Journal of Computational Information Systems 11: 24 (2015) 7805–7812
Available at http://www.Jofcis.com

A CUDA Solver for Helmholtz Equation

Mingming REN [1,2,*], Xiaoguang LIU [1,2], Gang WANG [1,2]

[1] *College of Computer and Control Engineering, Nankai University, Tianjin 300071, China*
[2] *College of Software, Nankai University, Tianjin 300071, China*

Abstract

In this paper, we present a CUDA based solver for the Helmholtz equation $\Delta u + \lambda u = f$ in three dimensional rectangular area. This solver follows the algorithm of FISHPACK – A Fortran package for solving elliptical equations. It uses DFT and its variants such as DST and DCT to efficiently find the numerical solution. It is able to solve five different kinds of boundary conditions for each coordinate direction. The experiments show that it can achieve a speedup of up to around 30 compared to the solver in FISHPACK. The source code can be downloaded from https://github.com/rmingming/cudahelmholtz/.

Keywords: CUDA; Helmholtz Equation; DFT; DST; DCT; FISHPACK

1 Introduction

The Helmholtz equation $\Delta u + \lambda u = f$ is a very special kind of partial differential equations (PDEs). It often arises in the study of physical problems. Laplace equation and Poisson equation are all its special forms. Finding a numerical solution to a single Helmholtz equation may not take you too much time, but solving it as fast as possible maybe essential in some numerical simulations.

Usually when solving partial differential equations, we first use finite difference method or finite element method to discretize the equation and form a very large and often sparse linear system, then by solving this linear system we can get the numerical solution to the original equation [1]. But since Helmholtz equation has a good form, it can be solved by the Fourier based methods. Using Fourier based methods to solve Helmholtz equation is usually much faster. FISHPACK [2] is a famous Fortran software package for solving elliptical equations including the Helmholtz equation and it is highly efficient.

The solver in FISHPACK for solving Helmholtz equation in three dimensional space is called *hw3crt*. Its algorithm is based on the Fourier methods, it uses Discrete Fourier Transform (DFT) and its variants Discrete Sine/Cosine Transform (DST/DCT) [3, 4] in dealing with different

[*]Corresponding author.
Email address: `renmingming@nbjl.nankai.edu.cn` (Mingming REN).

1553–9105 / Copyright © 2015 Binary Information Press
DOI: 10.12733/jcis15C0261
December 15, 2015

boundary conditions. The most important aspect of *hw3crt* is that it can solve several different kinds of boundary conditions for each coordinate.

In recent years, researchers and engineers are increasingly using Graphics Process Units (GPUs) to accelerate their codes. In order to address the computational challenges in many disciplines, the NVIDIA Corporation introduced Compute Unified Device Architecture (CUDA) [5] which is now supported on most graphical devices built by NVIDIA. CUDA provides a very minimal extension of C language. By using CUDA, the software developer can utilize the GPU horsepower easily. Currently it is the most popular way of using GPUs in general computing.

Using GPUs to solve PDEs could drastically reduce the execution time. Since CUDA has provided the Fast Fourier Transform (FFT) library CUFFT, it can be used to solve some special PDEs, such as Poisson equations [6, 7]. On the other hand, since CUFFT doesn't contain any function to compute DST or DCT, the PDEs with complex boundary conditions (not the periodic boundary condition) are not easy to solve by using Fourier based methods on GPUs.

In this paper, we present a CUDA based Helmholtz equation solver. It is a Fourier based solver and can deal with five different boundary conditions which are common in practice. We illustrate that our solver is accurate enough through experiments, and it achieves a speedup of up to around 30 compared to *hw3crt*. In particular, the code runs totally on GPU, it could be a good replacement for *hw3crt*, especially when user's code is already running on GPU.

2 Algorithm and Implementation

2.1 Problem

The three dimensional Helmholtz equation in Cartesian coordinate is:

$$\Delta u + \lambda u = f$$

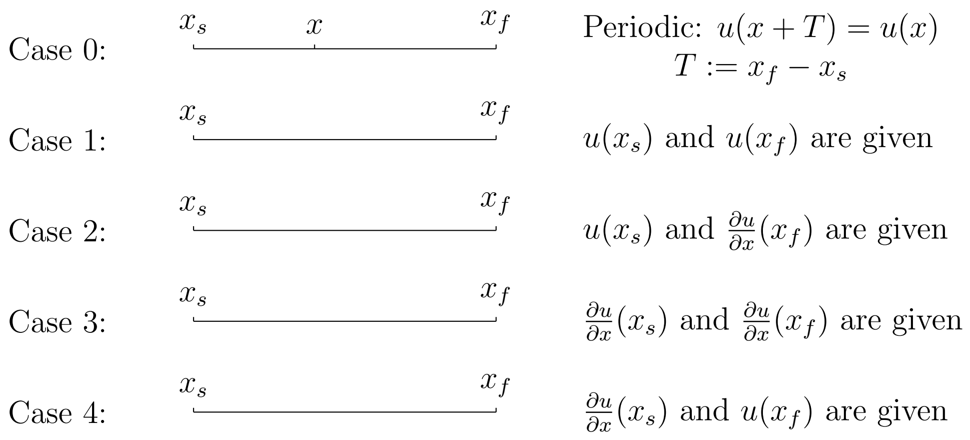
where u is an unknown function, λ is a constant value, f is a known function, Δ is $\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$. Function u and f are defined on a rectangular area $x_s \leq x \leq x_f,\ y_s \leq y \leq y_f,\ z_s \leq z \leq z_f$.

In order to numerically solve the Helmholtz equation, we divide x, y, z directions into $N_x,\ N_y,\ N_z$ panels respectively and try to obtain the u value at the grid points. Usually we enlarge $N_x,\ N_y,\ N_z$ to get more accurate solution. In this paper, let $N_x,\ N_y,\ N_z$ be the same N just for simplicity, but they can be different in our code. Also we let N to be some power of 2 in this paper (this is in order to compute the order of the error), but it can be any positive integer in our code.

2.2 Boundary condition

Boundary condition is essential in solving partial differential equation. There are three different kinds of boundary conditions often used in numerical simulations. They are periodic boundary condition, Dirichlet boundary condition and Neumman boundary condition.

As an illustration, we describe them in one dimensional case. Periodic boundary condition means that $u(x + T) = u(x)$, where $T := x_f - x_s$. We follow the convention and call it the periodic *boundary* condition, but essentially it is not a true *boundary* condition. Dirichlet boundary condition means that the u values at the boundary are given. Neumann boundary condition means that the $\frac{\partial u}{\partial x}$ values at the boundary are given. Since in each coordinate direction we have two boundaries, so there are five different boundary cases in each coordinate, as shown in Fig. 1.



Case 0:	x_s — x — x_f	Periodic: $u(x+T) = u(x)$ $T := x_f - x_s$
Case 1:	x_s — x_f	$u(x_s)$ and $u(x_f)$ are given
Case 2:	x_s — x_f	$u(x_s)$ and $\frac{\partial u}{\partial x}(x_f)$ are given
Case 3:	x_s — x_f	$\frac{\partial u}{\partial x}(x_s)$ and $\frac{\partial u}{\partial x}(x_f)$ are given
Case 4:	x_s — x_f	$\frac{\partial u}{\partial x}(x_s)$ and $u(x_f)$ are given

Fig. 1: Five different boundary cases

When using Fourier methods to solve PDEs, different boundary conditions indicate that different methods should be adopted. For the periodic boundary case, we can use DFT directly. But for the other four boundary cases, we need to use one of the variants of DFT, i.e., DST or DCT. Before using DST or DCT, the non-zero Dirichlet and Neumann boundary conditions need to be processed in our code. After that, the values of u or $\frac{\partial u}{\partial x}$ at the boundary would be all zero. In the following implementation of DST and DCT, we always assume the zero Dirichlet and Neumann boundary conditions are satisfied.

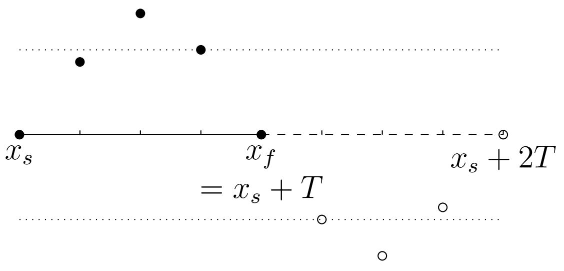
2.3 Implementation of DST and DCT

The CUDA library CUFFT only provides FFT functions, it is unable to use CUFFT to compute DST or DCT directly. But if we extend the original data array in specific ways, the DST/DCT of the original data array can be obtained from DFT (FFT) of the extended data array. As shown in Fig. 2, according to the boundary conditions, we use different extending methods and the purpose is to extend the data array to a whole period.

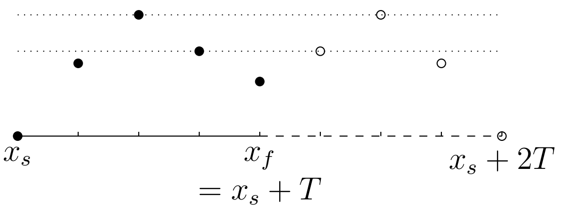
Take $N = 4$ as an illustration. For case 1, i.e., the Dirichlet boundary condition, we double the data array size, and extend the array through $u(x + T) = -u(T - x)$. For case 3, i.e., the Neumann boundary condition, we also double the data array size, but extend the array through $u(x+T) = u(T-x)$. For other two cases, we quadruple the data array size and extend the array in a symmetric or skew-symmetric way. The extending for all kinds of boundary conditions are shown in Fig. 2.

2.4 Sketch of CUDA Implementation

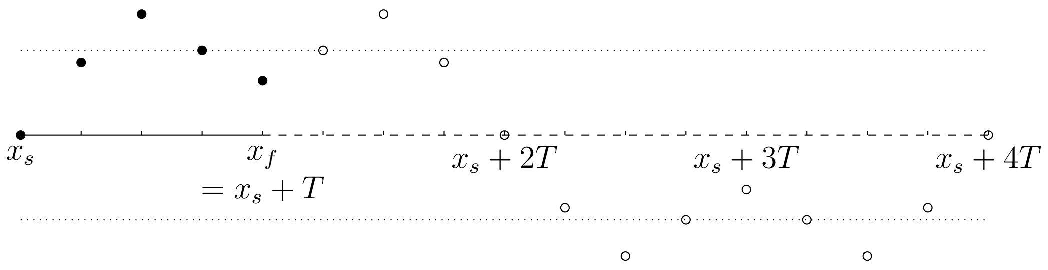
Algorithm 1 shows the sketch of our code. We have a few const values needed by each GPU kernel, so we wrap them into a single struct and transfer it to the GPU constant memory. In this way, we won't occupy too much registers, as a result we could launch more threads in a single CUDA multiprocessor to hide the global memory access latency as much as possible. For every GPU kernel, we try to maximize the memory throughput by adjusting the memory access pattern, these are all common optimization strategies for CUDA kernels, so we omit the details here.



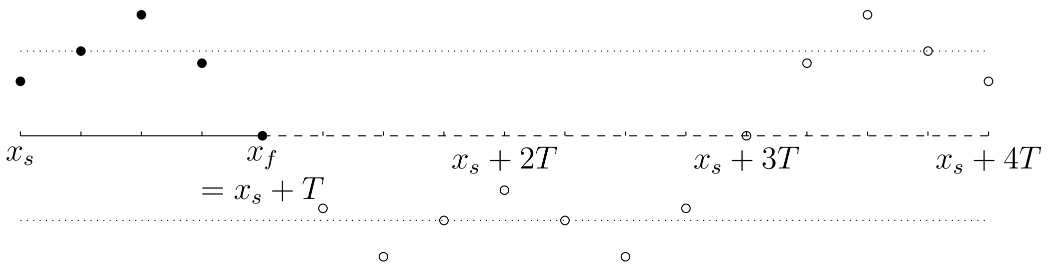
(a) Case 1. left: Dirichlet, right: Dirichlet



(b) Case 3. left: Neumann, right: Neumann



(c) Case 2. left: Dirichlet, right: Neumann



(d) Case 4. left: Neumann, right: Dirichlet

Fig. 2: Extending the original data array

3 Experiments and Analysis

3.1 Testing platform

The accuracy and performance tests for our CUDA based Helmholtz solver are conducted on a computer which equipped with a NVIDIA GeForce GTX TITAN card. Table 1 shows the major configuration of our testing platform.

Table 1: Testing platform

Operating System	CentOS
CPU	1x Intel(R) CPU i7-4770K, 4 cores, 3.5GHz
Host memory	32 GB
GPU	1x Nvidia(R) GeForce GTX TITAN, 2688 CUDA cores, 837MHz
GPU memory	6GB

Algorithm 1 CUDA based Helmholtz Solver

```
function SOLVER_3D_GPU( )
    Prepare a const struct (wrapped some const values) in CPU memory
    Transfer the const struct to GPU const memory by calling cudaMemcpyToSymbol( )
    for all Coordinate Directions do
        Launch GPU kernel to deal with the non-zero Dirichlet and Neumann boundary conditions
    end for
    Allocate enough space on GPU memory to store the extended data array
    for all Coordinate Directions do
        Launch GPU kernel to extend the original data array
    end for
    Call CUFFT to execute FFT on the extended data array
    Divide the extended array by a value generate by $\lambda$ and its position in the array (This is the essential step in solving Helmholtz equation)
    Call CUFFT to execute inverse FFT on the extended data array
    Launch GPU kernel to store the results in the original array
end function
```

We use compiler GCC with the -O2 flag to compile the CPU side code. For the comparison with FISHPACK, we compile FISHPACK using gfortran and the -O2 -fdefault-real-8 flags. We also implement a C wrapper for the FISHPACK function *hw3crt*. The source code can be downloaded from https://github.com/rmingming/cudahelmholtz/ (last accessed date Jun 10, 2015).

3.2 Experiments and discussion

We have tested a whole variety of functions with different boundary conditions. In this paper we only illustrate the test for one function and three different boundary conditions. The testing function is $u = \frac{1}{3}\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$ and let the rectangular area be $0 \le x \le 1, 0 \le y \le 1, 0 \le z \le 1$, λ be -1, so the Helmholtz equation would be

$$\Delta u - u = -\frac{1}{3}(1 + 12\pi^2)\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$$

We test three different boundary cases. The first case is all periodic boundary conditions in each coordinate direction. The second case is Dirichlet boundary condition in y axis and periodic boundary conditions in x, z axes. The third case is Neumann boundary condition in x axis and periodic boundary conditions in y, z axes. In the following, we briefly call them Periodic, Dirichlet, and Neumann respectively.

First we need to illustrate that our code gives enough accurate solution. We have a discretization error of $O(h^2)$, where h is the grid panel width, for instance, $h := (x_f - x_s)/N_x$ in x coordinate direction. Let $N = 32, 64, 128, 256, 512$ respectively, Table 2 gives the maximum error at all grid points. It shows that the solution is accurate enough and the order of error is indeed $O(h^2)$. The order of error is computed through the following formula:

$$\text{Error order}(N) = \log_2 \frac{\text{Error}(N)}{\text{Error}(2N)}$$

We just list the CPU results (by using FISHPACK) for the Dirichlet case, the other two cases are basically the same so we omit them. Compared with the results of FISHPACK, we can observe a tiny difference, the reason is that we use different method (compared to FISHPACK) to compute the DFT and its variants. For the two methods, the roundoff error is different and accumulated, but they all give enough accurate solutions.

Table 2: Maximum error and order of error

	Periodic		Dirichlet				Neumann	
	GPU		CPU		GPU		GPU	
N	Error	Error order	Error	Error order	Error	Error order	Error	Error order
32	0.00106398	2.00205807	0.0010639756230	2.00205807	0.0010639756230	2.00205807	0.00098352	1.99375369
64	0.00026561	2.00051425	0.0002656147244	2.00051425	0.0002656147244	2.00051425	0.00024695	2.00048120
128	0.00006638	2.00012854	0.0000663800155	2.00012854	0.0000663800155	2.00012854	0.00006172	2.00012031
256	0.00001659	2.00003220	0.0000165935252	2.00003219	0.0000165935253	2.00003220	0.00001543	2.00003014
512	0.00000415	—	0.0000041482891	—	0.0000041482888	—	0.00000386	—

Table 3 shows the running time for solving the testing problem. We can see that the CPU time for solving Periodic case is less than that for solving the Dirichlet case (for example when $N = 512$, they are 12.9 seconds and 20.4 seconds respectively) and the Neumann case, that's because in our algorithm the computation of DST or DCT is difficult than that of DFT. As a comparison, *hw3crt* also need more time to compute the Dirichlet case and Neumann case, but essentially they adopt different algorithms. It is obvious that when N is large, using our CUDA based Helmholtz solver can save us a lot of execution time.

Table 3: Execution time (in seconds)

	Periodic		Dirichlet		Neumann	
N	CPU	GPU	CPU	GPU	CPU	GPU
32	0.001523	0.001060	0.001679	0.001055	0.001593	0.001019
64	0.012632	0.002667	0.013869	0.002727	0.012892	0.002862
128	0.108058	0.008140	0.119173	0.014381	0.121248	0.012493
256	0.890860	0.050599	1.002024	0.105117	1.321821	0.093812
512	12.936950	0.392494	20.426926	0.717741	21.320410	0.750475

Fig. 3 shows the speedup of our code compared to *hw3crt*. For $N = 512$, we can achieve about 30 times speedup. For $N = 32$ and $N = 64$, we can see the speedup is basically the same for all three cases. The reason is that when N is small, we can not use all the GPU cores in some kernels, and the extra overhead is relatively high, the total execution time is majorly dependent on the extra overhead.

4 Conclusion and Future Work

We implement the CUDA based Helmholtz equation solver in three dimensional space and achieve about 30 times speedup compared to using the *hw3crt* from the FISHPACK software package. Our algorithm can deal with five different boundary conditions which are common in practice. This is achieved by transforming the computation of DST/DCT to DFT (FFT), since CUDA only provides the FFT library.

There are some improvements can be made in the next step. First, the CUDA code can be further optimized. In the present code, we only consider some obvious optimization techniques

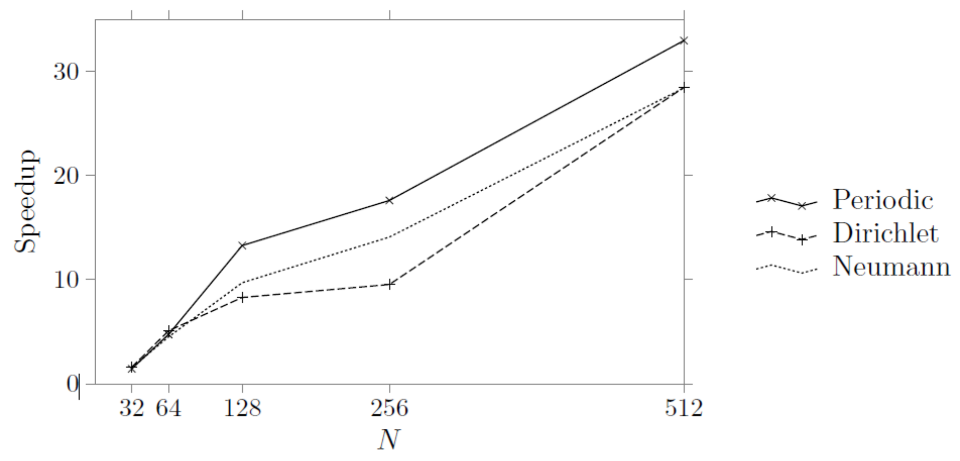


Fig. 3: Speedup

which are common in programming using CUDA. Second, the algorithm for the DST/DCT can be replaced. In our code, we compute DST/DCT by extending the original data array and then computing DFT of the extended array. The drawback of this strategy is that we need to compute the FFT of a much larger data array. This will take more GPU time and occupy larger device memory. There are other ways to compute DST/DCT which don't need to extend the data array, but these methods often need a pre- and post- processes on the data array. We will evaluate these strategies and implement them in the future.

Acknowledgement

This work was partially supported by NSF of China (Grant numbers: 61373018, 11301288), Program for New Century Excellent Talents in University (Grant number: NCET130301), and the Fundamental Research Funds for the Central Universities (Grant number: 65141021). This work was also partially supported by the NVIDIA GPU Education Center Program.

References

[1] William F. Ames, Numerical Methods for Partial Differential Equations, 2nd ed., Academic Press, 1977.

[2] J. Adams and P. Swarztrauber and R. Sweet, Elliptic Problem Solvers, Academic Press, 1981, pp. 187-190.

[3] W. Press, S. Teukolsky, W. Vetterling and B. Flannery, Numerical Recipes: The Art of Scientific Computing (3rd ed.), Cambridge University Press, 2007.

[4] M. Frigo and S. Johnson, The Design and Implementation of FFTW3, Proceedings of the IEEE (2005) 93 (2), pp. 216-231.

[5] NVIDIA CUDA Toolkit Documentation. Available at http://docs.nvidia.com/cuda/ (last accessed date Jun 10, 2015).

[6] J. Wu and J. JaJa, it An optimized FFT-based direct Poisson solver on CUDA GPUs. IEEE Transactions on Parallel and Distributed Systems, (2014) 25 (3), pp. 550-559.

[7] J. Wu and J. JaJa, High Performance FFT Based Poisson Solver on a CPU-GPU Heterogeneous Platform, International Parallel and Distributed Processing Symposium (IPDPS), May 2013, Boston, Massachusetts.