GPU and GPGPU Programming

Shuai Lu 170742

A CUDA implementation of 2D Possion's equation solver

Introduction

Partial differential equations can be seen everywhere in our life. Since the corresponding analytical solution is difficult to obtain, the computer is generally used to obtain the numerical solution. Numerical solution requires meshing the area according to certain rules and selecting appropriate numerical methods to solve the equations, such as: finite element method, finite difference method, finite volume method, etc.

However, with the increasing complexity of problems and the requirement of high efficiency, the traditional computing model can no longer meet the our expectation. So high-performance parallel computers are designed and various parallel computing methods, such as: GPU, MPI and OPENMP, are proposed and applied on numerical computing.

This project aim at implementing a solver for 2D Possion's equation based on CUDA.

Poisson equation

Poisson's equation is an elliptic partial differential equation, which has a wide range of uses in theoretical physics. For example, the solution of the Poisson equation is the potential field caused by a given charge or mass density distribution; the potential field is known, and then the electrostatic or gravitational (force) field can be calculated. It is a generalization of Laplace's equation and is also very common in physics. The equation is named after the French mathematician and physicist Simon Dennis Poisson[1].

Poisson's equation is

$$-\Delta u = f$$

where Δ is the Laplace operator, and f and u are real or complex-valued functions on a manifold. Usually, f is given and u is sought. When the manifold is Euclidean space, the Laplace operator is often denoted as ∇^2 and so Poisson's equation is frequently written as

$$-\nabla^2 u = f.$$

In two-dimensional Cartesian coordinates, it takes the form

$$-\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) u(x,y) = f(x,y)$$

When $f = 0$ identically we obtain Laplace's equation.

Usually it is difficult to find an analytical solution for Poisson's equation. So we need discrete it and try to get a numerical solution. For example, using the finite difference numerical method to discretize the 2 -dimensional Poisson equation (assuming a uniform spatial discretization, $\Delta x = \Delta y = h$) on an $m \times n$ grid gives the following formula[2]:

$$\frac{\partial^2}{\partial x^2} u(x,y) = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2}$$

$$\frac{\partial^2}{\partial y^2} u(x,y) = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2}$$

$$-\left(\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2}\right) = f\left(x_i, y_j\right)$$

So the Poisson's equation can be represented by

$$-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f\left(x_i, y_j\right)$$

where $2 \leq i \leq m-1$ and $2 \leq j \leq n-1$. The preferred arrangement of the solution vector is to use natural ordering which, prior to removing boundary elements, would look like:

$$\vec{u} = [u_{11}, u_{21}, \ldots, u_{m1}, u_{12}, u_{22}, \ldots, u_{m2}, \ldots, u_{mn}]^T$$

This will result in an $mn \times mn$ linear system:

$$A\vec{u} = \vec{b}$$

where

$$A = \begin{bmatrix} D & -I & 0 & 0 & 0 & \ldots & 0 \\ -I & D & -I & 0 & 0 & \ldots & 0 \\ 0 & -I & D & -I & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & -I & D & -I & 0 \\ 0 & \ldots & \ldots & 0 & -I & D & -I \\ 0 & \ldots & \ldots & \ldots & 0 & -I & D \end{bmatrix}$$

I is the $m \times m$ identity matrix, and D, also $m \times m$, is given by:

$$D = \begin{bmatrix} 4 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 4 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 4 & -1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -1 & 4 & -1 & 0 \\ 0 & \cdots & \cdots & 0 & -1 & 4 & -1 \\ 0 & \cdots & \cdots & \cdots & 0 & -1 & 4 \end{bmatrix},$$

and $\vec{b}$ is defined by

$$\vec{b} = -\Delta x^2 \left[f_{11}, f_{21}, \ldots, f_{m1}, f_{12}, f_{22}, \ldots, f_{m2}, \ldots, f_{mn}\right]^T.$$

Numerical Method

For solving the linear system:

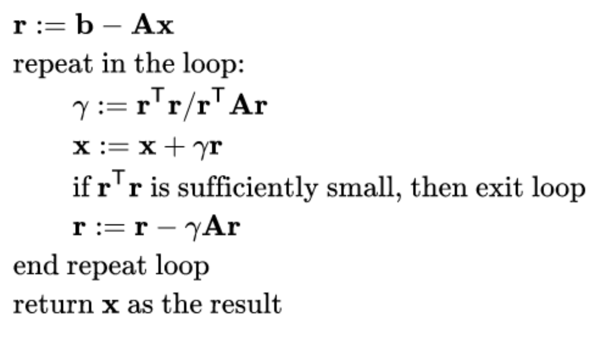
$$A\vec{u} = \vec{b}$$

some numerical methods are needed.

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent. It can be used to solve a system of linear equations.

$$A\mathbf{x} - \mathbf{b} = 0$$

The algorithm gradient descent method is shown in Fig.1[3].



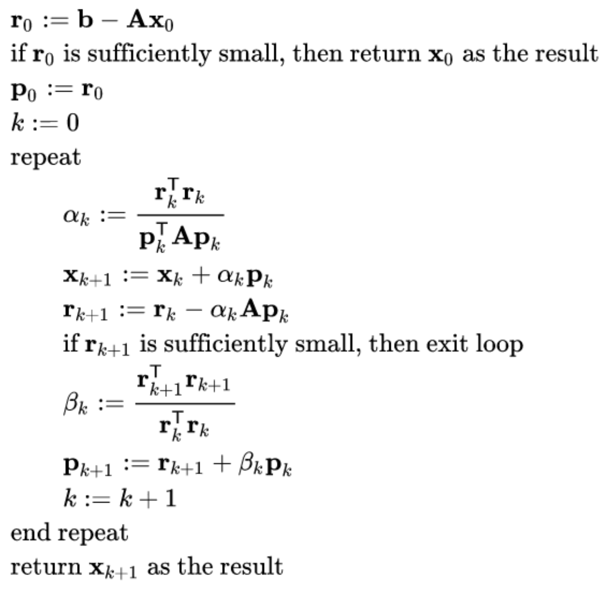
```
$\mathbf{r} := \mathbf{b} - \mathbf{Ax}$
repeat in the loop:
    $\gamma := \mathbf{r}^\mathsf{T}\mathbf{r}/\mathbf{r}^\mathsf{T}\mathbf{Ar}$
    $\mathbf{x} := \mathbf{x} + \gamma\mathbf{r}$
    if $\mathbf{r}^\mathsf{T}\mathbf{r}$ is sufficiently small, then exit loop
    $\mathbf{r} := \mathbf{r} - \gamma\mathbf{Ar}$
end repeat loop
return $\mathbf{x}$ as the result
```

Figure 1: The algorithm of gradient descent method

The number of gradient descent iterations is commonly proportional to the spectral condition number $\kappa(A)$ of the system matrix A, which makes it not that efficient.

Since the A matrix of the linear system is sparse, conjugate gradient method would be a alternative candidate to solve this equation. Conjugate gradient method is one of the most popular method which can applicable to sparse systems that are too large to be handled by a direct implementation. Different from gradient descent method, conjugate gradient method always move along the directions which are conjugate to each other. The algorithm of conjugate gradient method is shown in Fig.2[3].

The convergence of conjugate gradient method is typically determined by a square root of condition



```
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
    $\alpha_k := \frac{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
    if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop
    $\beta_k := \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
    $k := k + 1$
end repeat
return $\mathbf{x}_{k+1}$ as the result
```

Figure 2: The algorithm of conjugate gradient method

number $\kappa(A)$.

Result and conclusion

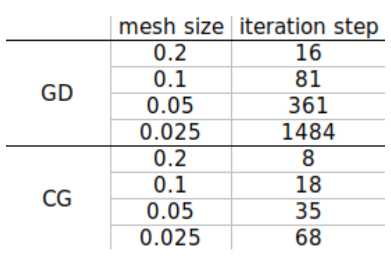
Both gradient descent method and conjugate gradient method are implemented in CPU and GPU version. The Poisson's equation is:

$$\nabla^2 u(x,y) = f(x,y) = 1.25\exp(x + y/2)$$

The analytical solution for this equation is

$$u(x,y) = \exp(x + y/2)$$

And the Dirichlet boundary conditions are equal to the analytical solution on the boundaries.
The comparison of different mesh size and iteration steps for different method is shown in Fig.3. The plot is shown in Fig.4. It is obvious that the iteration steps of gradient descent method increase dramatically with the refine of mesh while that of conjugate gradient method increase slightly.



	mesh size	iteration step
GD	0.2	16
	0.1	81
	0.05	361
	0.025	1484
CG	0.2	8
	0.1	18
	0.05	35
	0.025	68

Figure 3: The comparison of different mesh size and iteration steps

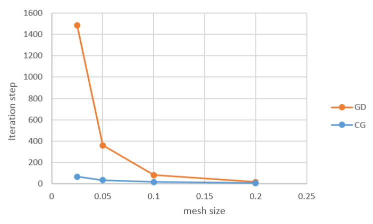
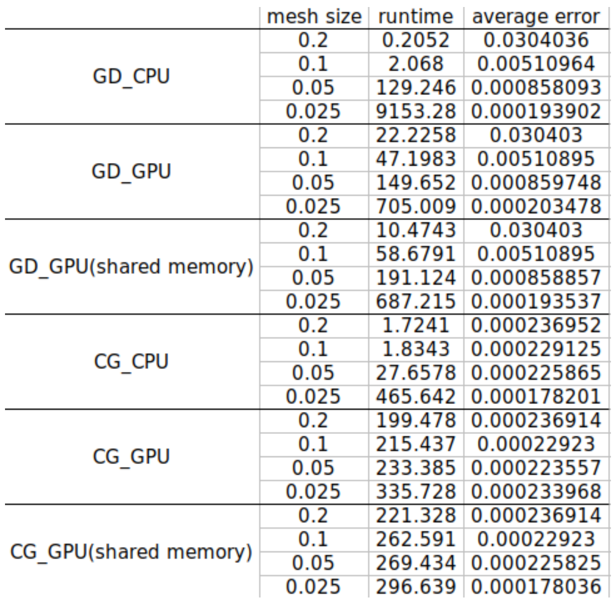


Figure 4: The comparison of different mesh size and iteration steps

The runtime and average error of different implementations are shown in Fig.5. The plots are shown in Fig.6 and 7. The runtime of gradient descent method(CPU) is the smallest but it will increase rapidly with the refined of mesh. The performance of gradient descent method(GPU) and gradient descent method(GPU with shared memory) are similar. The performance of conjugate gradient method(CPU) is similar with gradient descent method(CPU). The runtime of conjugate gradient method(GPU with shared memory) is larger than conjugate gradient method(GPU) when the mesh size is large. However, the advantage of shared memory appear when the mesh size is small.



	mesh size	runtime	average error
GD_CPU	0.2	0.2052	0.0304036
	0.1	2.068	0.00510964
	0.05	129.246	0.000858093
	0.025	9153.28	0.000193902
GD_GPU	0.2	22.2258	0.030403
	0.1	47.1983	0.00510895
	0.05	149.652	0.000859748
	0.025	705.009	0.000203478
GD_GPU(shared memory)	0.2	10.4743	0.030403
	0.1	58.6791	0.00510895
	0.05	191.124	0.000858857
	0.025	687.215	0.000193537
CG_CPU	0.2	1.7241	0.000236952
	0.1	1.8343	0.000229125
	0.05	27.6578	0.000225865
	0.025	465.642	0.000178201
CG_GPU	0.2	199.478	0.000236914
	0.1	215.437	0.00022923
	0.05	233.385	0.000223557
	0.025	335.728	0.000233968
CG_GPU(shared memory)	0.2	221.328	0.000236914
	0.1	262.591	0.00022923
	0.05	269.434	0.000225825
	0.025	296.639	0.000178036

Figure 5: The runtime and average error of different implementations

The exact solution is shown in Fig.8. The error of different implementations are shown in Fig.9-12. Overall, conjugate gradient method not only converge faster but also has less error than gradient descent method.

Reference

[1] Jackson, Julia A.; Mehl, James P.; Neuendorf, Klaus K. E., eds. (2005), Glossary of Geology, American Geological Institute, Springer, p. 503, ISBN 9780922152766.
[2] Hoffman, Joe (2001), "Chapter 9. Elliptic partial differential equations", Numerical Methods for Engineers and Scientists (2nd ed.), McGraw–Hill, ISBN 0-8247-0443-6.
[3] Bouwmeester, Henricus; Dougherty, Andrew; Knyazev, Andrew V. (2015). "Nonsymmetric Preconditioning for Conjugate Gradient and Steepest Descent Methods". Procedia Computer Science. 51: 276–285.

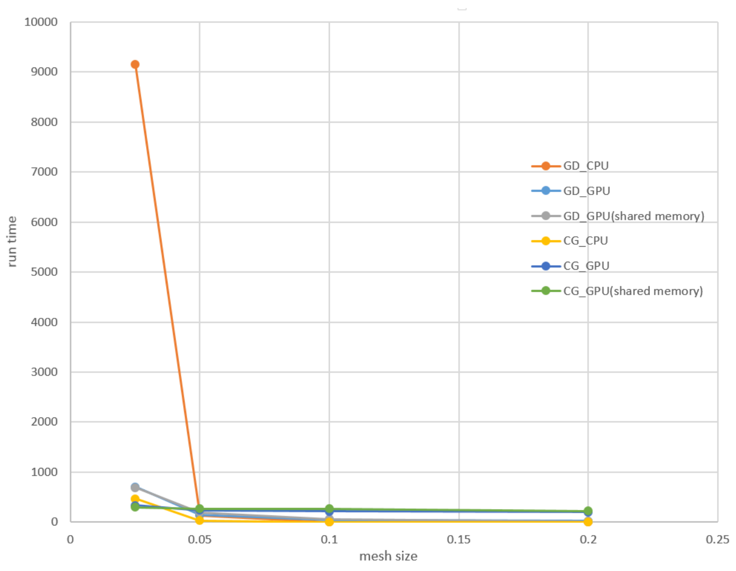


Figure 6: The runtime of different implementations

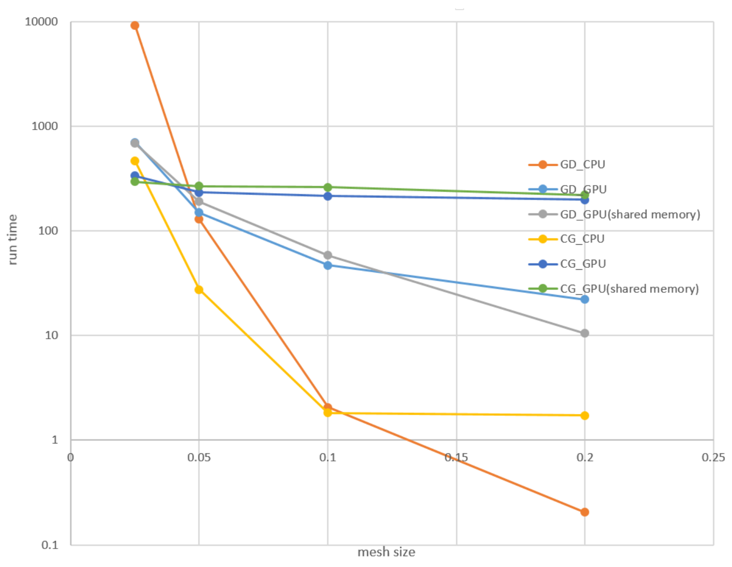


Figure 7: The runtime of different implementations

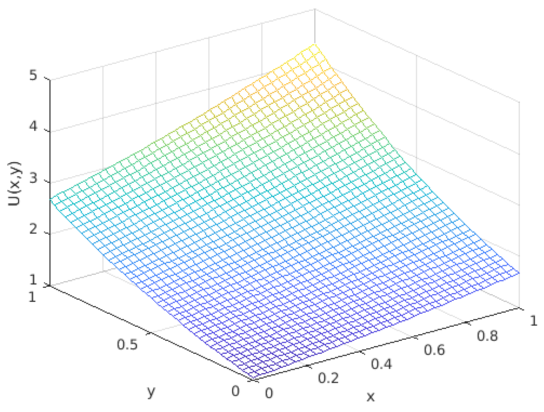


Figure 8: The exact solution

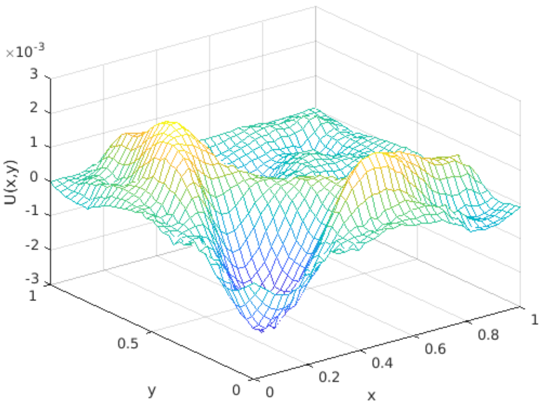


Figure 9: The error of conjugate gradient method(CPU)

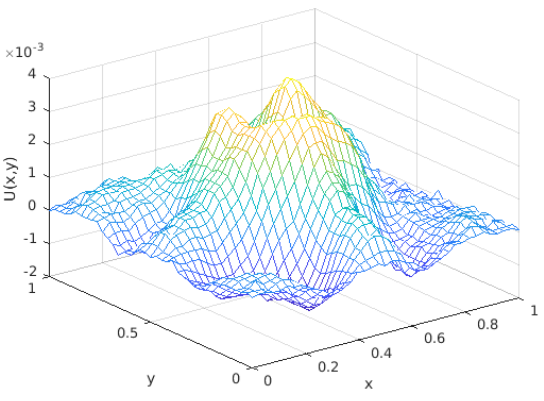


Figure 10: The error of conjugate gradient method(GPU)

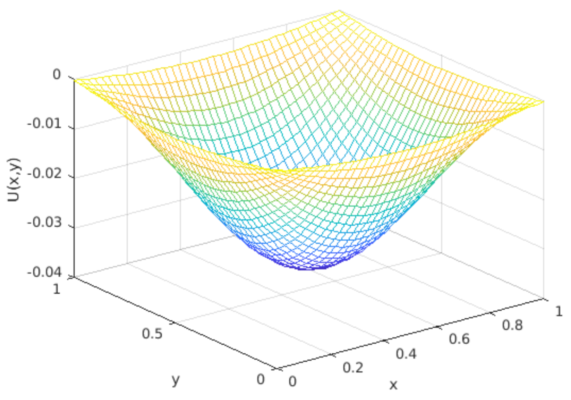


Figure 11: The error of gradient descent method(CPU)

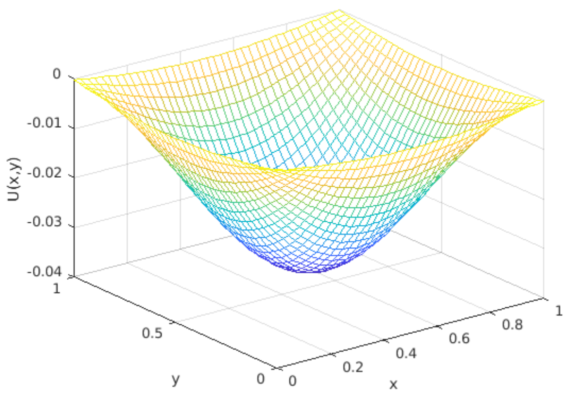


Figure 12: The error of gradient descent method(GPU)