25. 分佈函數的宣告與統整.tex

Chen Peng Chung

January 12, 2026

Contents

一　前言

此章節為將分佈函數的宣告以及使用進行記錄，大致分類爲兩類，其中一類為函數內宣告、函數內使用，此類分佈函數筆者稱為「簽套分度函數」，通常宣告形式為 double** f_new , double** f_old , double* f_new , double* f_old ; 另外一類則是主程式宣告之分佈函數，筆者稱之為「實際分佈函數」，經過指標陣列之宣告以及分配記憶體後，具體形式為三種 f_h_p[19][NX6*NY6*NZ6] , ft[19][NX6*NY6*NZ6] , fd[19][NX6*NY6*NZ6] ，牽涉到的檔案有：

- main.cu : 實際分佈函數的宣告與使用。
- memory.h ：對實際分佈函數擴展從指標陣列到二維連續記憶體
- evolution.h : 簽套分佈函數的宣告與使用。
- communication.h : 簽套分佈函數的宣告與使用，以及實際分佈函數的使用。

二　簽套分佈函數

此章節將羅列一系列函數式，牽涉雙重指標型記憶體：double** f_old , f_new。
或者指標型單一記憶體：double* f_old , f_new。

二、1　函數式：evolution.h : stream_collid

1. 函數內宣告：__global__ void stream_collid : evolution.h_337-338

```
__global__ void stream_collid(
double* f1_old, double*f2_old, double* f3_old, .....//編號重複
double* f1_new, double*f2_new, double* f3_new, .....//編號重複
//以下Lagrange權重參數
){//....函數內部指令}
```

Listing 1: evolution.h_stream_collid 的 f_old 與 f_new

2. 函數內使用：__global__ void stream_collid : edvolution.h_438

```
if(k == 3){
  //下邊界Half-way bounce-back
  F5_in  = f6_old[index];
  F11_in = f14_old[index];
  F12_in = f13_old[index];
  F15_in = f18_old[index];
  F16_in = f17_old[index];
  }
```

Listing 2: f_old 與、f_new 的使用情況.1

分析：由上述變數的使用可以知道，對於函數內宣告 stream_collid(double*, ...)，(R (A 指標型單一記憶體) 的 (函數內宣告)) 被定義為 (R (T (E (A 指標型單一記憶體) 的 (相應一維連續記憶體)) 的 (初始存放位址)) 的 (宣告)) 所以，此函數內所宣告的指標型單一記憶體 double* f1_old 意味著，此指標型單一記憶體必須指向一個尺寸為 NX6*NY6*NZ6 的一維連續記憶體

3. 實際陣列：f1_old[NX6*NY6*NZ6],f1_new[NX6*NY6*NZ6] 其中，NX6*NY6*NZ6 為陣列尺寸。

二、2 函數式：evolution.h : Launch_CollisionStreaming

在此我做了一點更改與原文不同，因為陣列的函數內部宣告將退化為指標形態，因此此篇為討論雙重指標的函數內部宣告。

1. 函數內宣告：void Launch_CollisionStreaming(double**, double**)：evolution.h_778

```
void Launch_CollisionStreaming(double** f_old, double** f_new){//..
}
```

Listing 3: Launch_CollisionStreaming: 宣告

2. 使用地點：void Launch_CollisionStreaming(double**, double**)：evolution.h_859

```
stream_collide<<<griddim, blockdim, 0, stream0>>>(
  f_old[0],f_old[1]//...第一個參數為分佈函數編號
  f_new[0],f_new[1]//...同上
  //以下爲Lagrange權重參數：逐一放入指標陣列的元素
  //權重指標型一維度連續記憶體的元素
  XPara0_d[0],XPara0_d[1]...
  YPara0_d[0],YPara0_d[1]...
```

Listing 4: Launch_CollisionStreaming: 使用

分析：由上述可以知道，對於函數內部宣告：Launch_CollisionStraming(double**) (T (A 雙重指標型單一記憶體) 的 (函數內部宣告)) 被定義為 (E (R (w (s (A 雙重指標記憶體) 的 (相應二維連續記憶體)) 的 (初始存放位址)) 的 (初始存放位址)) 的 (宣告))。所以，此函數內宣告 double** f_old 意味著，指向的二維連續記憶體第一個指標的空間至少爲 19 個元素，而因為將使用進入函數 stream _collid 中，第二格指標的空間至少爲 NX6*NY6*NZ6。

3. 實際陣列：f_old[19][NX6*NY6*NZ6],f_new[19][NX6*NY6*NZ6]；

二、3 函數式：communication.h : SendSrcToCPU

1. 函數內部宣告：void SendSrcToCPU：communication.h_233

```
void SendSrcToCPU(double** f_new, const size_t nBytes, const int
    num_arrays, ...) {//
}
```

Listing 5: SendSrcToCPU: 宣告

2. 函數內部使用：void SendSrcToCPU：communication.h_237

```
for( int i = 0; i < num_arrays; i++ ) {
  const int dir = va_arg(args, int);
  CHECK_CUDA( cudaMemcpy(fh_p[dir],f_new[dir],nBytes,
   cudaMemcpyDeviceToHost); )
// SendSrcToCPU(f_new, nBytes, 19, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,1
   6,17,18);
```

```
}
```

Listing 6: SendSrcToCPU: 使用

分析：在 cudaMemcpy 的函數簽名中，

```
cudaError_t cudaMemcpy( void* dst, const void* src, size_t count,
    cudaMemcpyKind kind )
```

Listing 7: cudaMemcpy 的函數簽名

對於函數 cudaMemcpy 第二參數為指標型記憶體，實際上，要嵌入的參數為「指向某一塊一維連續記憶體」的指標型單一記憶體，所以，若在函數內使用函數內宣告的雙重指標單一記憶體 f_new 嵌入 cudaMemcpy 的第二參數，則代表將會使用到如下操作：

```
for(int i = 0 ; i < 19 ; i++){
  for(int j = 0 ; j <= nBytes/(sizeof(double)) ; j++){
    *(*(f_new+i)+0+j).... ;
  }
}
```

因此，上述之函數內部宣告，應該要擴充至作為相應二維連續記憶體的初始存放位址的初始存放位址的宣告。

3. 實際陣列：f_new[19][NX6*NY6*NZ6]；

二、4 函數式：communication.h : SendDataToCPU

1. 函數內部宣告：void SendDataToCPU：communication.h_260

```
void SendDataToCPU(double** f_new){//....}
```

Listing 8: SendDataToCPU: 宣告

2. 函數內部宣告：void SendDataToCPU：communication.h_260

```
SendSrcToCPU(f_new, nBytes, 19, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,1
    7,18);
```

Listing 9: SendDataToCPU: 使用

分析: 由於在此函數式內出現為使用 void SendSrcToCPU，如前所述，函數 SendSrcToCPU 的第一個參數，雖然作為指定窗重指標記憶體，但由於對於函數內宣告作了以下操作

```
for(int i = 0 ; i < 19 ; i++){
  for(int j = 0 ; j <= nBytes/(sizeof(double)) ; j++){
    *(*(f_new+i)+0+j).... //嵌入每一個指標陣列，且使用到每一個指標陣列的相對
    應一維連續記憶體;
  }
}
```

，導致該雙重指標之函數內宣告，必須作為相應二維連續記憶體的初始存放位址的初始存放位址的宣告。

3. 實際陣列：f_new[19][NX6*NY6*NZ6]；

三　實際分佈函數

三、1　Host 端二維陣列：fh_p[19][NX6*NY6*NZ6]

下面將針對該二維陣列的宣告，維數擴展，以及使用，進行說明：雖然宣告的時候，表面上作為指標型一為連續記憶體，但是透過記憶體擴展，提升維數至二維陣列。

三、1.1　宣告

主程式宣告: double* fh_p[19] : main.cu_12

```
1 double* fh_p[19];
```

Listing 10: fh_p 的宣告

三、1.2　配置記憶體

維數擴展: void AllocateMemory(const size_t,const int,.....) : memory.h_56

```
1 size_t nBytes;
2 nBytes = NX6 * NYD6 * NZ6 * sizeof(double);
3 AllocateHostArray(nBytes,4,&rho_h_p,&u_h_p,&v_h_p,&w_h_p);\
4 for(int i = 0 ; i < 19 ; i++){
5   CHECK_CUDA(cudaMallocHost((void**)&fh_p[i] , nBytes))
6 //只需要簽入 指標型單一記憶體的初始存放位址
7 //CHECK_CUDA(cudaMallocHost((void**)fh_p+i , nBytes))
8   memset(fh_p[i],0.0,nBytes) ; }
```

Listing 11: 對指標陣列 double* f_h_p 配置記憶體擴展形成二維陣列

三、1.3　實際陣列

當對於一維指標陣列的每一個元素進行記憶體擴展時，整體上將會形成二為陣列，其行數最大值為 NX6*NY6*NZ6 ，列述最大值為 19。

$$\begin{bmatrix} \texttt{(double*) fh\_p[0]} \longrightarrow \texttt{(double) f\_h\_p[0][NX6*NY6*NZ6]} \\ \texttt{(double*) fh\_p[1]} \longrightarrow \texttt{(double) f\_h\_p[1][NX6*NY6*NZ6]} \\ \texttt{(double*) fh\_p[2]} \longrightarrow \texttt{(double) f\_h\_p[2][NX6*NY6*NZ6]} \\ \vdots \\ \texttt{(double*) fh\_p[18]} \longrightarrow \texttt{(double) f\_h\_p[18][NX6*NY6*NZ6]} \end{bmatrix}$$

實際陣列：fh_p[19][NX6*NY6*NZ6]

三、1.4　陣列使用

1. communication.h 執行 MPI 通訊以及 GPU-CPU 資料傳輸的函數中使用：
   - 91 行: MPI_Isend - 發送 fh_p[dir] 資料

```
1 //void Isend_Sideways(const int istart, const int sw_nbr, const int
      itag_sw[23], MPI_Request reqSideways[23], const int num_arrays,
      ...) {
2 for( int i = 0; i < num_arrays; i++ ) {
3   const int dir = va_arg(args, int);
```

```
4     CHECK_MPI(MPI_Isend(
5       (void *)&fh_p[dir][istart]
6       ,1
7       ,DataSideways
8       ,sw_nbr
9       ,itag_sw[i]
10      ,MPI_COMM_WORLD
11      ,&reqSideways[i])
12    );
13  }
```

- 124 行: MPI_Irecv - 接收資料到 fh_p[dir]

```
1  //void Irecv_Sideways(const int istart, const int sw_nbr, const int
       itag_sw[23], MPI_Request reqSideways[23], const int num_arrays,
       ...) {
2  for( int i = 0; i < num_arrays; i++ ) {
3    const int dir = va_arg(args, int);
4    CHECK_MPI(MPI_Irecv(
5      (void *)&fh_p[dir][istart]
6      ,1
7      ,DataSideways
8      ,sw_nbr
9      ,itag_sw[i]
10     ,MPI_COMM_WORLD
11     ,&reqSideways[i])
12   );
13 }
```

- 164 行: cudaMemcpyAsync - 將 fh_p[dir] 從 Host 複製到 Device (f_new)

```
1  //void Wait_Sideways(double *f_new[19], const int iend_sw,
       MPI_Request reqSend[23], MPI_Request reqRecv[23], const int
       transfsize, cudaStream_t stream0, const int num_arrays, ...){//\
       dots
2  CHECK_CUDA(cudaMemcpyAsync(
3    (void *)&f_new[dir][iend_sw]
4    ,(void *)&fh_p[dir][iend_sw]
5    ,transfsize*sizeof(double)
6    ,cudaMemcpyHostToDevice
7    ,stream0)
8  );
```

- 206 行: cudaMemcpyAsync - 將 f_new[dir] 從 Device 複製到 Host (fh_p[dir])

```
1  //void SendBdryToCPU_Sideways(cudaStream_t stream, double *f_new[19],
       const int istart, const int num_arrays, ...) {
2  for( int i = 0; i < num_arrays; i++ ) {
```

```
3   const int dir = va_arg(args, int);
4   CHECK_CUDA( cudaMemcpyAsync((void *)&fh_p[dir][istart],(void *)&
      f_new[dir][istart], nBytes, cudaMemcpyDeviceToHost, stream) );
5 }
6
```

- 225 行: cudaMemcpy - 將 fh_p[dir] 複製到 Device (fd[dir])
- 226 行: cudaMemcpy - 將 fh_p[dir] 複製到 Device (ft[dir])

```
1 //void SendSrcToGPU(const size_t nBytes, const int num_arrays, ...) {
2 for( int i = 0; i < num_arrays; i++ ) {
3   const int dir = va_arg(args, int);
4   CHECK_CUDA( cudaMemcpy( (void*)fd[dir], (void*)fh_p[dir], nBytes,
      cudaMemcpyHostToDevice ) );
5   CHECK_CUDA( cudaMemcpy( (void*)ft[dir], (void*)fh_p[dir], nBytes,
      cudaMemcpyHostToDevice ) );
6 }
7
```

- 239 行: cudaMemcpy - 將 f_new[dir] 從 Device 複製到 Host (fh_p[dir])

```
1 //void SendSrcToCPU(double** f_new, const size_t nBytes, const int
      num_arrays, ...) {
2 for( int i = 0; i < num_arrays; i++ ) {
3   const int dir = va_arg(args, int);
4   CHECK_CUDA( cudaMemcpy(fh_p[dir],f_new[dir],nBytes,
      cudaMemcpyDeviceToHost); )
5 }
6
```

2. initialization.h 初始化分佈函數的平衡態：
   - 36 行: 初始化 fh_p[0][index] (平衡態分佈函數，方向 0)
   - 38-40 行: 初始化 fh_p[dir][index] (平衡態分佈函數，方向 1-18)

```
1 //void InitialUsingDftFunc() {//....
2 fh_p[0][index] = W[0]*rho_h_p[index]*(1.0-1.5*udot); //36行
3 for( int dir = 1; dir <= 18; dir++ ) {
4   fh_p[dir][index] = W[dir] * rho_h_p[index] *(
5   1.0 +
6   3.0 *( e[dir][0] * u_h_p[index] + e[dir][1] * v_h_p[index] + e[dir
      ][2] * w_h_p[index])+
7   4.5 *( e[dir][0] * u_h_p[index] + e[dir][1] * v_h_p[index] + e[dir
      ][2] * w_h_p[index] )*( e[dir][0] * u_h_p[index] + e[dir][1] *
      v_h_p[index] + e[dir][2] * w_h_p[index] )-
8   1.5*udot );//38-40行
9 }
```

3. fileIO.h 輸出分佈函數資料：

- 277-295 行: 輸出 fh_p[0] 到 fh_p[18] 的資料 (共 19 個方向)

```
void fileIO_PDF()
{
OutputData(fh_p[0],  "f0",  myid);
OutputData(fh_p[1],  "f1",  myid);
OutputData(fh_p[2],  "f2",  myid);
OutputData(fh_p[3],  "f3",  myid);
OutputData(fh_p[4],  "f4",  myid);
OutputData(fh_p[5],  "f5",  myid);
OutputData(fh_p[6],  "f6",  myid);
OutputData(fh_p[7],  "f7",  myid);
OutputData(fh_p[8],  "f8",  myid);
OutputData(fh_p[9],  "f9",  myid);
OutputData(fh_p[10], "f10", myid);
OutputData(fh_p[11], "f11", myid);
OutputData(fh_p[12], "f12", myid);
OutputData(fh_p[13], "f13", myid);
OutputData(fh_p[14], "f14", myid);
OutputData(fh_p[15], "f15", myid);
OutputData(fh_p[16], "f16", myid);
OutputData(fh_p[17], "f17", myid);
OutputData(fh_p[18], "f18", myid);
}
```

讀取分佈函數資料：

- 335-353 行: 讀取資料到 fh_p[0] 到 fh_p[18] (共 19 個方向)

```
//void InitialUsingBkpData(){//...
ReadData(fh_p[0],  result, "f0",  myid);
ReadData(fh_p[1],  result, "f1",  myid);
ReadData(fh_p[2],  result, "f2",  myid);
ReadData(fh_p[3],  result, "f3",  myid);
ReadData(fh_p[4],  result, "f4",  myid);
ReadData(fh_p[5],  result, "f5",  myid);
ReadData(fh_p[6],  result, "f6",  myid);
ReadData(fh_p[7],  result, "f7",  myid);
ReadData(fh_p[8],  result, "f8",  myid);
ReadData(fh_p[9],  result, "f9",  myid);
ReadData(fh_p[10], result, "f10", myid);
ReadData(fh_p[11], result, "f11", myid);
ReadData(fh_p[12], result, "f12", myid);
ReadData(fh_p[13], result, "f13", myid);
ReadData(fh_p[14], result, "f14", myid);
ReadData(fh_p[15], result, "f15", myid);
```

```
ReadData(fh_p[16], result, "f16", myid);
ReadData(fh_p[17], result, "f17", myid);
ReadData(fh_p[18], result, "f18", myid);

```

三、1.5 主要功能總結

1. GPU-CPU 之間的資料傳輸（cudaMemcpy）
2. MPI 多處理器間的通訊（MPI_Isend/Irecv）
3. 初始化 LBM 分佈函數的平衡態
4. 檔案 I/O（備份與恢復模擬狀態）

三、2 Device 端二維陣列：fd[19][NX6*NY6*NZ6] 與 ft[19][NX6*NY6*NZ6]

上述之兩者在宣告時型態維指標型一維連續記憶體，使用上應填入 ft = f_new ; fd = f_old (double**) 。

三、2.1 宣告

主程式宣告: double* fd[19] 與 double* ft[19] : main.cu_17

```
double  *ft[19], *fd[19];
```

Listing 12: fd 與 ft 的宣告

三、2.2 配置記憶體

維數擴展：void AllocateMemory() : memory.h_62

```
size_t nBytes;
nBytes = NX6 * NYD6 * NZ6 * sizeof(double);
AllocateDeviceArray(nBytes, 4,  &rho_d, &u, &v, &w);
for( int i = 0; i < 19; i++ ) {
  CHECK_CUDA( cudaMalloc( &fd[i], nBytes ) ); //只需要簽入進去 指標型單一記憶體的初始存放位址
  CHECK_CUDA( cudaMemset( fd[i], 0.0, nBytes ) );
  CHECK_CUDA( cudaMalloc( &ft[i], nBytes ) );
  CHECK_CUDA( cudaMemset( ft[i], 0.0, nBytes ) );}
```

三、2.3 實際陣列

$$\begin{bmatrix} \text{(double*) f\_t[0]} \longrightarrow \text{(double) f\_t[0][NX6*NY6*NZ6]} \\ \text{(double*) f\_t[1]} \longrightarrow \text{(double) f\_t[1][NX6*NY6*NZ6]} \\ \text{(double*) f\_t[2]} \longrightarrow \text{(double) f\_t[2][NX6*NY6*NZ6]} \\ \vdots \\ \text{(double*) f\_t[18]} \longrightarrow \text{(double) f\_t[18][NX6*NY6*NZ6]} \end{bmatrix}$$

$$\begin{bmatrix} \texttt{(double*) f\_d[0]} \longrightarrow \texttt{(double) f\_d[0][NX6*NY6*NZ6]} \\ \texttt{(double*) f\_d[1]} \longrightarrow \texttt{(double) f\_d[1][NX6*NY6*NZ6]} \\ \texttt{(double*) f\_d[2]} \longrightarrow \texttt{(double) f\_d[2][NX6*NY6*NZ6]} \\ \vdots \\ \texttt{(double*) f\_d[18]} \longrightarrow \texttt{(double) f\_d[18][NX6*NY6*NZ6]} \end{bmatrix}$$

實際陣列：`ft[19][NX6*NY6*NZ6]`，`fd[19][NX6*NY6*NZ6]`

三、2.4 陣列使用 `ft[19][NX6*NYD6*NZ6]`

1. communication.h
   - 225 行: cudaMemcpy - 將 fh_p[dir] 從 Host 複製到 fd[dir] (初始化用)

```
CHECK_CUDA( cudaMemcpy( (void*)fd[dir], (void*)fh_p[dir], nBytes,
    cudaMemcpyHostToDevice ) );
```

2. main.cu
   - 173 行: Launch_CollisionStreaming(ft, fd) - 碰撞流動計算，ft 為輸入，fd 為輸出

```
Launch_CollisionStreaming( ft, fd );
```

   - 182 行: Launch_CollisionStreaming(fd, ft) - 碰撞流動計算，fd 為輸入，ft 為輸出

```
Launch_CollisionStreaming( fd, ft );
```

   - 184 行: Launch_TurbulentSum(ft) - 湍流統計計算，使用 ft 作為輸入

```
if( (int)TBSWITCH ) { Launch_TurbulentSum( ft ); }
```

   - 215 行: SendDataToCPU(ft) - 將 ft 資料傳回 CPU (用於全域質量守恆修正)

```
SendDataToCPU( ft );
```

   - 235 行: SendDataToCPU(ft) - 將 ft 資料傳回 CPU (用於質量守恆檢查，每 100 步)

```
SendDataToCPU( ft );
```

   - 260 行: SendDataToCPU(ft) - 將 ft 資料傳回 CPU (模擬結束時輸出)

```
SendDataToCPU( ft );
```

三、2.5 陣列使用 `fd[19][NX6*NYD6*NZ6]`

1. communication.h
   - 226 行: cudaMemcpy - 將 fh_p[dir] 從 Host 複製到 ft[dir] (初始化用)

```
CHECK_CUDA( cudaMemcpy( (void*)ft[dir], (void*)fh_p[dir], nBytes,
    cudaMemcpyHostToDevice ) );
```

2. main.cu
   - 173 行: Launch_CollisionStreaming(ft, fd) - 碰撞流動計算，ft 為輸入，fd 為輸出

```
Launch_CollisionStreaming( ft, fd );
```

- 175 行: Launch_TurbulentSum(fd) - 湍流統計計算，使用 fd 作為輸入

```
    if( (int)TBSWITCH ) { Launch_TurbulentSum( fd ); }

```

- 182 行: Launch_CollisionStreaming(fd, ft) - 碰撞流動計算，fd 為輸入，ft 為輸出

```
Launch_CollisionStreaming( fd, ft );
```