Cuda Lattice Gauge Document

目录

1 Data **4**
1.1 Index of lattice . . . 4
1.1.1 UINT Index of lattice . . . 4
1.1.2 SIndex of lattice . . . 4
1.1.3 Index and boundary condition, a int2 or a uint2 structure . . . 4
1.1.4 Index walking . . . 5
1.2 CParemeters . . . 5

2 Update scheme **6**
2.1 HMC . . . 6
2.1.1 The fermion action . . . 6
2.1.2 Basic idea, force from gauge field . . . 7
2.1.3 Force of pseudofermions . . . 10
2.1.4 Solver in HMC . . . 11
2.1.5 Leap frog integrator . . . 14
2.1.6 A summary of HMC with pseudofermions . . . 15
2.2 Optimization of HMC . . . 16
2.2.1 Omelyan integrator . . . 16
2.2.2 Multi-Step Updator . . . 16
2.3 Sparse linear algebra solver . . . 16

3 Measurement **17**
3.1 Plaquette Energy . . . 17
3.2 Meson Correlator . . . 17
3.2.1 Meson Wave Function . . . 17
3.2.2 Meson Correlator . . . 17
3.2.3 Monte Carlo in Monte Carlo . . . 18

4 Programming **19**
4.1 cuda . . . 19
4.1.1 blocks and threads . . . 19
4.1.2 device member function . . . 19
4.1.3 device virtual member function . . . 22

5 Testing **25**

5.1 random number 25

1 Data

1.1 Index of lattice

1.1.1 UINT Index of lattice

Generally, in CLG, we have three kinds of indexes:

- **site index**
- **link index**
- **fat index**

Let the lattice have $V = L_x \times L_y \times L_z \times L_t$ sites.

Note: for $D = 3$, we assume $L_x = 1$, $L_{y,z,t} > 1$; for $D = 2$, we assume $L_x = L_y = 1$, $L_{z,t} > 1$.

For a site at (x, y, z, t)

$$siteIndex = x \times L_y \times L_z \times L_t + y \times L_z \times L_t + z \times L_t + t \tag{1}$$

For a link at direction dir, link with site at (x, y, z, t), and on a lattice with number of directions of links is $dirCount$,

$$linkIndex = siteIndex \times dirCount + dir \tag{2}$$

Note: we do NOT assume dimension equal number of links. For example for $D = 2$ triangle lattice, number of directions of links is 6, for $D = 2$ hexagon number of directions of links is 3. Only for square lattice, number of links equal dimension.

For a link at direction dir, link with site at (x, y, z, t), and on a lattice with number of directions of links is $dirCount$,

$$fatIndex = \begin{cases} siteIndex \times (dirCount + 1); & for\ site. \\ siteIndex \times (dirCount + 1) + (dir + 1); & for\ link \end{cases} \tag{3}$$

1.1.2 SIndex of lattice

1.1.3 Index and boundary condition, a int2 or a uint2 structure

In CLGLib, sometimes, the index function return a uint2 structure.

1.1.4 Index walking

1.2 CParemeters

2 Update scheme

2.1 HMC

HMC is abbreviation for hybrid Monte Carlo.

2.1.1 The fermion action

Cooperating with HMC, the fermion is usually the 'Pseudofermions'.

We begin with Eq. (1.85) and Eq. (1.86) of Ref. [1].

$$Z = \int \mathcal{D}[U] \prod_{f=1}^{N_f} \mathcal{D}[\bar{\psi}_f]\mathcal{D}[\psi_f] \exp\left(-S_G[U] - \sum_{f=1}^{N_f} \bar{\psi}_f \left(\hat{D}_f\right) \psi_f\right) \tag{4}$$

where $\hat{D}_f = D + m_f$. (Note that, there seems a typo in Eq. (1.85), we have $S_F = +\bar{\psi} D \psi$, see also Eq. (5.39) of Ref. [2] and Eq. (7.6) of Ref. [3], Eq. (3.75) of Ref. [1], etc.)

It can be evaluated as Eq. (1.86) of Ref. [1] (or Eq. (4.19) of Ref. [4]) (Note, there is another minus sign in Eq. (5.28) of Ref. [2])

$$\begin{aligned} \int \mathcal{D}\bar{\psi}\psi \exp\left(-\bar{\psi} A \psi\right) &= \det\left(A\right), \\ Z = \prod_{f=1}^{N_f} \det\left(\hat{D}_f\right) &\int \mathcal{D}[U] \exp\left(-S_G[U]\right). \end{aligned} \tag{5}$$

On the other hand, with the help of Gaussian integral of complex vectors Eq. (3.17) of Ref. [4]

$$\int d\mathbf{v}^\dagger d\mathbf{v} \exp(-\mathbf{v}^\dagger \mathbf{A} \mathbf{v}) = \pi^N \left(\det \mathbf{A}\right)^{-1} \tag{6}$$

which is (3.31) of Ref. [1]

$$\frac{1}{\det(\mathbf{A})} = \int \mathcal{D}[\eta] \exp(-\eta^\dagger \mathbf{A} \eta) \tag{7}$$

where η now is a complex Bosonic field, and the normalization

$$\mathcal{D}[\eta] = \prod \frac{d\mathrm{Re}(\eta_i) d\mathrm{Im}(\eta_i)}{\pi}, \quad 1 = \int \mathcal{D}[\eta] \exp(-\eta^\dagger \eta) \tag{8}$$

is assumed. With the condition such that

$$\lambda(\mathbf{A} + \mathbf{A}^\dagger) > 0. \tag{9}$$

where $\lambda(\mathbf{M})$ denoted as eigen-values of $\mathbf{M}$.

We now, concentrate on two degenerate fermion flavours. i.e. considering

$$S_F = \bar{\psi}_u \hat{D} \psi_u + \bar{\psi}_d \hat{D} \psi_d. \tag{10}$$

Using $\det(DD^\dagger) = \det(D)\det(D^\dagger)$ and $\det(M^{-1}) = (\det(M))^{-1}$ and $\det(D) = \det(D^\dagger)$ (**Only for Wilson Fermions or γ_5-hermiticity fermions,** $\hat{D}^\dagger = \gamma_5 D \gamma_5 + m = \gamma_5(D + m)\gamma_5 = \gamma_5 \hat{D} \gamma_5$**, and** $\det(\hat{D}^\dagger) = \det(\gamma_5)\det(\hat{D})\det(\gamma_5) = \det(\hat{D})$**.** See also Ref. [5].), one can show Eq. (8.9) of Ref. [2] (Eq. (2.77) of Ref. [6])

$$\int \mathcal{D}[\bar{\psi}]\mathcal{D}[\psi] \exp\left(-\bar{\psi}_u \hat{D} \psi_u - \bar{\psi}_d \hat{D} \psi_d\right) = \det(\hat{D}\hat{D}^\dagger) = \int \mathcal{D}[\phi] \exp(-\phi^\dagger \left(\hat{D}\hat{D}^\dagger\right)^{-1} \phi) \tag{11}$$

where ϕ now is a complex Bosnic field.

So, generally, we are using HMC to evaluate the action with 'Pseudofermions', or in other words, we are working with an action including only gauge and bosons.

$$S = S_G + S_{pf} = S_G + \phi^\dagger \left(\hat{D}\hat{D}^\dagger\right)^{-1} \phi \tag{12}$$

where pf is short for pseudofermion.

2.1.2 Basic idea, force from gauge field

The basic idea is to use a molecular dynamics simulation, i.e, it is a integration of Langevin equation.

Treating $SU(N)$ matrix U on links as coordinate, HMC will generate a pair of configurations, (P, U), where P is momentum and $P \in \mathfrak{su}(N)$.

One can:

1. Create a random $P = i \sum_a \omega_a T_a$, where $\omega_a \in \mathbb{R}$.
2. Obtain $\dot{P}$, $\dot{U}$. Note that, dot is $d/d\tau$, where τ is 'Markov time'.
3. Numerically evaluate the differential equation, and use a Metropolis accept / reject to update.

- About the randomized P

The randomized P is chosen according to normal distribution $\exp(-P^2/2)$

Note that, here P corresponds to Q, not U, for $U = \exp(i \sum q_a T^a)$, there are 8 **real** variables denoting as ω_i.

Using $P = \sum \omega_a T^a$, $tr((T^a)\cdot(T^b)) = \frac{1}{2}\delta_{ab}$. So one have $\frac{1}{2}\sum_a \omega_a^2 = tr[P^2]$.

It is usually written as distribution $\exp(-tr(P^2))$ (where P is a matrix, and $tr[P^2] = \frac{1}{2}p^2$ where $p = (\omega_1, \omega_2, \ldots, \omega_8)$).

Using the property of normal distribution

$$\begin{aligned} &if\ \{X\} \sim N(\mu_X, \sigma_X^2),\ \ \{Y\} \sim N(\mu_Y, \sigma_Y^2), \\ &\{X+Y\} \sim N(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2). \end{aligned} \tag{13}$$

One can randomize ω_a using $\exp(-\omega_a\omega_a)$. Then using $P = \frac{1}{\sqrt{8N}}\sum \omega_a T^i$, where N is the number of links.

Note: Here is a difference between Refs. [2] and [1] and Bridge++ [7]

Note, by Eq. (8.16) of Ref. [2], $P^2 = \sum_{n\in\Lambda} P^2(n)$, so when the lattice is large, P become very small. See also the definition of $\langle P, P\rangle$ below Eq. (2.42) of Ref. [1].

However, in Bridge++, it uses distribution $\exp(-tr(P^2)/DOF)$, where 'DOF' is the degrees of freedom, i.e., number of links.

We use the distribution same as in Bridge++. Imagining that for a very small (hot) $\beta \to 0$, the force is also almost 0 so momentum is unchanged when evolution. Considering a very large lattice such that the momentum is very small when using distribution $\exp(-tr(P^2))$, the gauge field will stay near the initial value rather then becoming hot (randomized). So we think it should be $\exp(-tr(P^2)/DOF)$.

- Force

Defined by Newton, dp/dt is a force, so $\dot{P}$ is called 'force'. See Eqs. (2.53), (2.56) and (2.57) of Ref. [1], for $SU(N)$,

$$\begin{aligned} S_G[U_\mu(n)] &= -\frac{\beta}{N}\mathrm{Retr}\left[U_\mu(n)\Sigma_\mu^\dagger(n)\right] \\ \Sigma_\mu(n) &= \sum_{\mu\neq\nu}\left(U_\nu(n)U_\mu(n+a\nu)U_\nu^{-1}(n+a\mu) + U_\nu^{-1}(n-a\nu)U_\mu(n-a\nu)U_\nu(n-a\nu+a\mu)\right) \end{aligned} \tag{14}$$

Note that $S_G \neq \sum_{\mu,n} S_G[U_\mu(n)]$. $S_G[U_\mu(n)]$ is convenient for derivate which collecting all terms related to the specified bond. For plaquettes with 4 edges, $S_G = \frac{1}{4}\sum_{\mu,n} S_G[U_\mu(n)]$.

S_G the action for a particular $U_\mu(n)$. Σ is the 'staple'. The staple for $U_\mu(n)$ is independent of $U_\mu(n)$, denoting

$$U_\mu(n) = \exp\left(i\sum_a \omega_a(\mu, n)T_a\right)U_\mu^0(n) \tag{15}$$

so

$$\begin{aligned}
&\frac{\partial}{\partial\omega_a(\mu,n)}S_G = -\frac{\beta}{2N}\mathrm{Retr}\left[\frac{\partial}{\partial\omega_a}U_\mu(n)\Sigma_\mu^\dagger(n)\right] = -\frac{\beta}{2N}\mathrm{tr}\left[\frac{\partial}{\partial\omega_a}\left(U_\mu(n)\Sigma_\mu^\dagger(n)+\Sigma_\mu(n)U_\mu^\dagger(n)\right)\right]\\
&= -i\frac{\beta}{2N}\mathrm{tr}\left[T_aU_\mu(n)\Sigma_\mu^\dagger(n)-\Sigma_\mu(n)T_a^\dagger U_\mu^\dagger(n)\right] = -i\frac{\beta}{2N}\mathrm{tr}\left[T_a\left(U_\mu(n)\Sigma_\mu^\dagger(n)-\Sigma_\mu(n)U_\mu^\dagger(n)\right)\right]\\
&= -\frac{\beta}{N}i\mathrm{Im}\ \mathrm{tr}\left[T_aU_\mu(n)\Sigma_\mu^\dagger(n)\right]
\end{aligned} \tag{16}$$

This is the Eq. (8.41) of Ref. [2].

Using (Checked by Mathematica that Eq. (8.42) of Ref. [2] is incompatable with our notation, but replacing the $UA-A^\dagger U^\dagger$ of Eq. (8.42) with $\{UA\}_{TA}$ is correct. Also, Eq. (2.58) of Ref. [1] is different from ours, in our formulism, it is correct by replacing $2T_a\mathrm{Re}[tr[T_a\cdot W]]$ of Eq. (2.58) with $2iT_a\mathrm{Im}[tr[T_a\cdot W]]$)

$$\begin{aligned}
&\sum_a \mathrm{tr}\left[T_a\left(U_\mu(n)\Sigma_\mu^\dagger(n)-\Sigma_\mu(n)U_\mu^\dagger(n)\right)\right]T_a = \{U_\mu(n)\Sigma_\mu^\dagger(n)\}_{TA}\\
&\{W\}_{TA} = \frac{W-W^\dagger}{2} - \mathrm{tr}\left(\frac{W-W^\dagger}{2N}\right)\mathbb{I}
\end{aligned} \tag{17}$$

where $\mathbb{I}$ is identity matrix. Therefor

$$\begin{aligned}
&\dot\omega_a = -\frac{\partial}{\partial\omega_a(\mu,n)}S_G\\
&F_\mu(x) = \dot P_\mu(x) = i\sum\dot\omega_aT_a = -i\frac{\partial}{\partial\omega_a(\mu,n)}S_GT_a = -\frac{\beta}{2N}\{U_\mu(n)\Sigma_\mu^\dagger(n)\}_{TA}
\end{aligned} \tag{18}$$

Note that, $\{U_\mu(n)\Sigma_\mu^\dagger(n)\}_{TA} = -2i\sum_a T_a\mathrm{Im}[tr[T_a\cdot W]]$, so $\dot\omega_a = \frac{\beta}{N}\mathrm{Im}[tr[T_a\cdot W]]$ is still a **real** number.

Eq. (18) is same as Eqs. (2.53), (2.56) and (2.57) of Ref. [1].

- Integrator

  Knowing $\dot P$, and $\dot U$, to obtain U and P is simply

$$U(\tau+d\tau)\approx\dot U d\tau+U(\tau),\quad P(\tau+d\tau)\approx\dot P d\tau+P(\tau) \tag{19}$$

A more accurate calculation is done by integrator, for example, the leap frog integrator, the M step leap frog integral is described in Ref. [2],

$$\epsilon = \tfrac{\tau}{M} \tag{20a}$$

$$U_\mu(x,(n+1)\epsilon) = U_\mu(x,n\epsilon)+\epsilon P_\mu(x,n\epsilon)+\tfrac{1}{2}F_\mu(x,n\epsilon)\epsilon^2 \tag{20b}$$

$$P_\mu(x,(n+1)\epsilon) = P_\mu(x,n\epsilon)+\tfrac{1}{2}\left(F_\mu(x,(n+1)\epsilon)+F_\mu(x,n\epsilon)\right)\epsilon \tag{20c}$$

So, knowing $U(n\epsilon)$ we can calculate $F(n\epsilon)$ using Eq. (18). Knowing $U(n\epsilon), P(n\epsilon), F(n\epsilon)$, we can calculate $U((n+1)\epsilon)$ using Eq. (20).b. Then we are able to calculate $F((n+1)\epsilon)$ again using Eq. (18). Then we can calculate $P((n+1)\epsilon)$ using Eq. (20).c.

2.1.3 Force of pseudofermions

For important sampling, one can generate both U and ϕ by e^{-S}. In molecular dynamics simulation, it can be simplified as:

1. Evaluate U use force of U and ϕ on U.
2. Evaluate ϕ use force of U and ϕ on ϕ.

The second step can be simplified as, generating random complex numbers ϕ according to $\exp(-\phi^\dagger \left(\hat{D}\hat{D}^\dagger\right)^{-1} \phi) = \exp(-\phi^\dagger (\hat{D}^\dagger)^{-1}\hat{D}^{-1}\phi)$. $D[U]$ is a function of U.

How to get randomized ϕ? Let χ be random **complex** numbers according to $\exp(-\chi^\dagger\chi)$. Let $\hat{D}^{-1}\phi = \chi$, ϕ is the random **complex** number satisfying distribution we want ($\exp(-\phi^\dagger(\hat{D}^\dagger)^{-1}\hat{D}^{-1}\phi)$). So, first get χ and then let $\phi = D\chi$.

Using the Wilson Fermion action

$$\begin{aligned} \hat{D} &= C(D+1) \\ D &= -\kappa \sum_\mu \left((1-\gamma_\mu)U_\mu(x)\delta_{x,x+\mu} + (1+\gamma_\mu)U_\mu^{-1}(x-\mu)\delta_{x,x-\mu}\right) \end{aligned} \tag{21}$$

with $C = m_f + (4/a)$ and $\kappa = 1/(2am_f + 8)$.

The force of ϕ on U is obtained as $\partial_{\omega_a} S_{pf}$. The result for Wilson Fermion action is shown in Eqs. (8.39), (8.44) and (8.45) of Ref. [2] as

$$\begin{aligned} &F = i\sum_a \dot{\omega}_a T_a = i\sum_a \left(-\partial_{\omega_a}\left(S_G[U_\mu(n)] + S_{pf}[U_\mu(n)]\right)\right) T_a = F_G + F_{pf}. \\ &F_{pf} = i\sum_a \left(-\partial_{\omega_a} S_{pf}[U_\mu(n)]\right) T_a = -i\sum_a T^a \frac{\partial}{\partial\omega_a}\left(\phi^\dagger \left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right). \\ &\frac{\partial}{\partial\omega_a}\left(\phi^\dagger \left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right) = -\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right)^\dagger \left(\frac{\partial D}{\partial\omega_\mu^a}\hat{D}^\dagger + \hat{D}\frac{\partial D^\dagger}{\partial\omega_\mu^a}\right)\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right). \\ &\frac{\partial\hat{D}}{\partial\omega_\mu^a} = \left(\frac{\partial D}{\partial\omega_\mu^a}\right)_{x_L.x_R} = -i\kappa\left\{(1-\gamma_\mu)T^a U_\mu(x)\delta_{x,x_L}\delta_{x,(x+\mu)_R} - (1+\gamma_\mu)U_\mu^{-1}(x)T^a\delta_{x,(x-\mu)_L}\delta_{x,x_R}\right\} \\ &\hat{D}^\dagger = \gamma_5\hat{D}\gamma_5, \quad \frac{\partial D^\dagger}{\partial\omega_\mu^i} = \gamma_5\frac{\partial D}{\partial\omega_\mu^a}\gamma_5 \end{aligned} \tag{22}$$

where F_G is force from U introduced in Sec. 2.1.2, T^a are $SU(3)$ generators. x_L, x_R are coordinate index of the left and right pseudofermion field. And

$$\begin{aligned} &U_\mu = \exp(i\sum_a \omega_\mu^a T^a)U_0, \quad \frac{\partial U_\mu}{\partial \omega_\mu^a} = iT^a U_\mu, \quad \frac{\partial U_\mu^\dagger}{\partial \omega_\mu^a} = -iU_\mu^\dagger T^a, \\ &(T^a)^\dagger = T^a, \quad \frac{\partial M^{-1}}{\partial \omega_\mu^a} = -M^{-1}\frac{\partial M}{\partial \omega_\mu^a}M^{-1} \end{aligned} \tag{23}$$

are used.

We can simplify it further by $\left(\hat{D}^\dagger(\hat{D}\hat{D}^\dagger)^{-1}\phi\right)^\dagger = \left((\hat{D}\hat{D}^\dagger)^{-1}\phi\right)^\dagger \hat{D}$, so

$$\begin{aligned} &\phi_1 = \left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right), \quad \phi_2 = \hat{D}^\dagger\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right) = D^{-1}\phi, \quad \phi_1^\dagger D = \phi_2^\dagger, \\ &\frac{\partial}{\partial\omega_a}\left(\phi^\dagger\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right) = -\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right)^\dagger\left(\frac{\partial D}{\partial\omega_\mu^a}\hat{D}^\dagger + \hat{D}\frac{\partial D^\dagger}{\partial\omega_\mu^a}\right)\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right) \\ &= -\left(\phi_1^\dagger\frac{\partial D}{\partial\omega_\mu^a}\phi_2 + \phi_2^\dagger\frac{\partial D^\dagger}{\partial\omega_\mu^a}\phi_1\right) = -2\mathrm{Re}\left[\left(\phi_1^\dagger\frac{\partial D}{\partial\omega_\mu^a}\phi_2\right)\right] \end{aligned} \tag{24}$$

and

$$\begin{aligned} &\frac{\partial D}{\partial\omega_\mu^a} = -i\kappa M_a, \\ &(M_a)_{x_L,x_R} = \left\{(1-\gamma_\mu)T^a U_\mu(x)\delta_{x,x_L}\delta_{x,(x+\mu)_R} - (1+\gamma_\mu)U_\mu^{-1}(x)T^a\delta_{x,(x-\mu)_L}\delta_{x,x_R}\right\} \\ &\frac{\partial}{\partial\omega_a}\left(\phi^\dagger\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right) = -2\kappa\mathrm{Im}\left[\left(\phi_1^\dagger M\phi_2\right)\right] \end{aligned} \tag{25}$$

Again, $\dot{\omega}$ is a **real** number, and

$$F_{pf} = -i\sum_a T^a\frac{\partial}{\partial\omega_a}\left(\phi^\dagger\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right) = 2i\kappa\sum_a \mathrm{Im}\left[\left(\phi_1^\dagger M_a\phi_2\right)\right]T_a \tag{26}$$

So we can calculate ϕ_1 first, then $\phi_2 = \hat{D}^\dagger\phi_1$. Then contract the spinor and color space with $\partial D/\partial\omega$.

Note that, D is changing when integrating the Langevin equation.

The last part is how to calculate $(\hat{D}\hat{D}^\dagger)^{-1}$.

2.1.4 Solver in HMC

To calculate $(\hat{D}\hat{D}^\dagger)^{-1}$, we need a solver. The detail of solvers will be introduced in Sec. 2.3. Here we establish a simple introduction.

Let M be a matrix operating on a vector, for example, $M = (\hat{D}\hat{D}^{\dagger})$, the goal of the solver is to find x such $b = M \cdot x$, and therefor $x = (\hat{D}\hat{D}^{\dagger})^{-1}b$.

We first introduce the CG algorithm for real vector and real matrix, define

$$Q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \cdot A \cdot \mathbf{x} - \mathbf{x}^T\mathbf{b}. \tag{27}$$

so that one can try to find the minimum of Q, and at the minimum

$$\frac{\partial}{\partial \mathbf{x}}Q(\mathbf{x}) = 0 = A \cdot \mathbf{x} - \mathbf{b}. \tag{28}$$

To find the minimum, one can use gradient. Starting from a random point on a curve, calculate the falling speed and move it until it is stable.

For complex vector, one can use BiCGStab in Table. 6.2 in Ref. [2]. It can be described as

```
1
2  CField* pX, pR, pRH, pV, pP, pS, pT;
3
4  //use it to estimate relative error, a->Dot(b) means a_dagger . b
5  Real fBLength = pFieldB->Dot(pFieldB);
6
7  //Using b as the guess, (Assuming M is near identity?)
8  pFieldB->CopyTo(pX);
9
10 //r_0 = b - A x_0
11 pFieldB->CopyTo(pR);
12 pR->ApplyOperator(uiM, pGaugeFeild); //A x_0, Note D operator need gauge field
13 pR->ScalarMultply(-1); //-A x_0
14 pR->AxpyPlus(pX); //b - A x_0
15 pR->CopyTo(pRh);
16
17 Real rho = 0;
18 Real last_rho = 0;
19 Real alpha = 0;
20 Real beta = 0;
21 Real omega = 0;
22
23 for (UINT i = 0; i < m_uiReTry; ++i)
24 {
25     for (UINT j = 0; j < m_uiStepCount * m_uiDevationCheck; ++j)
26     {
27         //One step
28         rho = _cuCabsf(m_pRh->Dot(m_pR));//rho = rh dot r(i-1), if rho = 0, failed (assume will not)
```

```
29
30          if (0 == j) //if is the first iteration, p=r(i-1)
31          {
32              pR->CopyTo(pP);
33          }
34          else //if not the first iteration,
35          {
36              //beta = last_alpha * rho /(last_omega * last_rho)
37              beta = alpha * rho / (omega * last_rho);
38              //p(i) = r(i-1)+beta( p(i-1) - last_omega v(i-1) )
39              pV->ScalarMultply(omega);
40              pP->AxpyMinus(pV); //p = p - v
41              pP->ScalarMultply(beta);
42              pP->AxpyPlus(pR);
43          }
44
45          //v(i) = A p(i)
46          pP->CopyTo(pV);
47          pV->ApplyOperator(uiM, pGaugeFeild);
48
49          alpha = rho / (_cuCabsf(pRh->Dot(pV)));//alpha = rho / (rh dot v(i))
50
51          //s=r(i-1) - alpha v(i)
52          pR->CopyTo(pS);
53          pS->Axpy(-alpha, pV);
54
55          //t=As
56          pS->CopyTo(pT);
57          pT->ApplyOperator(uiM, pGaugeFeild);
58
59          omega = _cuCabsf(pT->Dot(pS)) / _cuCabsf(pT->Dot(pT));//omega = ts / tt
60
61          //r(i)=s-omega t
62          pS->CopyTo(pR);
63          if (0 == (j - 1) \% m_uiDevationCheck)
64          {
65              //Normal of S is small, then stop
66              Real fDeviation = _cuCabsf(pS->Dot(pS)) / fBLength;
67              appParanoiac(_T("CSLASolverBiCGStab::Solve deviation: restart:\%d, iteration:\%d,
                     deviation:\%f\n"), i, j, fDeviation);
68              if (fDeviation < m_fAccuracy)
69              {
70                  pX->Axpy(alpha, pP);
71                  pX->CopyTo(pFieldX);
72                  return;
73              }
74          }
```

```
75
76          pR->Axpy(-omega, pT);
77
78          //x(i)=x(i-1) + alpha p + omega s
79          pX->Axpy(alpha, pP);
80          pX->Axpy(omega, pS);
81
82          last_rho = rho;//last_rho = rho
83      }
84
85      //we are here, means we do not converge.
86      //we need to restart with a new guess, we use last X
87      pX->CopyTo(pR);
88
89      pR->ApplyOperator(uiM, pGaugeFeild); //A x_0
90      pR->ScalarMultply(-1); //-A x_0
91      pR->AxpyPlus(pX); //b - A x_0
92      pR->CopyTo(pRh);
93  }
94
95
96  //The solver failed.
```

2.1.5 Leap frog integrator

In Sec. 2.1.2, the basic idea is introduced. However, the implementation is slightly different.

$$U_\mu(0,x) = gauge(x), \quad P_\mu(0,x) = \sum_a r_a(\mu,x)T_a \tag{29a}$$

$$F_\mu(n\epsilon,x) = -\frac{\beta}{2N}\{U_\mu(n\epsilon,x)\Sigma_\mu(n\epsilon,x)\}_{TA} \tag{29b}$$

$$P_\mu(\tfrac{1}{2}\epsilon,x) = P_\mu(0,x) + \tfrac{\epsilon}{2}F_\mu(0,x) \tag{29c}$$

$$U_\mu((n+1)\epsilon,x) = \exp\left(i\epsilon P_\mu((n+\tfrac{1}{2})\epsilon,x)\right)U_\mu(n\epsilon,x) \tag{29d}$$

$$P_\mu((n+\tfrac{1}{2})\epsilon,x) = P_\mu((n-\tfrac{1}{2})\epsilon,x) + \epsilon F_\mu(n\epsilon,x) \tag{29e}$$

Note that, the sign of F is '+' here which is different from Ref. [2], because in Ref. [2], $F = \partial_{\mu,n}S = -\dot{P}$. Here we define $F = \dot{P} = -\partial_{\mu,n}S$.

or simply written as

$$P_\epsilon \circ U_\epsilon \circ P_{\frac{1}{2}\epsilon}\,(P_0,U_0) \tag{30}$$

The pseudo code can be written as

```
1
2  FieldGauge field = gaugeField.copy();
3
4  //sum _i i r_i T_i, where r_i are random numbers generated by Gaussian distribution
5  FieldGauge momentumField = FieldGauge::RandomGenerator();
6
7  //First half update
8  FieldGauge forceField = FieldGauge::Zero();
9  for (int i = 0; i < m_lstActions.Num(); ++i)
10 {
11     forceField += m_lstActions[i]->CalculateForceOnGauge(field);
12 }
13 //momentumField = momentumField + 0.5f * epsilon * forceField
14 momentumField.Axpy(fStep * 0.5f, forceField);
15
16 for (int i = 1; i < steps + 1; ++i)
17 {
18     field = FieldGauge::Exp(fStep * momentumField) * field;
19     forceField = FieldGauge::Zero();
20     for (int j = 0; j < m_lstActions.Num(); ++j)
21     {
22         forceField += m_lstActions[j]->CalculateForceOnGauge(field);
23     }
24     momentumField.Axpy((j < steps) ? fStep : (fStep * 0.5f), forceField);
25 }
```

2.1.6 A summary of HMC with pseudofermions

Now, every part is ready. We summary the HMC following the Sec.8.2.3 in Ref. [2]. The HMC with fermions can be divided into 6 steps.

1. Generate a complex Bosonic field with $\chi \sim \exp(-\chi^\dagger \chi)$, and $\phi = \hat{D}\chi$.
2. Generate a momentum field P by $\exp(-tr(P^2))$.
3. Calculate $E = tr(P^2) + S_G(U) + S_{pf}(U, \phi)$.
4. Use U_0 to calculate F, evaluate P and U using integrator. Here, ϕ is treated as a constant field.

5. Finally, use P', U' to calculate Calculate $E' = tr(P'^2) + S_G(U') + S_{pf}(U', \phi)$. Use a Metropolis to accept or reject the result (configurations) **Note, by Refs. [2] and [6] 'reject' means add a duplicated old configuration.**.

6. Iterate from 1 to 5, until the number of configurations generated is sufficient.

- More on Metropolis step:

If the hybrid Monte Carlo can be implemented exactly, then, when equilibrium is reached, H should be unchanged, so, in some implementation, the Metropolis step can be ignored to archive a better accept rate.

2.2 Optimization of HMC

2.2.1 Omelyan integrator

The Omelyan integrator can be simply written as (c.f. Eq. (2.80) of Ref. [1])

$$P_{\lambda\epsilon} \circ U_{\frac{1}{2}\epsilon} \circ P_{(1-2\lambda)\epsilon} \circ U_{\frac{1}{2}\epsilon} \circ P_{\lambda\epsilon} \left(P_0, U_0\right) \tag{31}$$

with

$$\lambda = \frac{1}{2} - \frac{\left(2\sqrt{326} + 36\right)^{\frac{1}{3}}}{12} + \frac{1}{6\left(2\sqrt{326} + 36\right)^{\frac{1}{3}}} \approx 0.19318332750378364 \tag{32}$$

In practical, the λ is a tunable parameter, and usually, $2\lambda = 0.3 \sim 0.5$ [6]. The `Omelyan2Lambda` parameter of `Updator` is a input parameter to set 2λ, which if left blank is set to be 0.38636665500756728 by default.

Usually, for each sub-step, it is 2 times slower then leap-frog, and for one trajectory, it is 1.5 time faster [6], implying the number of sub-step needed is about 1/3 of leap-frog.

2.2.2 Multi-Step Updator

2.3 Sparse linear algebra solver

3 Measurement

3.1 Plaquette Energy

For $SU(N)$, for square lattice, the gauge action can be written as

$$\begin{aligned}
S_G &= \beta\frac{1}{N}\sum_n\sum_{\mu>\nu}\left(N-\mathrm{tr}\left[U_\mu(n)U_\nu(n+a\mu)U_\mu^{-1}(n+a\nu)U_\nu^{-1}(n)\right]\right)\\
S_G &= \frac{1}{4}\beta\frac{1}{N}\sum_n\left((2(D-1))N-\mathrm{tr}\left[U_\mu(n)\Sigma_\mu(n)\right]\right),\\
\Sigma_\mu(n) &= \sum_{\mu\neq\nu}\left(U_\nu(n)U_\mu(n+a\nu)U_\nu^{-1}(n+a\mu)+U_\nu^{-1}(n-a\nu)U_\mu(n-a\nu)U_\nu(n-a\nu+a\mu)\right)
\end{aligned} \tag{33}$$

The plaquette energy is defined as

$$\begin{aligned}
\langle S\rangle &= \frac{1}{N\Lambda}\sum_n\sum_{\mu>\nu}\left(\mathrm{tr}\left[U_\mu(n)U_\nu(n+a\mu)U_\mu^{-1}(n+a\nu)U_\nu^{-1}(n)\right]\right).\\
&= \frac{1}{N\Lambda}\sum_{n,\mu}\mathrm{tr}\left[U_\mu(n)\Sigma_\mu(n)\right].
\end{aligned} \tag{34}$$

which is the average (average according to configurations) energy of plaquettes per plaquette (average according to plaquettes).

This is also $\langle W^{1\times1}\rangle$.

3.2 Meson Correlator

3.2.1 Meson Wave Function

We need at first construct an observable which is a bound state of two fermions and **has the same quantum number** as mesons. In short, we want to know

$$O(x)=\bar\psi(x)\Gamma\psi(x) \tag{35}$$

where Γ is a (product of) gamma matrix.

3.2.2 Meson Correlator

The correlator is defined as

$$C(x,y)=\langle\bar O(x)O(y)\rangle \tag{36}$$

where

$$\begin{aligned}\langle W\rangle &= \frac{1}{Z}\int \mathcal{D}[U,\bar{\psi},\psi]W\exp\left(-S\right)\\ Z &= \int \mathcal{D}[U,\bar{\psi},\psi]\exp\left(-S\right),\quad S=S_G+S_{pf}\end{aligned} \tag{37}$$

- iso-triplet

  Denote the variables as C_T and O_T.

  We need to calculate (green variables are constant)

$$\begin{aligned}C_T(n,m) &= \langle\bar{\psi}^{f_1}(n)\Gamma\psi^{f_2}(n)\bar{\psi}^{f_2}(m)\Gamma\psi^{f_1}(m)\rangle\\ &= \sum_{a,b,c_i}\Gamma_{a_1,b_1}\Gamma_{a_2,b_2}\langle\bar{\psi}^{f_1}_{a_1,c_1}(n)\psi^{f_2}_{b_1,c_1}(n)\bar{\psi}^{f_2}_{a_2,c_2}(m)\psi^{f_1}_{b_2,c_2}(m)\rangle\end{aligned} \tag{38}$$

  Note that, they are all Grassman numbers (exchange three times will introduce a minus sign), and they can be averaged according to different fields, so

$$C_T(n,m) = -\sum_{a,b,c_i}\Gamma_{a_1,b_1}\Gamma_{a_2,b_2}\langle\psi^{f_2}_{b_1,c_1}(n)\bar{\psi}^{f_2}_{a_2,c_2}(m)\rangle_{f_1}\langle\psi^{f_1}_{b_2,c_2}(m)\bar{\psi}^{f_1}_{a_1,c_1}(n)\rangle_{f_2} \tag{39}$$

  Using the Wick theorem for Grassman numbers (f is flavour index, c is color index, a,b are spinor index).

$$\begin{aligned}\langle\ldots\rangle &= \frac{1}{Z_f}\int\mathcal{D}\left[\psi\right]\ldots\exp\left(-\sum_{l,m}^{N}\bar{\psi}_l M_{lm}\psi_m\right).\\ \langle\psi_{i_1}\ldots\psi_{i_n}\bar{\psi}_{j_1}\ldots\bar{\psi}_{j_n}\rangle &= \sum_P \mathrm{sign}(P)\prod_n^N\left(M^{-1}\right)_{i_n,j_{P_n}}.\\ \langle\psi^f(n)_{a,c_1}\bar{\psi}^f_{b,c_2}(m)\rangle &= -D^{-1}_{f,a,b,c_1,c_2}(n,m).\end{aligned} \tag{40}$$

Then we can multiply gamma matrix back

$$C_T(n,m) = -\mathrm{tr}_{c,s}\left[\Gamma D^{-1}_{f_1}(n,m)\Gamma D^{-1}_{f_2}(m,n)\right] \tag{41}$$

  The trace is for both color and spinor space.

- iso-singlet

  Denote the variables as C_S and O_S.

3.2.3 Monte Carlo in Monte Carlo

Trace estimator.

4 Programming

4.1 cuda

4.1.1 blocks and threads

4.1.2 device member function

According to https://stackoverflow.com/questions/53781421/cuda-the-member-field-with-device-ptr-and-device-member-function-to-visit-it-i

To call device member function, the content of the class should be on device.

- First, new a instance of the class.
- Then, create a device memory using cudaMalloc.
- Copy the content to the device memory

In other words, it will work as

```
__global__ void _kInitialArray(int* thearray)
{
    int iX = threadIdx.x + blockDim.x * blockIdx.x;
    int iY = threadIdx.y + blockDim.y * blockIdx.y;
    int iZ = threadIdx.z + blockDim.z * blockIdx.z;
    thearray[iX * 16 + iY * 4 + iZ] = iX * 16 + iY * 4 + iZ;
}

extern "C" {
    void _cInitialArray(int* thearray)
    {
        dim3 block(1, 1, 1);
        dim3 th(4, 4, 4);

        _kInitialArray << <block, th >> > (thearray);
        checkCudaErrors(cudaGetLastError());
    }
}

class B
{
public:
    B()
    {
```

```
25          checkCudaErrors(cudaMalloc((void**)&m_pDevicePtr, sizeof(int) * 64));
26          _cInitialArray(m_pDevicePtr);
27      }
28      ~B()
29      {
30          cudaFree(m_pDevicePtr);
31      }
32      __device__ int GetNumber(int index)
33      {
34          m_pDevicePtr[index] = m_pDevicePtr[index] + 1;
35          return m_pDevicePtr[index];
36      }
37      int* m_pDevicePtr;
38  };
39
40  __global__ void _kAddArray(int* thearray1, B* pB)
41  {
42      int iX = threadIdx.x + blockDim.x * blockIdx.x;
43      int iY = threadIdx.y + blockDim.y * blockIdx.y;
44      int iZ = threadIdx.z + blockDim.z * blockIdx.z;
45      thearray1[iX * 16 + iY * 4 + iZ] = thearray1[iX * 16 + iY * 4 + iZ] + pB->GetNumber(iX * 16 +
            iY * 4 + iZ);
46  }
47
48  extern "C" {
49      void _cAddArray(int* thearray1, B* pB)
50      {
51          dim3 block(1, 1, 1);
52          dim3 th(4, 4, 4);
53          _kAddArray << <block, th >> > (thearray1, pB);
54          checkCudaErrors(cudaGetLastError());
55      }
56  }
57
58  class A
59  {
60  public:
61      A()
62      {
63          checkCudaErrors(cudaMalloc((void**)&m_pDevicePtr, sizeof(int) * 64));
64          _cInitialArray(m_pDevicePtr);
65      }
66      ~A()
67      {
68          checkCudaErrors(cudaFree(m_pDevicePtr));
69      }
70      void Add(B* toAdd/*this should be a device ptr(new on device function or created by cudaMalloc)
```

```
            */)
71      {
72          _cAddArray(m_pDevicePtr, toAdd);
73      }
74      int* m_pDevicePtr;
75  };
76
77
78
79  int main(int argc, char * argv[])
80  {
81      B* pB = new B();
82      A* pA = new A();
83      B* pDeviceB;
84      checkCudaErrors(cudaMalloc((void**)&pDeviceB, sizeof(B)));
85      checkCudaErrors(cudaMemcpy(pDeviceB, pB, sizeof(B), cudaMemcpyHostToDevice));
86      pA->Add(pDeviceB);
87      int* res = (int*)malloc(sizeof(int) * 64);
88      checkCudaErrors(cudaMemcpy(res, pA->m_pDevicePtr, sizeof(int) * 64, cudaMemcpyDeviceToHost));
89      printf("----------␣A=");
90      for (int i = 0; i < 8; ++i)
91      {
92          printf("\n");
93          for (int j = 0; j < 8; ++j)
94              printf("res␣%d=%d␣␣", i * 8 + j, res[i * 8 + j]);
95      }
96      printf("\n");
97      //NOTE: We are getting data from pB, not pDeviceB, this is OK, ONLY because m_pDevicePtr is a
            pointer
98      checkCudaErrors(cudaMemcpy(res, pB->m_pDevicePtr, sizeof(int) * 64, cudaMemcpyDeviceToHost));
99      printf("----------␣B=");
100     for (int i = 0; i < 8; ++i)
101     {
102         printf("\n");
103         for (int j = 0; j < 8; ++j)
104             printf("res␣%d=%d␣␣", i * 8 + j, res[i * 8 + j]);
105     }
106     printf("\n");
107     delete pA;
108     delete pB;
109     return 0;
110 }
```

Note: this is a copy of the original instance! It is ONLY OK to change the content of $pDevicePtr->m\_pOtherPtr$, NOT $pDevicePtr->somevalue$

4.1.3 device virtual member function

According to https://stackoverflow.com/questions/26812913/how-to-implement-device-side-cuda-virtual-functions

To call a device virtual member function, unlike Sec. 4.1.2, the pointer to the virtual function table should also be on device,

- First, cudaMalloc a sizeof(void*), for the device pointer.
- Then, use a kernel function to new the instance on device, and assign it to the device pointer created by cudaMalloc.
- One can copy the pointer, by using cudaMemcpy(void**, void**, sizeof(void*), device-todevice).
- When copy it to elsewhere, one need to copy it back to host, then copy it again to device. The example shows how to copy it to constant.

in other words, it will work as

```
1
2  class CA
3  {
4  public:
5      __device__ CA() { ; }
6      __device__ ~CA() { ; }
7      __device__ virtual void CallMe() { printf("This␣is␣A\n"); }
8  };
9
10 class CB : public CA
11 {
12 public:
13     __device__ CB() : CA() { ; }
14     __device__ ~CB() { ; }
15     __device__ virtual void CallMe() { printf("This␣is␣B\n"); }
16 };
17
18 __global__ void _kernelCreateInstance(CA** pptr)
19 {
20     (*pptr) = new CB();
21 }
22
23 __global__ void _kernelDeleteInstance(CA** pptr)
24 {
```

```
25      delete (*pptr);
26  }
27
28  extern "C" {
29      void _kCreateInstance(CA** pptr)
30      {
31          _kernelCreateInstance << <1, 1 >> >(pptr);
32      }
33
34      void _kDeleteInstance(CA** pptr)
35      {
36          _kernelDeleteInstance << <1, 1 >> >(pptr);
37      }
38  }
39
40  __constant__ CA* m_pA;
41
42  __global__ void _kernelCallConstantFunction()
43  {
44      m_pA->CallMe();
45  }
46
47
48  extern "C" {
49      void _cKernelCallConstantFunction()
50      {
51          _kernelCallConstantFunction << <1, 1 >> > ();
52      }
53  }
54
55  int main()
56  {
57      CA** pptr;
58      cudaMalloc((void**)&pptr, sizeof(CA*));
59      _kCreateInstance(pptr);
60
61      //I can NOT use a kernel to set m_pA = (*pptr), because it is constant.
62      //I can NOT use cudaMemcpyToSymbol(m_pA, (*pptr)), because * operator on host is incorrect when
            pptr is a device ptr.
63      //I can NOT use cudaMemcpyToSymbol(m_pA, (*pptr)) in kernel, because cudaMemcpyToSymbol is a
            __host__ function
64      //I have to at first copy it back to host, then copy it back back again to constant
65      CA* pptrHost[1];
66      cudaMemcpy(pptrHost, pptr, sizeof(CA**), cudaMemcpyDeviceToHost);
67      cudaMemcpyToSymbol(m_pA, pptrHost, sizeof(CA*));
68      _cKernelCallConstantFunction();
69
```

```
70      _kDeleteInstance(pptr);
71      cudaFree(pptr);
72      return 0;
73  }
```

5 Testing

5.1 random number

索引

BiCGStab, 12

correlator, 17

equilibrium, 16

fat index, 4
force, 7

hmc, 6

Integrator, 8

Langevin equation, 7
leap frog, 14
link index, 4

meson, 17
molecular dynamics, 7

Omelyan, 15

plaquette energy, 17
pseudofermions, 6

site index, 4
solver, 11, 16
staple, 8

参考文献

[1] Michael Günther Francesco Knechtli and Michael Peardon. Lattice Quantum Chromodynamics Practical Essentials. 2017.

[2] C. Gattringer and C.B. Lang. Quantum Chromodynamics on the Lattice. 2010.

[3] Rajan Gupta. Introduction to Lattice QCD. 1998, arXiv:hep-lat/9807028.

[4] Alexander Altland and Ben Simons. Condensed Matter Field Theory 2nd edition. 2010.

[5] D. H. Weingarten and D. N. Petcher. Monte Carlo integration for lattice gauge theories with fermions. *Phys. Lett. B*, 99(4):333 – 338, 1981.

[6] Martin Lüscher. Computational Strategies in Lattice QCD. 2009, arXiv:1002.4232.

[7] S. Ueda et. al. Development of an object oriented lattice QCD code "Bridge++" on accelerators. *Journal of Physics: Conference Series*, 523:012046, 2014.