $j = 1, \cdots, N_y$, $l = 1, \ldots, N_z$. Set a matrix in z direction: $\mathbf{C} = \frac{D}{h_z^2}\mathbf{G}_{N_z \times N_z}$. Similar to two dimension, (1) can be discrete as follows:

$$\frac{d\mathbf{U}}{dt} = \mathbf{A} \oplus \mathbf{U} + \mathbf{B} \otimes \mathbf{U} + \mathbf{C}\Theta\mathbf{U} - \kappa\mathbf{U} - \mathbf{F}(\mathbf{U}), \tag{8}$$

Where $\oplus$ and $\otimes$ are different from them in two dimension. In addition, we need to define another operator Θ. The definition of them are as follows:

$$(\mathbf{A} \oplus \mathbf{U})_{i,j,k} = \sum_{l=1}^{N_x} (\mathbf{A})_{i,l} u_{l,j,k}, \quad (\mathbf{B} \otimes \mathbf{U})_{i,j,k} = \sum_{l=1}^{N_y} (\mathbf{B})_{j,l} u_{i,l,k}, \quad (\mathbf{C}\Theta\mathbf{U})_{i,j,k} = \sum_{l=1}^{N_z} (\mathbf{C})_{k,l} u_{i,j,l}.$$

Similar to $\mathbf{A}$ and $\mathbf{B}$, set $\mathbf{C} = \mathbf{P}_z \widetilde{\mathbf{D}}_z \mathbf{P}_z^{-1}$, where $\widetilde{\mathbf{D}}_z = diag[d_1^z, d_2^z, \ldots, d_{N_z}^z]$. So we can get the format in three dimension:

$$\mathbf{U}_{n+1} = \mathbf{P}_z \Theta \mathbf{P}_y \otimes \mathbf{P}_x \oplus \left((e^*)^{\mathbf{H}\Delta t} \odot \left(\mathbf{P}_z^{-1} \Theta \mathbf{P}_y^{-1} \otimes \mathbf{P}_x^{-1} \oplus \mathbf{U}_n \right) - \mathbf{Q}^F \right). \tag{9}$$

Where $\mathbf{H} = (h_{i,j,k})_{N_x \times N_y \times N_z}$ and $h_{i,j,k} = d_i^x + d_j^y + d_k^z - \kappa$. We can also use the Runge-Kutta method to approximate $\mathbf{Q}^F$. In the same way, with the aid of three-dimensional Fast Fourier Transform, three-dimensional computational complexity can also be reduced from $O(N_x N_y N_z N)$ to $O(N_x N_y N_z \log_2 N)$, where $N = \max\{N_x, N_y, Nz\}$.

3. The CUDA model based on GPU acceleration

Although the development of the compact exponential time difference method in the literature reduces the computational complexity of the algorithm by means of the discrete Fast Fourier Transform, these algorithms still fail to meet the needs of large-scale parabolic problems with stiffness. Especially for three dimension, due to the sharp increase in computation, the calculation time is far beyond the acceptable range. This will require further development of accelerated techniques for such exponential time difference methods to achieve efficient solutions to large-scale parabolic problems.

In November 2006, NVIDIA released its first GPU based on CUDA. And the emergence of the CUDA architecture breaks the application bottleneck of GPU acceleration in general computing which promotes the fast development and extensive application of GPU acceleration technology. Compared with the serialized CPU computation, GPU parallel acceleration technology can improve the performance of single instruction and multi-data numerical algorithms. While the compact exponential time difference format proposed in the literature [9, 10, 13, 5] is an explicit iteration format. One of the characteristics of the algorithm is that each time layer needs to do the same work for different data. This kind of characteristics are good for parallel acceleration by using GPUs.

In the system of CPU and GPU, the CPU is called "host" and the GPU is called "device"[11]. CPUs are the main body of the entire computer system while GPUs are only part of the overall computer system. Both the CPU and the GPU have their own independent memory and resources. The functions that are invoked on the host side and executed on the host side are called host functions;

Those who are invoked on the host side and executed on the device side are kernel functions; While the functions that are invoked and executed both on the device are called the device functions. During the computation, the CPU sends instructions to the GPU and GPU calculate the instruction by using multithreading that can achieve the goal of parallel acceleration.

In the case of two dimension, the form of computation on the GPU is as follows:

```
__global__ void kernel(typename args)
{
    /*The ID of the position*/
    int idx = threadIdx.x + blockIdx.x _ blockDim.x;
    int idy = threadIdx.y + blockIdx.y _ blockDim.y;
    /*other program*/
}
```

The kernel function here must be declared with __global__ . ThreadIdx, blockIdx and blockDim are the index of the Thread, Block and the size of Block in the CUDA structure. This structure supports one-dimensional, two-dimensional and three-dimensional position indexes. When calculating the problem, the data need to be passed from the host to the device to be calculated and the results are passed back to the host after the device is computed. Here the main function is called as follows:

```
int main(int argc, char *argv[ ])
{
    /*Allocation*/
    /*Initialize*/
    /*Pass the data to GPU*/
    kernel<<<GridDim,BlockDim>>>(argc);
    /*Pass the data to CPU*/
    /*Free memory*/
}
```

Where the GridDim and BlockDim are the number of blocks and the number of threads. In a kernel, multiple blocks can be set in a grid and multiple threads can be set in a block. The logical structure of the block and thread is shown in the Fig.1.

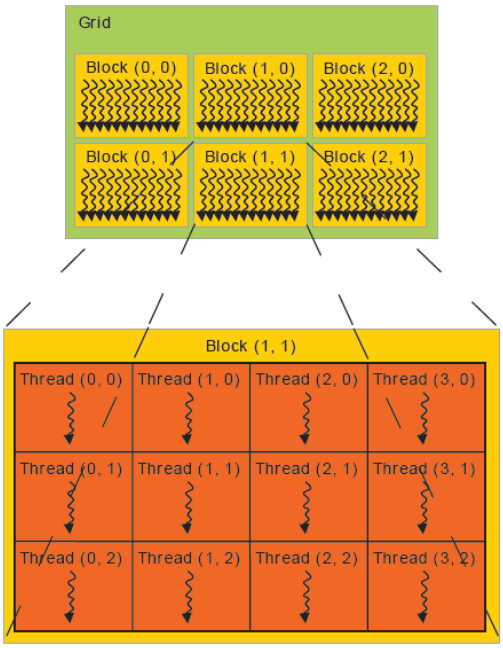

Fig. 1 The structure of Block and Thread

As for the fast compact exponential time difference method, another reason we use GPU to accelerate in parallel is that NVIDIA developed cuFFT, a function library based on CUDA's

computational discrete Fast Fourier Transform on GPU. The selection of the cuFFT library as an accelerated strategy for the algorithm has following advantages:

1) The library provides a high quality acceleration that is easy to implement;
2) GPU acceleration is achieved without the need for in-depth GPU programming knowledge;
3) It provides a standard API interface that reduces the program's changes while achieving GPU acceleration;
4) The library provides efficient applications for a wide range of applications;
5) The function of the library is safe and reliable because of testing by experts.

The way to call the library is as follows:

```
cufftHanle plan; //Declare a variable
cufftPlan2d(&plan,N,N,CUFFTZ2Z); //Set variable parameters(2 dimension)
cufftExecZ2Z(plan,idata,odata,CUFFT FORWARD); //Execute the Fourier Transform
```

We adopted the acceleration strategy above to accelerate the fast compact Runge-Kutta exponential time difference method. The numerical results show that the parallel acceleration strategy based on GPU can achieve better acceleration performance.

4. Numerical experiments

Next, we invalid the accelerated performance of the proposed accelerated strategy by numerical example. Now we simulate an average curvature flow problem to validate acceleration. Considering the initial value of the average curvature flow [3, 1]:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \Delta u - \dfrac{1}{\epsilon^2}(u^3 - u), & (x, y) \in \Omega, t \in [0, T], \\ u(0, \mathbf{x}) = tanh\left(\dfrac{R_0 - \|\mathbf{x}\|_2}{\sqrt{2}\epsilon}\right), & (x, y) \in \Omega. \end{cases} \tag{10}$$

Where $R_0 = 0.4$ and $\Omega = [-0.5, 0.5]^d$. And d>1 is the dimension of space. This problem is typical of nonlinear parabolic problem with stiffness. The smaller the parameter ϵ, the more rigid the problem. And the challenge for numerical computing is greater [3, 1]. The problem describes the contraction of the circle and the sphere in two dimensions and in three dimensions. Fig. 2 and Fig. 3 show the contraction of a two-dimensional shrinking circle with a space grid of 256^2 and a three-dimensional shrinking sphere with a space grid of 128^2.

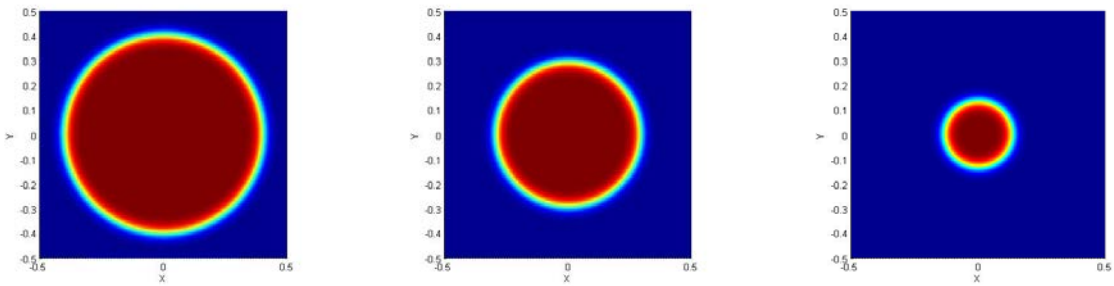

Fig. 2 Shrinking circle (left: t=0, middle: t=0.04, right: t=0.075)

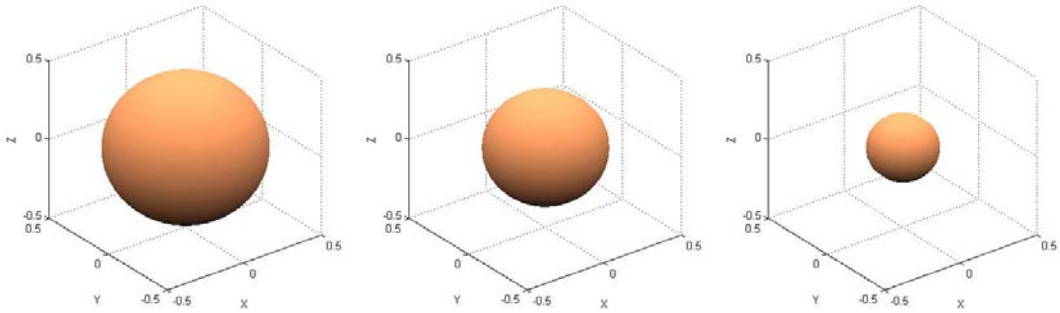

Fig. 3 Shrinking sphere (left: t=0, middle: t=0.02, right: t=0.0375)

This example compares the time in CPU environment and the time in GPU environment respectively. The graphics card used here is Tesla K20m and the driver version is cuda6.5. Here we use the second order format for the efficiency comparison. Set the final time in two dimension $T = 0.075$. We select the grid of space and time for $4096^2 \times 4096$ when calculate the radius of the R_ϵ instead of precise Radius. Here the $\epsilon = 0.02$. For the three-dimensional contraction sphere, the final time is $T = 0.0375$. We select the grid of space and time for $256^3 \times 2048$ when calculate the radius of the R_ϵ instead of precise Radius.

As we can see from the Table 1 and Table 2, with the encryption of the spatial grid, the errors gradually reach to the second order convergence accuracy. And by comparing CPU and GPU calculations in two dimension, the GPU can increase the efficiency of computing to about 13 times. For three dimension, using GPU computing can increase computing effciency to about 8 times.

Table 1 Comparison of computational efficiency for CPU and GPU in two dimension

N_t	$\lvert R - R_\epsilon \rvert$	*ord*	CPU time (s)	GPU time (s)	Speedup
128	1.9701e-01	---	1.15659e+03	8.82830e+01	13.1009
256	1.2814e-01	0.6205	2.31848e+03	1.77420e+02	13.0678
512	6.3743e-02	1.0074	4.62468e+03	3.52049e+02	13.1365
1024	2.3516e-02	1.4386	9.28456e+03	7.03604e+02	13.1957
2048	5.8855e-03	1.9984	1.93535e+04	1.40709e+03	13.7542

Table 2 Comparison of computational efficiency for CPU and GPU in three dimension

N_t	$\lvert R - R_\epsilon \rvert$	*ord*	CPU time (s)	GPU time (s)	Speedup
64	1.9367e-01	---	7.38360e+02	9.32561e+01	7.9175
128	1.2535e-01	0.6276	1.47616e+03	1.86406e+02	7.9191
256	6.1837e-02	1.0194	3.00808e+03	3.72572e+02	8.0738
512	2.2584e-02	1.4532	6.04700e+03	7.72522e+02	7.8276
1024	5.6121e-03	2.0087	1.18297e+04	1.48876e+03	7.9460

5. Conclusion

In this work, a compact exponential time difference method based on GPU acceleration is presented for parabolic equations with stiffness. The fast compact exponential time difference method has a low computational complexity in itself. We implemented the acceleration of fast algorithm on hardware by using GPU acceleration technology based on CUDA architecture. The numerical experiments showed that the acceleration efficiency can be increased by 8 to 13 times under certain grid conditions. While due to a single GPU and CPU memory limit, so the acceleration algorithm developed in this paper also has some limitations on the scale of the problem. Meanwhile, When the GPU is calculated, the CPU is always idle. So, the development of a collaborative acceleration algorithm for multiple CPUs and multi-GPU on heterogeneous platforms is a direction to further study in the future.

References

[1] P.M. De, M. Schatzman, Geometrical evolution of developed interfaces, T.Am.Math.Soc. 347 (1995) 1533-1589.
[2] Q. Du, W. Zhu, Analysis and Applications of the Exponential Time Differencing Schemes and Their Contour Integration Modifications, BIT. 45.2 (2005) 307-328.
[3] L.C. Evans, J. Spruck, Motion of Level Sets by Mean Curvature I. J. Geom. Anal. 5.2 (1992) 77-114.
[4] Information on http://docs.nvidia.com/cuda/cu_t/index.html#axzz4do60s9gZ.
[5] L. Ju, J. Zhang, Q. Du, Fast and accurate algorithms for simulating coarsening dynamics of Cahn-Hilliard equations. Comp. Mater. Sci. 108 (2015) 272-282.

[6] L. Ju, J. Zhang, L. Zhu, Fast Explicit Integration Factor Methods for Semilinear Parabolic Equations. J. Sci. Comput. 62.2 (2015) 431-455.
[7] Kessler, Daniel, Nochetto, A posteriori error control for the Allen-Cahn problem: circumventing Gronwall's inequality. Esaim. Math. Model. Num. 38 (2004) 129-142.
[8] Y. Li, H.G. Lee, D. Jeong, An unconditionally stable hybrid numerical method for solving the Allen-Cahn equation. Comput. Math. Appl. 60.6 (2010) 1591-1606.
[9] L. Zhu, L. Ju, W. Zhao, Fast High-Order Compact Exponential Time Differencing Runge-Kutta Methods for Second-Order Semilinear Parabolic Equations. J. Sci. Comput. 67.3 (2016) 1043-1065.
[10] L. Zhu, Efficient and Stable Exponential Runge-Kutta Methods for Parabolic Equations. 9.1 (2017) 157-172.
[11] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming. 11.4 (2010) 387-415.
[12] C. Xu, T. Tang, Stability Analysis of Large Time-Stepping Methods for Epitaxial Growth Models. Siam. J. Numer. Anal. 44.4 (2006) 1759-1779.
[13] X. Wang, L. Ju, Q. Du, Efficient and stable exponential time differencing Runge-Kutta methods for phase field elastic bending energy models. J. Comput. Phys. 316 (2016) 21-38.
[14] X. Yang, J.J. Feng, C. Liu, Numerical simulations of jet pinching-off and drop formation using an energetic variational phase-field method. J. Comput. Phys. 218.1 (2006) 417-428.
[15] J. Zhang, Q. Du, Numerical Studies of Discrete Approximations to the Allen-Cahn Equation in the Sharp Interface Limit. Siam. J. Sci. Comput. 31.4 (2009) 3042-3063.