Cuda Lattice Gauge Document

目录

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 $\hat{D}_f = D + m_f$.

It can be evaluated as Eq. (1.86) of Ref. [1] (or Eq. (4.19) of Ref. [2])

$$Z = \prod_{f=1}^{N_f} \det\left(\hat{D}_f\right) \int \mathcal{D}[U] \exp\left(-S_G[U]\right) \tag{5}$$

On the other hand, with the help of Gaussian integral of complex vectors Eq. (3.17) of Ref. [2]

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

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

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

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 $\det(\hat{D}^\dagger)=\det(\gamma_5)\det(\hat{D})\det(\gamma_5)=\det(\hat{D})$.**), one can show Eq. (8.9) of Ref. [4]

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

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.

(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.

- 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 $SU(N)$,

$$\begin{aligned}&F_\mu(x)=\dot{P}_\mu(x)=-\frac{\beta}{2N}\{U_\mu(x)\Sigma_\mu(x)\}_{TA}\\&\{W\}_{TA}=\frac{W-W^\dagger}{2}-\mathrm{tr}\left(\frac{W-W^\dagger}{2N}\right)\mathbf{I}\end{aligned} \tag{13}$$

where $\mathbf{I}$ is identity matrix, Σ is the 'Staple'.

- Integrator

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

$$U(\tau + d\tau) \approx \dot{U}d\tau + U(\tau), \quad P(\tau + d\tau) \approx \dot{P}d\tau + P(\tau) \tag{14}$$

A more accurate calculation is done by integrator, for example, the leap frog integrator, the M step leap frog integral is described in Ref. [4],

$$\epsilon = \tfrac{\tau}{M} \tag{15a}$$

$$U_\mu(x,(n+1)\epsilon) = U_\mu(x,n\epsilon) + \epsilon P_\mu(x,n\epsilon) + \tfrac{1}{2}F_\mu(x,n\epsilon)\epsilon^2 \tag{15b}$$

$$P_\mu(x,(n+1)\epsilon) = P_\mu(x,n\epsilon) + \tfrac{1}{2}\left(F_\mu(x,(n+1)\epsilon) + F_\mu(x,n\epsilon)\right)\epsilon \tag{15c}$$

So, knowing $U(n\epsilon)$ we can calculate $F(n\epsilon)$ using Eq. (13). Knowing $U(n\epsilon), P(n\epsilon), F(n\epsilon)$, we can calculate $U((n+1)\epsilon)$ using Eq. (15).b. Then we are able to calculate $F((n+1)\epsilon)$ again using Eq. (13). Then we can calculate $P((n+1)\epsilon)$ using Eq. (15).c.

- About the randomized P

The randomized P is chosen according to normal distribution $\exp\left(-<P,P>/2\right)$. Using $P=\sum \omega_i T^i$, $tr((T^i)\cdot(T^j)) = 2\delta_{ij}$. It is usually written as distribution $\exp\left(-tr(P^2)\right)$. One can randomize ω_i using $\exp\left(-\omega_i^*\omega_i\right)$. Then using $P=\sum \omega_i T^i$.

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.

(1) Evaluate ϕ use force of U and ϕ on ϕ.

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

Let χ be random complex numbers according to $\exp(-\chi^\dagger\chi)$. Let $(D)^{-1}\phi = \chi$, ϕ is the random number satisfying distribution we want. And $\phi = D\chi$.

Using the Wilson Fermion action (TODO: Add ref here after the section of Wilson Fermion is added.).

$$\begin{aligned}\hat{D} &= D + 1\\ D &= -\kappa\sum_\mu\left((1-\gamma_\mu)U_\mu(x)\delta_{x,x+\mu} + (1+\gamma_\mu)U_\mu^{-1}(x-\mu)\delta_{x,x-\mu}\right)\end{aligned} \tag{16}$$

with $\kappa = 1/(2am_f + 8)$.

The force of ϕ on U is obtained as $\partial_{U_\mu} S_{pf}$. The result for Wilson Fermion action is shown in Eqs. (8.39), (8.44) and (8.45) of Ref. [4] as

$$\begin{aligned}
&F = F_G + \sum_i T^i \nabla^i \left(\phi^\dagger \left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right).\\
&\nabla^i \left(\phi^\dagger \left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right) = -\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right)^\dagger \left(\frac{\partial D}{\partial \omega^i_\mu}\hat{D}^\dagger + \hat{D}\frac{\partial D^\dagger}{\partial \omega^i_\mu}\right)\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right).\\
&\frac{\partial \hat{D}}{\partial \omega^i_\mu} = \frac{\partial D}{\partial \omega^i_\mu} = -i\kappa\left\{(1-\gamma_\mu)T^i U_\mu(x)\delta_{x,x+\mu} - (1+\gamma_\mu)U^{-1}_\mu(x-\mu)T^i\delta_{x,x-\mu}\right\}\\
&\hat{D}^\dagger = \gamma_5\hat{D}\gamma_5,\;\; \frac{\partial D^\dagger}{\partial \omega^i_\mu} = \gamma_5\frac{\partial D}{\partial \omega^i_\mu}\gamma_5
\end{aligned} \tag{17}$$

where F_G is force from U introduced in Sec. 2.1.2, T^i are $SU(3)$ generators. And

$$U_\mu = \exp(i\sum_i \omega^i_\mu T^i),\;\; \frac{\partial U_\mu}{\partial \omega^i_\mu} = iT^i U_\mu,\;\; \frac{\partial U^\dagger_\mu}{\partial \omega^i_\mu} = -iU^\dagger_\mu T^i,\;\; \left(T^i\right)^\dagger = T^i. \tag{18}$$

is used.

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

$$\begin{aligned}
&\phi_1 = \left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right),\;\; \phi_2 = \hat{D}^\dagger\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right),\;\; \phi_1^\dagger D = \phi_2^\dagger\\
&\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right)^\dagger \left(\frac{\partial D}{\partial \omega^i_\mu}\hat{D}^\dagger + \hat{D}\frac{\partial D^\dagger}{\partial \omega^i_\mu}\right)\left(\left(\hat{D}\hat{D}^\dagger\right)^{-1}\phi\right)\\
&= \phi_1^\dagger\frac{\partial D}{\partial \omega^i_\mu}\phi_2 + \phi_2^\dagger\frac{\partial D^\dagger}{\partial \omega^i_\mu}\phi_1 = 2\mathrm{Re}\left(\phi_1^\dagger\frac{\partial D}{\partial \omega^i_\mu}\phi_2\right)
\end{aligned} \tag{19}$$

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, it seems we do not need ϕ but just χ, because $\phi_1 = (\hat{D}^\dagger)^{-1}\hat{D}^{-1}\phi = (\hat{D}^\dagger)^{-1}\chi$ and $\phi_2 = \chi$. However, this is **incorrect** because 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. 2.2. 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

$$Q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \cdot A \cdot \mathbf{x} - \mathbf{x}^T\mathbf{b}. \tag{20}$$

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

$$\frac{\partial}{\partial \mathbf{x}} Q(\mathbf{x}) = 0 = A \cdot \mathbf{x} - \mathbf{b}. \tag{21}$$

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. [4]. It can be described as

```

CField* pX, pR, pRH, pV, pP, pS, pT;

//use it to estimate relative error, a->Dot(b) means a_dagger . b
Real fBLength = pFieldB->Dot(pFieldB);

//Using b as the guess, (Assuming M is near identity?)
pFieldB->CopyTo(pX);

//r_0 = b - A x_0
pFieldB->CopyTo(pR);
pR->ApplyOperator(uiM, pGaugeFeild); //A x_0, Note D operator need gauge field
pR->ScalarMultply(-1); //-A x_0
pR->AxpyPlus(pX); //b - A x_0
pR->CopyTo(pRh);

Real rho = 0;
Real last_rho = 0;
Real alpha = 0;
Real beta = 0;
Real omega = 0;

for (UINT i = 0; i < m_uiReTry; ++i)
{
    for (UINT j = 0; j < m_uiStepCount * m_uiDevationCheck; ++j)
    {
        //One step
        rho = _cuCabsf(m_pRh->Dot(m_pR));//rho = rh dot r(i-1), if rho = 0, failed (assume will not)
```

```

        if (0 == j) //if is the first iteration, p=r(i-1)
        {
            pR->CopyTo(pP);
        }
        else //if not the first iteration,
        {
            //beta = last_alpha * rho /(last_omega * last_rho)
            beta = alpha * rho / (omega * last_rho);
            //p(i) = r(i-1)+beta( p(i-1) - last_omega v(i-1) )
            pV->ScalarMultply(omega);
            pP->AxpyMinus(pV); //p = p - v
            pP->ScalarMultply(beta);
            pP->AxpyPlus(pR);
        }

        //v(i) = A p(i)
        pP->CopyTo(pV);
        pV->ApplyOperator(uiM, pGaugeFeild);

        alpha = rho / (_cuCabsf(pRh->Dot(pV)));//alpha = rho / (rh dot v(i))

        //s=r(i-1) - alpha v(i)
        pR->CopyTo(pS);
        pS->Axpy(-alpha, pV);

        //t=As
        pS->CopyTo(pT);
        pT->ApplyOperator(uiM, pGaugeFeild);

        omega = _cuCabsf(pT->Dot(pS)) / _cuCabsf(pT->Dot(pT));//omega = ts / tt

        //r(i)=s-omega t
        pS->CopyTo(pR);
        if (0 == (j - 1) \% m_uiDevationCheck)
        {
            //Normal of S is small, then stop
            Real fDeviation = _cuCabsf(pS->Dot(pS)) / fBLength;
            appParanoiac(_T("CSLASolverBiCGStab::Solve␣deviation:␣restart:\%d,␣iteration:\%d,␣
                 deviation:\%f\n"), i, j, fDeviation);
            if (fDeviation < m_fAccuracy)
            {
                pX->Axpy(alpha, pP);
                pX->CopyTo(pFieldX);
                return;
            }
        }
```

```

            pR->Axpy(-omega, pT);

            //x(i)=x(i-1) + alpha p + omega s
            pX->Axpy(alpha, pP);
            pX->Axpy(omega, pS);

            last_rho = rho;//last_rho = rho
        }

        //we are here, means we do not converge.
        //we need to restart with a new guess, we use last X
        pX->CopyTo(pR);

        pR->ApplyOperator(uiM, pGaugeFeild); //A x_0
        pR->ScalarMultply(-1); //-A x_0
        pR->AxpyPlus(pX); //b - A x_0
        pR->CopyTo(pRh);
    }


    //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.

$$U_\mu(0,x) = gauge(x), \quad P_\mu(0,x) = i\sum_a r_a(\mu,x)T_a \tag{22a}$$

$$F_\mu(n\epsilon,x) = -\frac{\beta}{2N}\{U_\mu(n\epsilon,x)\Sigma_\mu(n\epsilon,x)\}_{TA} \tag{22b}$$

$$P_\mu(\tfrac{1}{2}\epsilon,x) = P_\mu(0,x) + \frac{\epsilon}{2}F_\mu(0,x) \tag{22c}$$

$$U_\mu((n+1)\epsilon,x) = \exp\left(\epsilon P_\mu((n+\tfrac{1}{2})\epsilon,x)\right)U_\mu(n\epsilon,x) \tag{22d}$$

$$P_\mu((n+\tfrac{1}{2})\epsilon,x) = P_\mu((n-\tfrac{1}{2})\epsilon,x) + \epsilon F_\mu(n\epsilon,x) \tag{22e}$$

or simply written as

$$P_\epsilon \circ U_\epsilon \circ P_{\frac{1}{2}\epsilon}(P_0,U_0) \tag{23}$$

The pseudo code can be written as

```

FieldGauge field = gaugeField.copy();

//sum _i i r_i T_i, where r_i are random numbers generated by Gaussian distribution
FieldGauge momentumField = FieldGauge::RandomGenerator();

//First half update
FieldGauge forceField = FieldGauge::Zero();
for (int i = 0; i < m_lstActions.Num(); ++i)
{
    forceField += m_lstActions[i]->CalculateForceOnGauge(field);
}
//momentumField = momentumField + 0.5f * epsilon * forceField
momentumField.Axpy(fStep * 0.5f, forceField);

for (int i = 1; i < steps + 1; ++i)
{
    field = FieldGauge::Exp(fStep * momentumField) * field;
    forceField = FieldGauge::Zero();
    for (int j = 0; j < m_lstActions.Num(); ++j)
    {
        forceField += m_lstActions[j]->CalculateForceOnGauge(field);
    }
    momentumField.Axpy((j < steps) ? fStep : (fStep * 0.5f), forceField);
}
```

2.1.6 Omelyan integrator

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

$$P_{\lambda\epsilon} \circ U_{\frac{1}{2}\epsilon} \circ P_{(1-2\lambda)\epsilon} \circ U_{\frac{1}{2}\epsilon} \circ P_{\lambda\epsilon}\left(P_0, U_0\right) \tag{24}$$

with

$$\lambda = \frac{1}{2} - \frac{\left(2\sqrt{326}+36\right)^{\frac{1}{3}}}{12} + \frac{1}{6\left(2\sqrt{326}+36\right)^{\frac{1}{3}}} \approx 0.19318332750378364 \tag{25}$$

2.1.7 A summary of HMC with pseudofermions

Now, every part is ready. We summary the HMC following the Sec.8.2.3 in Ref. [4]. 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).
6. Iterate from 1 to 5, until the number of configurations generated is sufficient.

2.2 Sparse linear algebra solver

3 Programming

3.1 cuda

3.1.1 blocks and threads

3.1.2 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()
    {
```

```
        checkCudaErrors(cudaMalloc((void**)&m_pDevicePtr, sizeof(int) * 64));
        _cInitialArray(m_pDevicePtr);
    }
    ~B()
    {
        cudaFree(m_pDevicePtr);
    }
    __device__ int GetNumber(int index)
    {
        m_pDevicePtr[index] = m_pDevicePtr[index] + 1;
        return m_pDevicePtr[index];
    }
    int* m_pDevicePtr;
};

__global__ void _kAddArray(int* thearray1, B* pB)
{
    int iX = threadIdx.x + blockDim.x * blockIdx.x;
    int iY = threadIdx.y + blockDim.y * blockIdx.y;
    int iZ = threadIdx.z + blockDim.z * blockIdx.z;
    thearray1[iX * 16 + iY * 4 + iZ] = thearray1[iX * 16 + iY * 4 + iZ] + pB->GetNumber(iX * 16 +
         iY * 4 + iZ);
}

extern "C" {
    void _cAddArray(int* thearray1, B* pB)
    {
        dim3 block(1, 1, 1);
        dim3 th(4, 4, 4);
        _kAddArray << <block, th >> > (thearray1, pB);
        checkCudaErrors(cudaGetLastError());
    }
}

class A
{
public:
    A()
    {
        checkCudaErrors(cudaMalloc((void**)&m_pDevicePtr, sizeof(int) * 64));
        _cInitialArray(m_pDevicePtr);
    }
    ~A()
    {
        checkCudaErrors(cudaFree(m_pDevicePtr));
    }
    void Add(B* toAdd/*this should be a device ptr(new on device function or created by cudaMalloc)
```

```
            */)
    {
        _cAddArray(m_pDevicePtr, toAdd);
    }
    int* m_pDevicePtr;
};



int main(int argc, char * argv[])
{
    B* pB = new B();
    A* pA = new A();
    B* pDeviceB;
    checkCudaErrors(cudaMalloc((void**)&pDeviceB, sizeof(B)));
    checkCudaErrors(cudaMemcpy(pDeviceB, pB, sizeof(B), cudaMemcpyHostToDevice));
    pA->Add(pDeviceB);
    int* res = (int*)malloc(sizeof(int) * 64);
    checkCudaErrors(cudaMemcpy(res, pA->m_pDevicePtr, sizeof(int) * 64, cudaMemcpyDeviceToHost));
    printf("----------- A=");
    for (int i = 0; i < 8; ++i)
    {
        printf("\n");
        for (int j = 0; j < 8; ++j)
            printf("res %d=%d  ", i * 8 + j, res[i * 8 + j]);
    }
    printf("\n");
    //NOTE: We are getting data from pB, not pDeviceB, this is OK, ONLY because m_pDevicePtr is a pointer
    checkCudaErrors(cudaMemcpy(res, pB->m_pDevicePtr, sizeof(int) * 64, cudaMemcpyDeviceToHost));
    printf("----------- B=");
    for (int i = 0; i < 8; ++i)
    {
        printf("\n");
        for (int j = 0; j < 8; ++j)
            printf("res %d=%d  ", i * 8 + j, res[i * 8 + j]);
    }
    printf("\n");
    delete pA;
    delete pB;
    return 0;
}
```

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

3.1.3 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. 3.1.2, 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*), devicetodevice).
- 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

```

class CA
{
public:
    __device__ CA() { ; }
    __device__ ~CA() { ; }
    __device__ virtual void CallMe() { printf("This is A\n"); }
};

class CB : public CA
{
public:
    __device__ CB() : CA() { ; }
    __device__ ~CB() { ; }
    __device__ virtual void CallMe() { printf("This is B\n"); }
};

__global__ void _kernelCreateInstance(CA** pptr)
{
    (*pptr) = new CB();
}

__global__ void _kernelDeleteInstance(CA** pptr)
{
```

```
    delete (*pptr);
}

extern "C" {
    void _kCreateInstance(CA** pptr)
    {
        _kernelCreateInstance << <1, 1 >> >(pptr);
    }

    void _kDeleteInstance(CA** pptr)
    {
        _kernelDeleteInstance << <1, 1 >> >(pptr);
    }
}

__constant__ CA* m_pA;

__global__ void _kernelCallConstantFunction()
{
    m_pA->CallMe();
}


extern "C" {
    void _cKernelCallConstantFunction()
    {
        _kernelCallConstantFunction << <1, 1 >> > ();
    }
}

int main()
{
    CA** pptr;
    cudaMalloc((void**)&pptr, sizeof(CA*));
    _kCreateInstance(pptr);

    //I can NOT use a kernel to set m_pA = (*pptr), because it is constant.
    //I can NOT use cudaMemcpyToSymbol(m_pA, (*pptr)), because * operator on host is incorrect when pptr is a device ptr.
    //I can NOT use cudaMemcpyToSymbol(m_pA, (*pptr)) in kernel, because cudaMemcpyToSymbol is a __host__ function
    //I have to at first copy it back to host, then copy it back back again to constant
    CA* pptrHost[1];
    cudaMemcpy(pptrHost, pptr, sizeof(CA**), cudaMemcpyDeviceToHost);
    cudaMemcpyToSymbol(m_pA, pptrHost, sizeof(CA*));
    _cKernelCallConstantFunction();

```

```
    _kDeleteInstance(pptr);
    cudaFree(pptr);
    return 0;
}
```

4 Testing

4.1 random number

索引

参考文献

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

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

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

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