24.evolution 零碎筆記

Chen Peng Chung

January 13, 2026

Contents

以下為其他程式碼的補充。補充:

```
#define  NX   32      //x方向物理空間網格數：32
#define  NY   128     //y方向物理空間網格數：128
#define  NZ   64      //z方向物理空間網格數：64

#define  jp   4       //GPU的使用數量：4

#define  NX6 (NX+7)
//(1)x方向物理空間節點數：NX(Link-wise)(2)x方向總節點數：NX+7(定義采用前3後4)
#define  NYD6 (NY/jp+7)//還不知道為什麼
#define  NY6 (NY+7)
//(1)y方向物理空間節點數：NY(Link-wise)(2)y方向總節點數：NY+7(定義采用前3後4)
#define  NZ6 (NZ+6)  //(?)我覺得他寫錯了，應該是NZ+7
```

Listing 1: variables.h10-19

在 Link-wise 節點佈局中，物理空間變的儲存點個數 = 物理空間之網格數

為什麼總節點數相較於物理間總節點數多 6 個單位?

因為本程式碼為用到 Lagrange 6 階空間精度內插公式:

$$f(x) = \sum_{i=0}^{6} f(x_i) \prod_{\substack{j=0 \\ j\neq i}}^{6} \frac{x - x_j}{x_i - x_j}$$

來取值執行 collision step 前的分佈函數，在三維空間中，上述之內插公式則需要 7*7*7 個鄰近空間點 (stencil) 的函數值來決定:

以下將列出三維空間 7 點 Lagrange 內插公式：

Let $(x_{\hat{i}}, y_{\hat{j}}, z_{\hat{k}})$ denote a reference grid point. Define the normalized offsets

$$\delta x = \frac{x - x_{\hat{i}}}{\Delta x}, \quad \delta y = \frac{y - y_{\hat{j}}}{\Delta y}, \quad \delta z = \frac{z - z_{\hat{k}}}{\Delta z},$$

where $\delta x, \delta y, \delta z \in R$.

The three-dimensional 7-point Lagrange interpolation in tensor-product form is written as

$$f\big(x_{\hat{i}} + \delta x \Delta x,\, y_{\hat{j}} + \delta y \Delta y,\, z_{\hat{k}} + \delta z \Delta z\big) = \sum_{i=-3}^{3} \sum_{j=-3}^{3} \sum_{k=-3}^{3} f_{\hat{i}+i,\,\hat{j}+j,\,\hat{k}+k}\, \ell_i(\delta x)\, \ell_j(\delta y)\, \ell_k(\delta z), \tag{零.1}$$

where the one-dimensional Lagrange basis functions are defined as

$$\ell_i(\delta x) = \prod_{\substack{m=-3 \\ m\neq i}}^{3} \frac{\delta x - m}{i - m}, \tag{零.2}$$

$$\ell_j(\delta y) = \prod_{\substack{n=-3 \\ n\neq j}}^{3} \frac{\delta y - n}{j - n}, \tag{零.3}$$

$$\ell_k(\delta z) = \prod_{\substack{q=-3 \\ q\neq k}}^{3} \frac{\delta z - q}{k - q}. \tag{零.4}$$

所以一邊之邊界點最多需要向外插值三個點的值來完成 Lagrange 七點內插。而對於完整的一維空間內插設置，則需要多設置六個節點，這六層向外使用的節點層稱為 buffer layers。

$$f\big(x_{\hat{i}} + \delta x\Delta x,\, y_{\hat{j}} + \delta y\Delta y,\, z_{\hat{k}} + \delta z\Delta z\big) = \sum_{i=-3}^{3}\sum_{j=-3}^{3}\sum_{k=-3}^{3} f_{\hat{i}+i,\,\hat{j}+j,\,\hat{k}+k}\,\ell_i(\delta x)\,\ell_j(\delta y)\,\ell_k(\delta z), \tag{零.5}$$

因此，在每一個方向上，都需要多新增 3 層 budder layers 來提供插值所需的鄰近點值。故總節點數 = 物理空間節點數 + 6 (3 層 buffer layers)。又因為週期性邊條件在 Stream-Wise 以及 Span-Wise 方向，需要複製壁面來向的撞遷移過程，所以在維持相同碰撞 pattern 的條件下，多設立一層作為實現週期性邊條件的技巧。

Lagrange 七點內插公式在 Latticeb Boltzmann Method 中的應用:

Interpolation-Based Lattice Boltzmann Method :

所以，插值，是插誰? 插【非均勻網格節點的遷移前空間點的碰撞前分佈函數】，如圖說明: 12/29 更新: 插值: 【(R(T(A 物理空間計算點) 的 (遷移前空間點)) 的 (碰撞後分佈函數))】所以整體碰撞演算法為: 插值 > 碰撞 > 遷移。先透過插值的方式得到不存在的分佈函數，再碰撞更新半步長的時間步，最後遷移得到往格中心點實際存在的碰撞遷移後分佈函數。

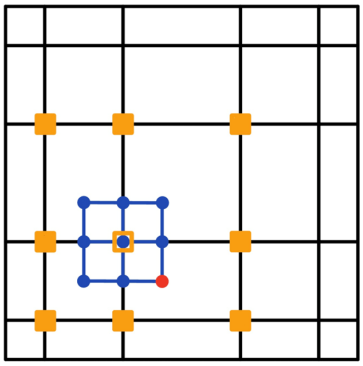

Figure 零.1: Schematic illustration of the interpolation-based Lattice Boltzmann method (LBM).

Source: Y.-H. Chiu, *Simulations of turbulent Poiseuille duct flow with the lattice Boltzmann method on non-uniform meshes*, 2025.

所以插值晶格波茲曼法依據如下方程式執行:

$$f^{pre-collision(time=n)}_{\text{遷移前空間點 (非網格節點)}} = \sum_{i=-3}^{3}\sum_{j=-3}^{3}\sum_{k=-3}^{3} f_{\hat{i}+i,\,\hat{j}+j,\,\hat{k}+k}\,\ell_i(\delta x)\,\ell_j(\delta y)\,\ell_k(\delta z), \tag{零.6}$$

其中，$f^{pre-collision(time=n)}_{\text{遷移前空間點 (非網格節點)}}$ 為 (D(S(A 非均勻往個節點) 的 (遷移前空間點)) 的 (碰撞前分佈函數))

$$f^{\star}_{\text{遷移前空間點 (非網格節點)}} = f^{pre-collision}_{\text{遷移前空間點 (非網格節點)}} - \Omega(f^{eq}(\vec{r},t), f(\vec{r},t)) \tag{零.7}$$

其中，$\Omega(f^{eq}(\vec{r},t), f(\vec{r},t))$ 為碰撞算子 (Collision Operator)，且上述方程式稱為 LBM 之碰撞步 (collision step)。

$$f^{n+1}_{\text{遷移後空間點 (網格節點)}} = f^{\star}_{\text{遷移前空間點 (非網格節點)}} \tag{零.8}$$

上式則稱為 LBM 之遷移步 (Streaming Step)。

以下為程式碼補充: 補充一:

```
#define F0_Intrpl7(f, i, j, k)      \
    idx = j*nface + k*nline + i;    \
    F0_in = ( f[idx] );
```

Listing 2: interpolationHillISLBM.h9-11

在程式碼規則中，#define 稱為”宏定義”，宏定義內部的變數 idx 不需要特別宣告，只需要在使用到此宏定義的函數或者變數的區域內加以宣告即可。”宏定義”為對特定變數的擬

合器，如上片段程式碼 ?? 所示，”F0_Intrpl7” 作為一個定義式，只要在其他函數中，一講出”F0_Intrpl7”，編譯器就會自動將其替換成定義式內的內容: F0_in = (f[idx]) :

Table 1: 巨集定義與分佈函數變數之對應關係

宏定義	對應變數
#define F0_Intrpl7	F0_Intrpl7 ⇔ F0_in
#define F1_Intrpl7	F1_Intrpl7 ⇔ F1_in
#define F2_Intrpl7	F2_Intrpl7 ⇔ F2_in
#define F3_Intrpl7	F3_Intrpl7 ⇔ F3_in
#define F4_Intrpl7	F4_Intrpl7 ⇔ F4_in
#define F5_Intrpl7	F5_Intrpl7 ⇔ F5_in
#define F6_Intrpl7	F6_Intrpl7 ⇔ F6_in
#define F7_Intrpl7	F7_Intrpl7 ⇔ F7_in
⋮	
#define F18_Intrpl7	F18_Intrpl7 ⇔ F18_in

補充二： 討論：Z 方向的網格劃分與非均勻設定: NZ6 = NZ + 6

在 hyperbolic tangent function 中，z 方向採用如下函數做非均勻網格劃分:

$$y_j = \frac{h}{a}\tanh\left(\frac{\xi_j}{2}\ln\frac{1+a}{1-a}\right)$$

其中，ξ_j 為正規化座標，$\xi_j \in [-1,+1]$ 當 $j=0$ 時，$\xi_j=-1$；當 $j=N$ 時，$\xi_j=1$，所以分母的 N 為切割之網格數而非切割後的節點數。程式碼在initializationTool.h4-7 中有定義這個非均勻網格化分之函數:

```
#define tanhFunction( L, LatticeSize, a, j, N )     \
(           \
    L/2.0 + LatticeSize/2.0 + ((L/2.0)/a)*tanh((-1.0+2.0*(double)(j)/(double)(N
    ))/2.0*log((1.0+a)/(1.0-a)))     \
)
```

Listing 3: initializationTool.h4-7

而在initialization.h 的函數式GenerateMesh_Z() 中，做了如下定義來分割 z 方向的非均勻網格：

```
void GenerateMesh_Z(){...//省略
    double a = GetNonuniParameter();//用以取得最合適之非均勻網格劃分參數
    for( int j = 0; j < NYD6; j++ ){//範圍：遍歷y方向之總節點數
        double total = LZ - HillFunction( y_h[j] ) - minSize;
        //LZ為含山坡之總高度 ;

```

```
        for( int k = bfr; k < NZ6-bfr; k++ ){
          z_h[j*NZ6+k] = tanhFunction(total,minSize,a,(k-3),(NZ6-7)) +
                         HillFunction(y_h[j]);
        }
        z_h[j*NZ6+2] = HillFunction( y_h[j] );
        z_h[j*NZ6+(NZ6-3)] = (double)LZ;
    }
    for( int k = bfr; k < NZ6-bfr; k++ ){
      xi_h[k] = tanhFunction( LXi, minSize, a, (k-3), (NZ6-7) ) - minSize/2.0;
    }
    ....//省略
}
```

Listing 4: initialization.h153-162

從上述程式碼可以發現，Z 方向切割後的網格數量"NZ6-6"，進一步推得，Z 方向之節點數應為"NZ6-6"。所以如果 NZ 被定義為物理空間切割之網格數 (計算點在 Link-Wise systerm 中作為網格中心點)，則 NZ+6 就是總 (包含 buffer) 之計算點數。

補充三：y_h[i] 在哪裡被宣告？為什麼可以直接使用該矩陣？

對於指標變數 double* y_h 的宣告，為什麼單單只在主程式：main.cu 中宣告，而卻在 initialization.h 直接使用到 y_h[i]？(未將指標變數映射到某一個特定記憶體)

我的問題是：如果今天宣告一個矩陣 A ，只要一宣告，則 A 自動作為一個指標行記憶，且映射到 A 矩陣第一個元素的初始存放位址。

```
double A[10] ; //宣告一個尺寸10的矩陣
double B = A ;
//則 B 為 A矩陣 第一個元素的初始存放位址
```

Listing 5: problem disussion

但是如果單單宣告一個指標變數 double* y_h ，則 y_h 作為一個指標變數，還沒有分配記憶體的位址 (映射到某一個記憶體)，並不能直接使用矩陣元素 y_h[i]，且利用指標運算子 (*) : *(y_h + i) 來表示相應矩陣 y_h 第 i 個元素 (y_h[i])

```
double *x_h, *y_h, *z_h, *xi_h ; //宣告y_h作為一個指標變數，尚未映射到任何記憶體
```

Listing 6: main.cu35

```
nBytes = NYD6 * sizeof(double);
AllocateHostArray(  nBytes, 4,  &y_h, &Ydep_h[0], &Ydep_h[1], &Ydep_h[2]);
AllocateDeviceArray(nBytes, 4,  &y_d, &Ydep_d[0], &Ydep_d[1], &Ydep_d[2]);
```

Listing 7: memory.h78-80

```
void AllocateHostArray(const size_t nBytes, const int num_arrays, ...) {
 va_list args;
 //筆記：va_list, va_start, va_arg, va_end 為C/C++中處理可變參數函數的標準宏。
```

```
4   //va_list : 可變參數列表型記憶體
5   // nBytes 為固定參數 。
6   //va_start(可變參數列表型記憶體, 不可變參數型記憶體)：將“不可變參數型記憶體的後
      一個參數位址”映射到“可變參數列表型記憶體”
7   va_start( args, num_arrays );//定義可變參數
8   for( int i = 0; i < num_arrays; i++ ) {
9    double **tmp = va_arg(args, double**);//挑出型別為 double** 的記憶體
10   CHECK_CUDA( cudaMallocHost( (void**)tmp, nBytes) );
11   //對可變參數列表的所有記憶體都分配 nBytes 大小的記憶體空間
12  }
13  va_end( args );
14 }
```

Listing 8: memory.h4-14

```
1  #include <cstdarg>
2  #include <iostream>
3  // 计算任意数量整数的总和
4  int sum(int count, ...) {  // ... 表示可变参数
5
6    va_list args;           // 声明参数列表
7    va_start(args, count); // 初始化,count是最后一个固定参数
8
9    int total = 0;
10   for(int i = 0; i < count; i++) {
11     total += va_arg(args, int);  // 取出下一个 int 类型参数
12   }
13
14   va_end(args);  // 清理
15   return total;
16 }
17
18 int main() {
19   std::cout << sum(3, 10, 20, 30) << std::endl;  // 输出: 60
20   std::cout << sum(5, 1, 2, 3, 4, 5) << std::endl;  // 输出: 15
21 }
```

Listing 9: AllocateHostArray 的模板

其中，AllocateHostArray 稱為映射記憶體函數, 兩件事情可以回答為什麼 double* y_h 可以映射為尺寸大小為「NYD6 * sizeof(double)」的矩陣：y_h[NYD6 * sizesaof(double)] 的第零個元素。

- AllocateHostArray 內部呼叫 cudaMallocHost：
  1. 第一次迴圈 ($i = 0$)：

```
1  tmp = va_arg(args, double**)  // tmp 得到 &y_h
```

```
cudaMallocHost((void**)&y_h, NYD6*sizeof(double))
```

⇒ 分配記憶體 nBytes 給 y_h

2. 第二次迴圈 ($i = 1$)：

```
tmp = va_arg(args, double**)  // tmp 得到 &Ydep_h[0]
cudaMallocHost((void**)&Ydep_h[0], NYD6*sizeof(double))
```

⇒ 分配記憶體 nBytes 給 Ydep_h[0]

3. 第三次迴圈 ($i = 2$)：
   tmp 得到 &Ydep_h[1]，分配記憶體 nBytes 給 Ydep_h[1]
4. 第四次迴圈 ($i = 3$)：
   tmp 得到 &Ydep_h[2]，分配記憶體 nBytes 給 Ydep_h[2]

所以 cudaMallocHost 實現了以指標變數為名稱的矩陣存在，我只要有了一個特定指標變數，透過函數:cudaMallocHost 將此指標變數已指定記憶體大小映射為相對應名稱的矩陣。下面再給更詳細的說明關於函數: cudaMallocHost 的功能:

```
double *y_h;  // 只是一個指標變數名稱

cudaMallocHost((void **)&y_h, 3*sizeof(double));
// 分配記憶體後，y_h 就可以當作陣列使用了！

y_h[0], y_h[1], y_h[2]  // 像陣列一樣存取
```

Listing 10: cudaMallocHost function

補充結束