Cuda Lattice Gauge Document

目录

1 Data

1.1 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 $dirCount$,

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

1.2 CParemeters

2 Update scheme

2.1 HMC

HMC is abbreviation for hybrid Monte Carlo.

2.1.1 Basic idea

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 in CLG, $\dot{P}$ is called 'force'. See Eqs. (2.53), (2.56) and (2.57) of Ref. [1], for $SU(N)$,

$$\begin{aligned} F_\mu(x) = \dot{P}_\mu(x) &= -\frac{\beta}{2N}\{U_\mu(x)\Sigma_\mu(x)\}_{TA} \\ \{W\}_{TA} &= \frac{W - W^\dagger}{2} - \mathrm{tr}\left(\frac{W - W^\dagger}{2N}\right)\mathbf{I} \end{aligned} \tag{4}$$

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

- Integrator

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

$$U(\tau + d\tau) \approx \dot{U}d\tau + U(\tau), \quad P(\tau + d\tau) \approx \dot{P}d\tau + P(\tau) \tag{5}$$

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],

$$\epsilon = \tfrac{\tau}{M} \tag{6a}$$

$$U_\mu(x, (n+1)\epsilon) = U_\mu(x, n\epsilon) + \epsilon P_\mu(x, n\epsilon) + \tfrac{1}{2}F_\mu(x, n\epsilon)\epsilon^2 \tag{6b}$$

$$P_\mu(x, (n+1)\epsilon) = P_\mu(x, n\epsilon) + \tfrac{1}{2}\left(F_\mu(x, (n+1)\epsilon) + F_\mu(x, n\epsilon)\right)\epsilon \tag{6c}$$

So, knowing $U(n\epsilon)$ we can calculate $F(n\epsilon)$ using Eq. (4). Knowing $U(n\epsilon), P(n\epsilon), F(n\epsilon)$, we can calculate $U((n+1)\epsilon)$ using Eq. (6).b. Then we are able to calculate $F((n+1)\epsilon)$ again using Eq. (4). Then we can calculate $P((n+1)\epsilon)$ using Eq. (6).c.

2.1.2 Leap frog integrator

In Sec. 2.1.1, the basic idea is introduced. However, the implementation is slightly different.

$$U_\mu(0,x) = gauge(x), \quad P_\mu(0,x) = i\sum_a r_a(\mu,x)T_a \tag{7a}$$

$$F_\mu(n\epsilon,x) = -\tfrac{\beta}{2N}\{U_\mu(n\epsilon,x)\Sigma_\mu(n\epsilon,x)\}_{TA} \tag{7b}$$

$$P_\mu(\tfrac{1}{2}\epsilon,x) = P_\mu(0,x) + \tfrac{\epsilon}{2}F_\mu(0,x) \tag{7c}$$

$$U_\mu((n+1)\epsilon,x) = \exp\left(\epsilon P_\mu((n+\tfrac{1}{2})\epsilon,x)\right)U_\mu(n\epsilon,x) \tag{7d}$$

$$P_\mu((n+\tfrac{1}{2})\epsilon,x) = P_\mu((n-\tfrac{1}{2})\epsilon,x) + \epsilon F_\mu(n\epsilon,x) \tag{7e}$$

or simply written as

$$P_\epsilon \circ U_\epsilon \circ P_{\frac{1}{2}\epsilon}\,(P_0,U_0) \tag{8}$$

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  }
```

2.1.3 Omelyan integrator

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

$$P_{\lambda\epsilon} \circ U_{\frac{1}{2}\epsilon} \circ P_{(1-2\lambda)\epsilon} \circ U_{\frac{1}{2}\epsilon} \circ P_{\lambda\epsilon} \left(P_0, U_0\right) \tag{9}$$

with

$$\lambda = \frac{1}{2} - \frac{\left(2\sqrt{326}+36\right)^{\frac{1}{3}}}{12} + \frac{1}{6\left(2\sqrt{326}+36\right)^{\frac{1}{3}}} \approx 0.19318332750378364 \tag{10}$$

3 Programming

3.1 cuda

3.1.1 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()
    {
```

```
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 $pDevicePtr->somevalue$

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*), 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  }
```

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