分子動力學模擬方法與技巧

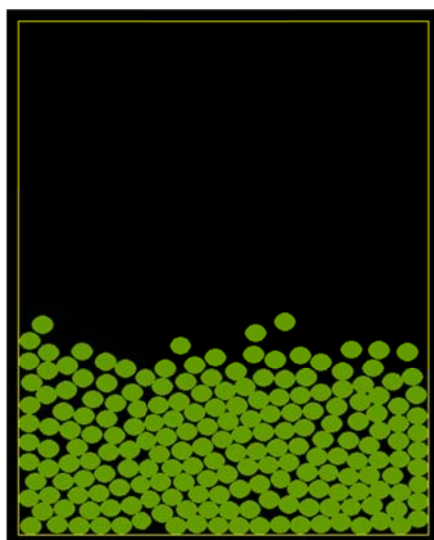
Molecular Dynamics Simulation : Method and Technique

曾飛煥

本講義分為四個章節進行。第一章，將以實例逐步講解如何用 intel fortran f90 程式碼建立分子動力學的二維模擬(2dMDS); 第二章則將程式稍作修改，推廣到兩種類顆粒混合的三維系統(3dMDS); 第三章，轉向介紹如何使用 PGI fortran 的 CUDA 模組將 2dMDS 平行化。最後第四章則會介紹如何使用 CUDA fortran 的平行化來模擬或說求解二維的反應擴散方程。

在往下閱讀講義時，請確定你已安裝好所需要的程式，並且能讓講義附帶的範例程式正常執行，否則只單純的閱讀講義將會枯燥乏味，保證對你學習程式毫無幫助。關於 intel fortran 的安裝與範例執行，請見附錄(A)。附錄(B)是 CUDA Fortran 的安裝設定。附錄(C)是串行程式與平行程式的效率比較。

第一章: 振動容器中二維圓盤顆粒的模擬



圖一:我們即將討論的 2dMDS 在某一瞬間的狀態

我們要模擬的實例為振動容器中二維圓盤狀的顆粒的系統。以下的所有例仔我們都不考慮顆粒的轉動，若想要知道如何將轉動包括進來，可以參考書籍[1,2]。

由於 MDS 的精神是讓每個粒子精確遵守牛頓運動方程，因此我們首要做的事是決定粒子的受力。我們考慮的顆粒只有在碰撞接觸時才有受短程力，其它時候只受重力。真實顆粒不是剛體，碰撞的會引起變形，假設這變形是很小的，則碰撞力就可以用虎克定律來近似。

$$F = -k\xi \tag{1}$$

其中ξ是碰撞引起的形變量，k 是材料的楊氏模量。宏觀上的顆粒碰撞總是有能量的損耗，因此可以再引進損耗項

$$F = -k\xi - 2\beta d\xi/dt \tag{2}$$

讓碰撞時帶有能量損耗。原則上，此時給定參數 k 和β應該就可以計算力了，然而如何適當地選擇參數值卻還未明確。以下的分析就希望把參數與實驗常用的碰撞恢復係數 ε 的數值及碰撞的接觸時間聯繫起來。

現在我們來詳細分析，顆粒 i 和 j 碰撞時的受力方程。設顆粒 i 和 j 的位置項量分別為 $\vec{r}_i$ 和 $\vec{r}_j$ ，速度為 $\vec{v}_i$ 和 $\vec{v}_j$ ，直徑為 d_i 和 d_j ，質量為 m_i 和 m_j 。所以當它們的距離 $\left|\vec{r}_i - \vec{r}_j\right|$ 小於最短距離 $\frac{1}{2}(d_i + d_j)$ 時，表示兩顆粒開始碰撞而碰撞的形變量取為 $\xi = \left|\vec{r}_i - \vec{r}_j\right| - \frac{1}{2}\left(d_i + d_j\right)$ 。兩顆粒的運動方程分別為 $m_i \frac{d^2\vec{r}_i}{dt^2} = F\,\hat{r}_{ij}$, $m_j \frac{d^2\vec{r}_j}{dt^2} = F\,\hat{r}_{ji}$ ，其中 $\hat{r}_{ij} = \frac{\vec{r}_i - \vec{r}_j}{\left|\vec{r}_i - \vec{r}_j\right|}$ 為 i 到 j 的連心單位向量。把兩式相減就得到

$$\frac{d^2}{dt^2}\left(\vec{r}_i - \vec{r}_j\right) = \left(\frac{1}{m_i} + \frac{1}{m_j}\right) F\,\hat{r}_{ij} \tag{3}$$

所以在連心方向上就有形變量 ξ 所滿足的運動方程

$$\frac{d^2\xi}{dt^2} = -\frac{1}{m_{ij}}\left(k\xi + 2\beta\frac{d\xi}{dt}\right) \tag{4}$$

此方程的解為

$$\xi(t) = \frac{v_0}{\Omega} e^{-\beta t/m_{ij}} \sin(\Omega t) \text{ ，} \frac{d\xi(t)}{dt} = \frac{v_0}{\Omega} e^{-\beta t/m_{ij}} \left[-\frac{\beta}{m_{ij}}\sin(\Omega t) + \Omega\cos(\Omega t)\right] \tag{5}$$

其中 $\Omega = \sqrt{k/m_{ij} - \beta^2/m_{ij}^2}$ ，v_0 為開始碰撞時 $t = 0$，兩顆粒的相對速度。碰撞完成時 $t = \pi/\Omega \equiv t_c$，相對速度為 $-v_0 e^{-\beta t_c/m_{ij}}$ ，因此按照定義，恢復係數就為

$$\varepsilon \equiv -\frac{\left[d\xi/dt\right]_{t=t_c}}{\left[d\xi/dt\right]_{t=0}} = e^{-\beta t_c/m_{ij}} \tag{6}$$

這表示我們可以根據實驗中的 t_c 和 ε 的值通過式(6)決定β，在經過Ω決定楊氏模量 k 的值。至此，加上重力後，我們就有每一顆粒子的完整運動方程。

$$m_i\frac{d\vec{v}_i}{dt} = \vec{F}_{ij} - m_i\vec{g} \qquad \frac{d\vec{r}_i}{dt} = \vec{v}_i \qquad \text{其中 } \vec{F}_{ij} = \begin{cases} -\left[k\xi + 2\beta\left(\vec{v}_{ij}\cdot\hat{r}_{ij}\right)\right]\hat{r}_{ij} & if\ \xi < 0 \\ 0 & else \end{cases} \tag{7}$$

決定了運動方程，接下來就是要選擇一種數值方法來解運動方程。

在分子動力學模擬中，很少使用 RK4，原因是 RK4 需要用到 t+0.5dt 的位置與速度去計算 t+dt 的力，而分子動力學模擬中最耗費時間的部分就是作用力的計算，原本每個粒子的演進

都是以一整個 dt 為單位，現在卻要多去估算 t+0.5dt 的數值就顯得多餘。一般上要比較有效的方法是用 Gear predictor-corrector[1]，在這裡我們選用另外一種比較簡單常用的方法，叫做 Verlet method. 由泰勒展開我們知道

$$\vec{r}(t+dt)=\vec{r}(t)+\vec{v}(t)dt+0.5\vec{a}(t)dt+...$$
$$\vec{r}(t-dt)=\vec{r}(t)-\vec{v}(t)dt+0.5\vec{a}(t)dt-... \quad (8)$$

兩式相加和相減分別可得

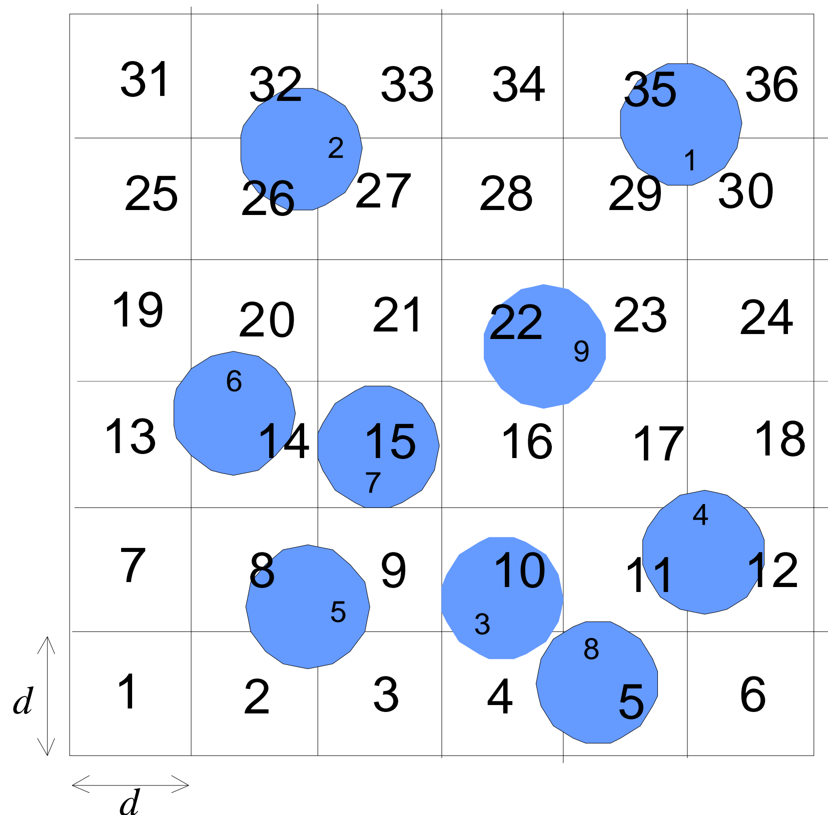
$$\vec{r}(t+dt)=2\vec{r}(t)-\vec{r}(t-dt)+\vec{a}(t)dt^2$$
$$\vec{v}(t)=\left(\vec{r}(t+dt)-\vec{r}(t-dt)\right)/2dt \quad (9)$$

因此，若我們先把 t-dt 時的位置記錄下來，那計算 t+dt 的位置時就不需要用到 t 時的速度。而 t+dt 的速度也由 t+dt 和 t-dt 的位置及 t 時的加速度決定

$$\vec{v}(t+dt)=\left(\vec{r}(t+dt)-\vec{r}(t-dt)\right)/2dt+0.5\vec{a}(t)dt^2 \quad (10)$$

有了積分方法，接下來是演算時最重要的一環，即搜尋碰撞事件的方法。整個 MDS 最耗時間的部分就在於搜尋碰撞事件。由於我們使用的是二體力，因此若有 N 個顆粒，一般上就是要使用 N(N-1)個迴圈來判斷。使用牛頓第三定律，i 和 j 與 j 和 i 的作用力大小相同方向相反，迴圈數可減少一半為 N(N-1)/2。但還是 N 平方的數量級，對顆粒數很大的系統非常不利。N^2 的來源是因為我們會去檢查每一對顆粒是否發生碰撞，但事實上，每一個顆粒只會與它附近的顆粒發生碰撞，不可能與超過兩倍直徑距離的其它顆粒碰撞，因此使用這種逐個判定的方式雖然簡單，卻是很浪費資源。在此，我們使用建立每個顆粒的鄰居的表單，來使迴圈數降至 N 的數量級。

這個方法的想法是這樣的。



圖二: 用來記錄顆粒鄰居的表單示意圖(見內容解說)

假設現在我們有 9 個直徑為 d 的顆粒，36 格長寬也為 d 的方格。上圖中大號碼代表方格的編號，小號碼則為顆粒的編號。這樣的安排讓我們很容易從顆粒的位置知道它處在的格子，例如若 i 顆粒的位置為(x(i),y(i))，則它的格子為 ibox(i)=1+int(x(i)/d)+gridx*int(y(i)/d)。在此例中 gridx = 6，即水平方向格子的數目。另外我們需要儲存編號為 ibox 的格子中有哪些顆粒及共有多少顆粒，我們用 iboxcount(ibox)= iboxcount(ibox)+1 來累加 ibox 格子中顆粒的總數，再用 ipd(ibox,iboxcount)=i 來記錄格子中顆粒的編號。寫成副程式如下:

```
SUBROUTINE neighborlist(ibox ,iboxcount,ipd)
USE GLOBAL
IMPLICIT INTEGER*4(i,j)
DIMENSION iboxcount(inbox),ibox(ipn) !inbox是格子的總數，ipn是顆粒的總數
DIMENSION ipd(5,inbox)    !當顆粒的質心都落在格子的角頭上時，每個格子可能會有4個顆粒，取5為寬鬆條件。
ipd(:,:)=0  !歸零
iboxcount(:)=0  !歸零
DO i = 1, ipn
    ibox(i) = 1+INT(x(i)/d)+INT(y(i)/d+A)*igrid(1) ! i顆粒在ibox格子上，若三維系統，
                                                   ! 就再加上INT(z(i)/d)*igrid(1)*igrid(2)。
                                                   ! 程式中會用pos(i,1)和pos(i,2)代表x(i)和y(i)
    iboxcount(ibox(i))=iboxcount(ibox(i)) + 1      ! 這是ibox格子上的第iboxcount顆粒
    ipd(iboxcount(ibox(i)),ibox(i))   = i          ! ibox 格子上顆粒諦編號
END DO
```

```
END SUBROUTINE
```

有了這些訊息，之後搜尋時就只要找每個顆粒附近 9 格的顆粒就好，其它顆粒就不用費時去判斷了。也因此搜尋遞迴圈只正比於顆粒的數目而非顆粒數目的平方。

接下來我們可以開始寫程式了。

首先我們先寫一個模組設定所需要的變數與參數，檔名為 global.f90。

```
MODULE Global

  REAL*8, ALLOCATABLE, DIMENSION(:,:) :: fos,vel,pos,opos !力，速度，位置與上一時刻位置
  REAL*8 vw,pw,pwh                                        !底盤速度，位置與頂盤位置
  REAL*8 dtsq,dt2,dt                                      !積分時要用的時間
  REAL*8 widthx,height                                    !容器的寬和高
  REAL*8 A,f,gama,normg                                   !振動的振幅，頻率，加速度，規一化的重力加速度
  REAL*8 r,m                                              !規一化的顆粒半徑與質量
  REAL*8 betap,betaw,oomew,oomep,ome,mius,miuk,epp,epw     !規一化的顆粒及容器壁的彈簧參數
  INTEGER ipn,inbox,idraw,igrid(2)                                  !粒子數，格子總數
  PARAMETER (pi=2.0*ASIN(1.0))
  CONTAINS

       SUBROUTINE setconstant(icycle)        !參數設定副程式
       IMPLICIT REAL*8(a-h,k-z) ,INTEGER*4(i,j)

       OPEN (unit=20,file='input.txt')        !從外界讀入常需要更改的參數
       READ (20,*) ipn,widthx,height      !粒子數
       READ (20,*) gama,f,epp,epw,mius,miuk
       READ (20,*) iperiod,idraw           !模擬總週期，畫圖及輸出間隔次數
       READ (20,*) rs,ms                       !真實顆粒半徑與質量
       READ (20,*) igrid(1),igrid(2)           !x,y格子數
       inbox=igrid(1)*igrid(2)                  !格子總數

       d0=0.6                     ! 長度參考單位cm
       w=2*pi*f                   ! 時間參考單位1 / w
       mo=0.2                     ! 質量參考單位gram
       g=980                      ! 重力加速度cm/s^2
       r = rs/d0                  ! 歸一化的半徑
       m = ms/mo                  ! 歸一化的質量
       tc     = w*5.0e-5          ! 歸一化的碰撞時間
       ome    = pi/tc             ! 歸一化的彈簧振動頻率
```

```
        betap=-LOG(epp)/tc          ! 歸一化的顆粒彈簧損耗
        betaw=-LOG(epw)/tc          ! 歸一化的牆壁彈簧損耗
        oomep = betap*betap+ome*ome ! 歸一化的顆粒彈簧自然頻率
        oomew = betaw*betaw+ome*ome ! 歸一化的牆壁彈簧自然頻率
        normg = g/d0/w**2           ! 歸一化的重力加速度
        A     = gama*normg          ! 歸一化的振幅
        div   = 30
        dt    = tc/div              ! 積分時間間隔
        dtsq  = dt*dt               ! verlet方法會用到的
        dt2   = 2.0*dt              ! verlet方法會用到的
        icycle= div*2*pi/tc*iperiod ! 時間迴圈總數
        return
        End subroutine
!!! UNIT VECTORS OF BODY FRAME
    SUBROUTINE unitvector(relp,normV,tangentV)
    IMPLICIT REAL*8 (a-h,k-z)
    DIMENSION relp(2),relv(2),normV(2),tangentV(2)
    DIMENSION nV(2),pdV(2)
      dd          = DSQRT(relp(1)*relp(1)+relp(2)*relp(2))
      normV(1)    = relp(1)/dd ;normV(2)    = relp(2)/dd
      tangentV(1)= -normV(2)   ;tangentV(2)= normV(1)
    END SUBROUTINE
!**************************************************
END MODULE
```

接著，我們要寫一個主程式，檔名為 MDSmain.f90:

```
PROGRAM 2dMDS
USE Global
USE ifqwin
IMPLICIT REAL*8(a-h,k-z) ,INTEGER*4(i,j)
ALLOCATABLE ipd(:,:)
ALLOCATABLE iboxcount(:),ibox(:)

OPEN (unit=15,file='pos.txt') ! 開啟用來存資料的檔案
OPEN (unit=16,file='vel.txt')
OPEN (unit=17,file='fos.txt')
OPEN (unit=18,file='pla.txt')

CALL setconstant(icycle) ! 呼叫外部設定的參數
!!******************************** 決定變數陣列的維度*********************************
```

```
ALLOCATE(iboxcount(inbox),ibox(ipn))
ALLOCATE(ipd(5,inbox))
ALLOCATE(fos(2,ipn),opos(2,ipn),pos(2,ipn),vel(2,ipn))

!!*********************************************************************************
CALL initiate ()   ! 設定顆粒的初始位置與速度

t=0;
     DO it=1,icycle     !!!!!!!!!!!! 開始時間迴圈!!!!!!!!!!!!
        CALL neighborlist(ibox,iboxcount,ipd)      ! 取得t時刻每個顆粒鄰居的資訊
        CALL search_event(ibox,iboxcount,ipd)      ! 搜尋碰撞事件，並計算t時刻的碰撞力

        it1 = it - INT(it / idraw) * idraw
         IF(it1.eq.0) THEN
           CALL drawing()                          ! 每隔idraw步在螢幕上畫出顆粒的位置
           !CALL save_data(t)                      ! 每隔idraw步存取相關數值
        END IF
        CALL integration()                         !進行運動方程的積分，計算t+dt時刻的位置與速度
        t   = t+dt                                 !讓時間前進一步
        vw  = A*DSIN(t)                            ! 下一時刻底盤的速度，頂盤的速度也一樣
        pw  = -A*DCOS(t)                           !下一時刻底盤的位置
        pwh = pw +height                           !下一時刻頂盤的位置
      END DO  ! 時間迴圈結束
iresult= setexitqq(qwin$exitnopersist)  !關閉視窗
END
```

注意，因為我們使用了 d_0,ω^{-1} 為長度和時間的單位，因此原本的簡諧振動 $-A\cos\omega t$ 就變為 $-\bar{A}\cos\bar{t}$ ，而速度為 $\bar{A}\sin\bar{t}$ 。其中 $\bar{A}=A/d_0,\bar{t}=\omega t$ 。只是在程式中，我們略去了變數上方的橫槓。換句話說，如果程式中算出的距離為 1，則對應實際的距離應該是 d_0 ，而時間為 1 對應實際的時間應該是 ω^{-1} ，若模擬中給出的速度為 1，則實際的速度應該是 $d_0\omega$ 。

在時間迴圈開始前我們必須先呼叫副程式 initiate 來設定顆粒的位置和速度，

```
SUBROUTINE initiate()
USE Global
USE ifqwin   !微軟提供的繪圖模組
IMPLICIT REAL*8(a-h,k-z), INTEGER(i,j)
REAL*4 rnd
!********* 畫圖需要的視窗設定***************
```

```
range=100
xs=-range*0.2; zd=-range*0.1  !視窗左上角座標
xe=range*0.6 ;zt=range*0.5    !視窗右下角座標
iresult=setwindow(.true.,xs,zt,xe,zd)
iresult=clickmenuqq(LOC(winfullscreen))
!******************************************
CALL RANDOM_SEED()   !亂數種子
pw  = -A             ! 底盤的初始位置
pwh = pw +height     ! 頂盤的初始位置
vw  = 0.0            ! 容器的初始速度
iresult=setcolorRGB(#00ffff)   ! 設定RGB顏色
iresult=rectangle_w($GBORDER,0.0d0,pw,widthx,pw-r)  !畫出容器的邊界
ix=0;iy=1

DO i=1,ipn           ! 顆粒數迴圈
110 CONTINUE
   ix=ix+1
   IF(MOD(iy,2).eq.0) THEN
     xo=2.0*r+0.5
   END IF

   IF(MOD(iy,2).eq.1) THEN
     xo=r +0.5
   END IF

   pos(1,i)=xo+(ix-1)*2.0*r               ! 顆粒x位置
   IF(pos(1,i)-0.99*widthx>=0) then           ! 若位置超出容器寬度，則往上一層堆疊。
     ix=0 ; iy=iy+1
     GOTO 110
   END IF
   pos(2,i)=1.1*r+(iy-1)*1.75*r+pw        ! 顆粒y位置
   fos(:,i)=0.0                           ! 顆粒受力歸零
   CALL RANDOM_NUMBER(rnd)
      vel(1,i) = 0.1*(0.5-rnd)            ! 顆粒x方向初始速度
      opos(1,i)=pos(1,i)-vel(1,i)*dt2    ! 顆粒前一刻的x位置
   CALL RANDOM_NUMBER(rnd)
      vel(2,i) = 0.1*(0.5-rnd)            ! 顆粒y方向初始速度
      opos(2,i)=pos(2,i)-vel(2,i)*dt2     ! 顆粒前一刻的y位置
      iresult=setcolorrgb(#ff0000)
      iresult=ellipse_w($GFILLINTERIOR,pos(1,i)-r,pos(2,i)+r,pos(1,i)+r,pos(2,i)-r)
```

```
        ! 用實心圓畫出顆粒的位置
END DO
END SUBROUTINE
```

時間迴圈開始後的第一個動作就是去呼叫 neighborlist，建立顆粒處在格子的資訊。有了這個資訊就可以去呼叫 search_event 來搜尋碰撞事件，並計算碰撞的力。

```
SUBROUTINE search_event(ibox,iboxcount,ipd)
USE Global
IMPLICIT REAL*8(a-h,k-z), integer*4 (i,j)
DIMENSION iboxcount(inbox),ibox(ipn)
DIMENSION ib(9)
Dimension ipd(5,inbox)
!這裡共有三層迴圈，最外層的是顆粒數的i迴圈，第二層是鄰居格子數的ik迴圈，最後是每個格子中粒子數的ii迴圈。
DO i=1,ipn
   ibx = ibox(i)  !把i顆粒處在ibox格子的訊息暫存到ibx中
   ib(1)=ibx;ib(2)=ibx+1 ;ib(3)=ibx+igrid(1)-1;ib(4)=ibx+igrid(1);ib(5)=ibx+igrid(1)+1
   !第ibx格子的鄰居加上自己有格，暫存在ib()陣列中。(原本應該有格，但因為有牛頓第三定律，所以只要搜尋右上角
   !的鄰居就可以了)
   DO ik=1,5
        ibx = ib(ik)                         ! 將鄰居格子編號ib暫存到ibx
        ibxc = iboxcount(ibx)                ! 將ibx格子中含有的粒子數目iboxcount暫存到ibxc
        IF (ibx < inbox+1.and.ibxc > 0) then  !當格子編號介於到inbox之間時才往下做
           DO ii = 1,ibxc
              j = ipd(ii,ibx)                ! 將ibx格子中的第ii個顆粒的編號暫存到j
              CALL ppcollision(i,j)          ! 呼叫ppcollision計算i和j顆粒間的碰撞力
           END DO
        END IF
   END DO
   CALL pwcollision(i)                       !檢查完i顆粒與其鄰居的碰撞，也要檢查i顆粒是否與牆壁碰撞
END DO
END SUBROUTINE
```

ppcollision 副程式會接受兩個輸入變數 i 和 j，然後判斷第 i 顆粒與第 j 顆粒是否碰撞，若發生碰撞，則計算碰撞產生的力。

```
SUBROUTINE ppcollision(i,j)
USE Global
IMPLICIT REAL*8(a-h,k-z),INTEGER*4 (i,j)
DIMENSION relp(2),relv(2),normV(2),tangentV(2) !
```

```
relp(1)=pos(1,i)-pos(1,j); relp(2)=pos(2,i)-pos(2,j) !相對位置的x,y分量

dd = DSQRT(relp(1)*relp(1)+relp(2)*relp(2))          !ij顆粒的距離
eta=dd-2*r                                           !ij顆粒在連心方向上的變形量

  IF (eta>-0.5*r.and.eta < 0.0 ) THEN   !若eta介於(-0.5rd,0)，則表示兩顆粒發生了碰撞，所以開始計算碰撞力
      relv(1)=vel(1,i)-vel(1,j); relv(2)=vel(2,i)-vel(2,j)          !相對速度的x,y分量
      mij = 0.5*m                                                   !i和j的化約質量reduced mass
      CALL unitvector(relp,normV,tangentV)                          !這是一個輸入相對位置relp後，會給出連心線
     !的單位向量normV與切方向的單位向量tangentV的副程式，已寫在global裡。
      ForceN = -mij*oomep*eta-2.0*betap* DOT_PRODUCT (relv,normV) !碰撞的連心力，即式()的彈簧力的大小
       Fv = mij*miuk*DOT_PRODUCT(relv,tangentV)                     !與切速度正比的動摩擦力大小
       Fs = mius*ForceN                                             !與正向壓力成正比的靜摩擦力大小
      ForceT = MIN(ABS(Fs),ABS(Fv))                                 !切分量上的力取Fv與Fs間的最小值
      !把粒子坐標系的力(ForceN, ForceT)換成直角坐標系的力fos，累加起來，
      !因為這一時刻可能同時有好幾個粒子都與i顆粒碰撞。
      fx = ForceN*normV(1)-ForceT*tangentV(1)
      fy = ForceN*normV(2)-ForceT*tangentV(2)
      fos(1,i) =fos(1,i)+ fx
      fos(2,i) =fos(2,i)+ fy

      !由於牛頓反作用力原理，所以j號顆粒的力的為大小與i顆粒相同，但方向相反
      fos(1,j)=fos(1,j)-fx
      fos(2,j)=fos(2,j)-fy
  END IF
END SUBROUTINE
```

pwcollision副程式只接收一個變數i，就會去判斷i顆粒是否與邊界碰撞，若有就會計算碰撞力。

```
SUBROUTINE pwcollision(i)
USE Global
IMPLICIT REAL*8(a-h,k-z) , integer*4 (i,j)
!iw = 1,檢查與底盤的碰撞
       eta = (pos(2,i)-pw)-r
       IF (eta < 0.0) THEN
          fos(2,i) = fos(2,i)-m*oomew*eta -2.0*betaw*(vel(2,i)-vw)
       END IF
!iw = 2,檢查與頂盤的碰撞
       eta = (pwh-pos(2,i))-r
       IF (eta < 0.0) THEN
```

```
         fos(2,i) =fos(2,i)+ m*oomew*eta +2.0*betaw*(vel(2,i)-vw)
      END IF
!iw = 3,檢查與左牆的碰撞
      eta =  pos(1,i) -r
      IF (eta < 0.0) THEN
         fosN = -m*oomew*eta -2.0*betaw*vel(1,i)
         frictionz = -m*miuk*(vel(2,i)-vw)
         IF((vel(2,i)-vw)< -0.01) THEN
             frictionz=mius*ABS(fosN)
         END IF
         fos(1,i) = fos(1,i) + fosN
         fos(2,i) =fos(2,i)+ frictionz
       END IF
  !iw = 4,檢查與右牆的碰撞
      eta = (widthx-pos(1,i)) -r
       IF (eta < 0.0) THEN
         fosN = m*oomew*eta + 2.0*betaw*vel(1,i)
         frictionz = -m*miuk*(vel(2,i)-vw)
         IF((vel(2,i)-vw)< -0.01) THEN
             frictionz=mius*ABS(fosN)
         END IF
         fos(1,i) = fos(1,i)+fosN
         fos(2,i) =fos(2,i)+ frictionz
       END IF
END SUBROUTINE
```

經過 search_event中的ppcollision 和pwcollision計算完每一個顆粒受到的碰撞合力後，就可以去呼叫integration來做積分，計算下一步的位置和速度。

```
SUBROUTINE integration()
USE GLOBAL
IMPLICIT REAL*8(a-h,k-z), INTEGER*4(i,j)
DIMENSION posnew(6) !新位置的暫存陣列
DO i=1,ipn
         id=1
             posnew(id)=2.0*pos(id,i)-opos(id,i)+dtsq*(fos(id,i))
             vel(id,i)=(posnew(id)-opos(id,i))/dt2+0.5*dtsq*(fos(id,i))
             opos(id,i)=pos(id,i)
            pos(id,i)=posnew(id)
         id=2
```

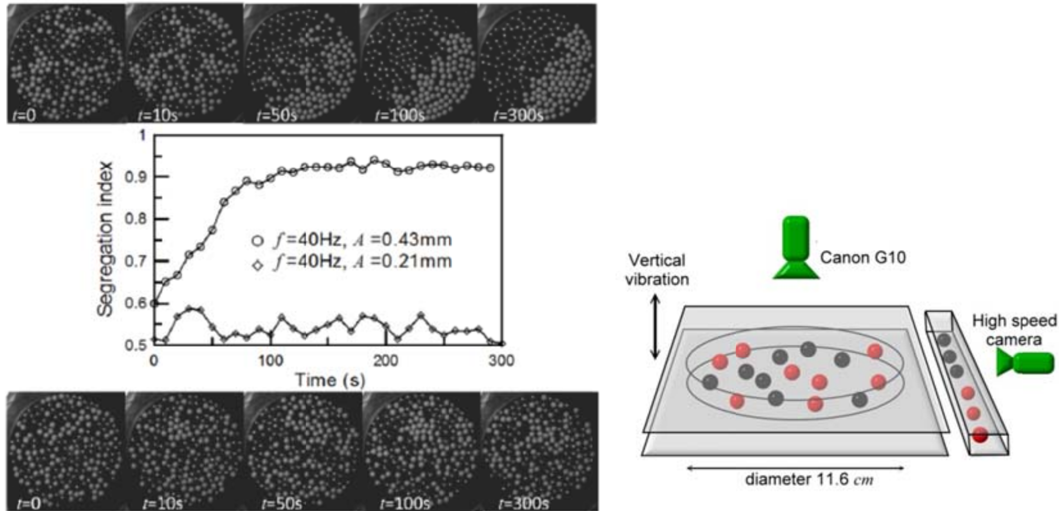
```
            posnew(id)=2.0*pos(id,i)-opos(id,i)+dtsq*(fos(id,i)-normg)
            vel(id,i)=(posnew(id)-opos(id,i))/dt2+0.5*dtsq*(fos(id,i)-normg)
            opos(id,i)=pos(id,i)
            pos(id,i)=posnew(id)

        fos(:,i)=0.0  !! ## RESET THE FORCE
END DO
END SUBROUTINE
```

到此，二維顆粒在振動容器中的分子動力學模擬已經完成。剩下兩個是可有可無的副程式，即存資料save及即時動畫drawing。

```
SUBROUTINE save_data(t)
USE Global
IMPLICIT REAL*8(a-h,k-z)
9 FORMAT(15(f12.4,3x))
DO i=1,ipn
  WRITE(15,9) pos(1:2,i)
  WRITE(16,9) vel(1:2,i)
  WRITE(17,9) fos(1:2,i)
End do
     WRITE(18,9) t/2/pi, pw
END SUBROUTINE

SUBROUTINE drawing()
USE Global
USE IFQWIN
IMPLICIT REAL*8(a-h,k-z)
DATA ired /100/, igreen /255/, iblue /0/
   iresult=setcolorrgb(#000000)
   iresult=rectangle_w($GFILLINTERIOR,-2.0d0,1.5*height,1.5*widthx,-4.0*A)  ! 用黑區塊清除畫面
   icolor=rgbtointeger(ired,igreen,iblue)
   iresult=setcolorrgb(icolor)
      DO i=1,ipn
          iresult=ellipse_w($GFILLINTERIOR,pos(1,i)-r,pos(2,i)+r,pos(1,i)+r,pos(2,i)-r) !畫圓
          iresult=setpixelrgb_w(pos(1,i),pos(2,i),#ffffff) !畫點
      END DO
   iresult=setcolorRGB(#00ffff)
   iresult=rectangle_w($GBORDER,0.0d0,pwh,widthx,pw)
END SUBROUTINE
```

第二章 ： 混合顆粒在振動圓柱形容器中的模擬

當兩種恢復係數不同，但質量與半徑都相同的顆粒，混合後被放在高度是1.5倍顆粒直徑的圓柱形容器中上下振動時，兩種顆粒會自發的分別向同類聚集，最後分成兩個區域[3]。雖然容器高度只允許單層的顆粒，所以是個類二維系統，但它必須使用完整的三維模擬才能產生這個實驗結果。這個章節就是講解能夠模擬出這個實驗結果的MDS。我們稱它為3dMDS。



圖三: 從文獻[]摘錄出來的實驗結果與裝置示意圖

首先我們要修改的是initiate副程式，要把兩種顆粒均勻混合後排列在圓柱容器上。注意，在此模擬中，我們選擇的坐標系是以-z方向為重力場方向，而xy平面為容器底盤。在顆粒位置，速度及作用力的陣列中，我們用奇數編號代表A類顆粒，偶數編號代表B類顆粒。

```
SUBROUTINE initiate()
USE Global
USE ifqwin    !微軟提供的繪圖模組
IMPLICIT REAL*8(a-h,k-z), INTEGER(i,j)
DIMENSION  p1(3),p2(3)
REAL*4 rnd
!********* 畫圖需要的視窗設定***************
range=150
xs=-range*0.4;xe=range*0.4
zd=-range*0.3;zt=range*0.3
iresult=setwindow(.true.,xs,zt,xe,zd)
!*******************************************
CALL RANDOM_SEED()    !亂數種子
pw(:,:) =0.0 ; vw(:,:)=0.0
pw(3,1)=-A              ! 底盤的初始位置
```

```
pw(3,2)=Height+pw(3,1)    ! 頂盤的初始位置
vw(3,1)  = 0.0            ! 容器的初始速度
vw(3,2)  = vw(3,1)
iresult=setcolorRGB(#00ffff)   ! 設定RGB顏色
iresult=elllipse_w($GBORDER,-rp,rp,rp,-rp)  !畫出容器的邊界
IF (rA .ne. rB) THEN
   rss= MAX(rA,rB)     !使用較大的顆粒半徑作為基準
ELSE
   rss=rA
END IF

theta = 0.0
i = 1 ;ir = 0
rr= rp - (1+2*ir)*rss  !從最外圈開始填入顆粒
dtheta = 2.1*rss/rr
pos(1,i) = rr* DCOS(theta) ; pos(2,i) = rr* DSIN(theta); pos(3,i) = rss+pw
x0 = pos(1,i);y0 = pos(2,i)

DO i=2,ipn          ! 顆粒數迴圈
   rr= rp - (1+2*ir)*rss
   theta = theta+dtheta
   pos(1,i) = rr* DCOS(theta) ; pos(2,i) = rr* DSIN(theta); pos(3,i) = rss+pw
   dis2 = DSQRT((pos(1,i)-x0)*(pos(1,i)-x0)+ (pos(2,i)-y0)*(pos(2,i)-y0))
   IF(dis2<rA+rB) THEN
      ir = ir + 1
      rr= rp - (1+2*ir)*rs
      dtheta = 2.1*rs/rr
      theta=0.0
      pos(1,i) = rr* DCOS(theta) ; pos(2,i) = rr* DSIN(theta); pos(3,i) = rss+pw
      x0 = pos(1,i);y0 = pos(2,i)
   END IF
END DO
!PSEUDO MIX PROCESS 虛擬混合過程，隨機選取兩個號碼，讓這兩個號碼的顆粒交換位置，這樣反覆多次，
!可以讓AB兩種顆粒看起來是隨機分布在晶格點上
DO i = 10000
CALL RANDOM_NUMBER(rnd)
  ip1=INT(ipn*rnd)+1
  CALL RANDOM_NUMBER(rnd)
  ip2=INT(ipn*rnd)+1
  IF((ip1.ne.ip2).and.(ip1<ipn).and.(ip2<ipn)) THEN
```

```
      DO id=1,3
         p1(id)=pos(id,ip1) ;p2(id)=pos(id,ip2)
         pos(id,ip1)=p2(id) ;pos(id,ip2)=p1(id)
      END DO
   END IF
END DO

DO i=1,ipn
    m(i) = mA;rd(i) = rA
    iresult=setcolorrgb(#ff0000)
    IF (MOD(i,2).eq.0) THEN
         m(i) = mB;rd(i) = rB
        iresult=setcolorrgb(#0000ff)
    END IF
      iresult=ellipse_w($GFILLINTERIOR,pos(1,i)-rd(i),pos(2,i)+rd(i),pos(1,i)+rd(i),pos(2,i)-rd(i))
    DO id=1,3
      fos(id,ib)=0.0
      CALL RANDOM_NUMBER(rnd)
      vel(id,i) = 0.1*(0.5-rnd)
      opos(id,i)=pos(id,i)-vel(id,i)*dt2
    END DO
END DO
END SUBROUTINE
```

接著我們先到global模組中修改一些參數設定。

1. 關於容器的座標與速度改為 pw(3,3), vw(3,3)。第一個指標為x,y,z分量，第二個指標表示底盤，頂盤和圓周邊界。
2. 容器的幾何參數改為 rp, height
3. 顆粒的質量和半徑改用mA,mB,rA,rB，另外因為顆粒質量半徑可以不同，所亦要宣告陣列m和rd。
4. 彈簧參數betaAA,betaBB,betaAB,oomeAA,oomeBB,oomeAB,epAA,epBB,epAB,其它特徵則簡單的假設兩種顆粒都一樣，因此不在參數名稱上給予區別，如mius,miuk,ome。
5. 改陣列igrid(3) 成igridx和igridxy

自然的，在setconstant副程式中必須做相對應的修改。其中要注意的是現在要做三維網格來記錄顆粒鄰居訊息，所以inbox = igrid(3)*igridxy (igridx=igrid(1),igridxy=igridx*igrid(2))。

在主程式中，只有一個地方要修改，就是容器的運動。使用下面四行程式碼取代

```
     vw(3,1) = A*DSIN(t)             ! BOTTOM PLATE VELOCITY
    pw(3,1) =-A*DCOS(t)              ! BOTTOM PLATE POSITION
```

```
        vw(3,2) = vw(3,1)               ! UPPER PLATE VELOCITY
        pw(3,2) = pw(3,1) + Height      ! UPPER PLATE POSITION
```

另外記得決定m和rd陣列的維度。

neighborlist中格子編號的式子要變成

```
ibox(i) = 1+INT(pos(1,i)+rp)+INT((pos(2,i)+rp))*igridx +INT((pos(3,i)+A))*igridxy
```

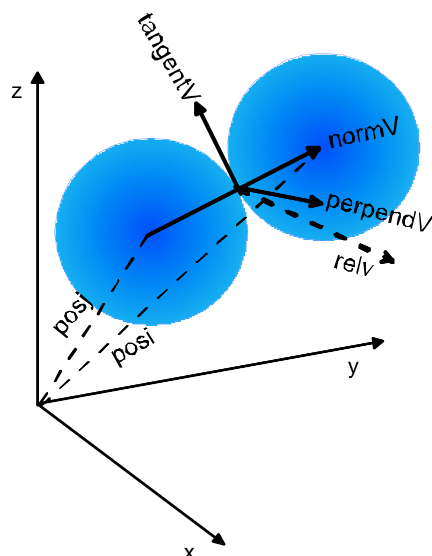
加上rp只是為了把圓盤中心移到(rp,rp)處，使得ibox都是正數指標。

search_event中搜尋的格子數目要從5增加到14。一樣的，原本三維空間中應該要搜尋27個格子，因為有牛頓第三定律，因此減半。ib的陣列數值要改成

```
ib(1)=ibx ; ib(2)=ibx  +1 ; ib(3)=ibx+igridx  ; ib(4)=ib(3)+1 ;  ib(5)=ib(3)-1
ib(6)=ib(1)+igridxy  ; ib(7) =ib(6) +1 ;   ib(8)=ib(6)-1  ; ib(9)=ib(6)+igridx
ib(10)=ib(9) +1  ;  ib(11)=ib(9)-1 ;  ib(12)=ib(6)-igridx  ; ib(13)=ib(12)+1
ib(14)=ib(12)-1
```

ik的迴圈變成1到14。

ppcolision和pwcollision用下列程式碼取代。要注意的地方是三維空間中，顆粒碰撞時的連心座標系統與靜止的實驗室座標系統之間的變換。兩個座標系的關係定義如下。



圖四: 實驗室座標與兩顆粒的連心座標關係示意圖

實驗室坐標系的單位向量為x,y,z，而連心座標系的單位向量計為normV,perpendV,和tangentV。normV由兩個粒子的位置向量定義:normV = (posi-posj)/ |posi-posj|，而perpendV由normV和粒子的相對速度relv的外積定義:perpendV = relv x normV/ |relv x normV|。最後再由normV和perpendV外積定義:tangentV = perpendV x normV/ | perpendV x normV|。這些動作都寫在global模組中unitvector和krossproduct這兩個副程式中完成。這樣的定義會使得相對速度在切平面上的投影必然與tangentV平行，也就會是摩擦力的反方向。而perpendV就會是因為摩擦力所產生的力矩方向，因為我們不考慮轉動，因此perpendV在此的作用只是用來定出tangentV，而不會有力的分量出現。

```fortran
SUBROUTINE ppcollision(i,j)
USE Global
IMPLICIT REAL*8(a-h,k-z),INTEGER*4(i,j)
DIMENSION relp(3),relv(3),normV(3),tangentV(3),perpendV(3)  ! 由於三維系統，多了一個變數陣列perpendV

relp(1)=pos(1,i)-pos(1,j);relp(2)=pos(2,i)-pos(2,j);relp(3)=pos(3,i)-pos(3,j) ! i,j顆粒相對位置分量
dd = DSQRT(dot_product(relp,relp))    ! 兩顆粒質心距離
eta = dd-(rd(i)+rd(j))                ! 形變量

IF (eta>-0.25.and.eta<0.0) THEN       !若形變量介於(-0.25,0.0)之間則計算碰撞力
    relv(1)=vel(1,i)-vel(1,j);relv(2)=vel(2,i)-vel(2,j);relv(3)=vel(3,i)-vel(3,j)
    ! i,j顆粒相對速度分量
    CALL unitvector(relp,relv,normV,tangentV,perpendV)   ! 計算顆粒連心坐標系的單位向量
    mij= m(i)*m(j)/(m(i)+m(j))                           ! 兩顆粒約化質量
    IF ((MOD(i,2).eq.1).and.(MOD(j,2).eq.1)) THEN   !判斷兩顆粒是否同類
      oome = oomeAA;  beta = betaAA
      ELSE IF ((MOD(i,2).eq.0).and.(MOD(j,2).eq.0)) THEN
      oome = oomeBB;  beta = betaBB
      ELSE
      oome = oomeAB;  beta = betaAB
    END IF

    ForceN = -mij*oome*eta-2.0*beta*dot_product(relv,normV)  ! 連心方向的受力
    ForceT = -mij*miuk*dot_product(relv,tangentV)            ! 碰撞切平面上的摩擦力
    !ForceP = 0  沒有力分量
    DO id =1,3
       !將顆粒隨體坐標系上的受力分量ForceN,ForceT,ForceP換回實驗室坐標系。
       Labforce = (ForceN*normV(id)+ForceT*tangentV(id))
       fos(id,i) =fos(id,i) + Labforce
       fos(id,j) =fos(id,j) - Labforce
    END DO
END IF
END SUBROUTINE

SUBROUTINE pwcollision(i)
USE Global
IMPLICIT REAL*8(a-h,k-z),INTEGER*4(i,j)
DIMENSION relp(3),relv(3),normV(3),tangentV(3),perpendV(3)
rr=rd(i)
```

```
!iw = 1 是否與底盤碰撞
     eta= (pos(3,i)-pw(3,1))-rr
     IF(eta<0.0)  THEN
           IF (MOD(i,2).eq.1) THEN !判斷是哪類顆粒
              oome = oomeAA;  beta = betaAA
           ELSE
             oome = oomeBB;  beta = betaBB
           END IF
        fos(3,i) = fos(3,i) - m(i)*oome*eta - 2.0*beta*(vel(3,i)-vw(3,1))
        fos(2,i) = fos(2,i) - miuk*vel(2,i)
        fos(1,i) = fos(1,i) - miuk*vel(1,i)
     END IF

 !iw = 2  是否與頂盤碰撞
     eta= (pw(3,2)-pos(3,i))-rr
     IF(eta<0.0)  THEN
           IF (MOD(i,2).eq.1) THEN !判斷是哪類顆粒
              oome = oomeAA;  beta = betaAA
           ELSE
             oome = oomeBB;  beta = betaBB
           END IF

        fos(3,i) = fos(3,i) + m(i)*oome*eta - 2.0*beta*(vel(3,i)-vw(3,2))
        fos(2,i) = fos(2,i) - miuk*vel(2,i)
        fos(1,i) = fos(1,i) - miuk*vel(1,i)
     END IF

 !iw = 3  是否與邊界碰撞。因為邊界是圓，所以我們把邊界當成是一顆大球來處理
     dd = dsqrt((pos(1,i))*(pos(1,i))+(pos(2,i))*(pos(2,i))) !i顆粒與圓盤中心的水平距離
     cos = (pos(1,i))/dd ; sin = (pos(2,i))/dd                !i顆粒在水平面上的方向
     pw(1,3)=rp*cos; pw(2,3)=rp*sin ;pw(3,3)=pos(3,i)         !假設大球處在和顆粒同方向上，距離中心rp位置
     eta = rp -(dd+rr)        !碰撞形變量
     IF(eta < 0.0)  THEN
           IF (MOD(i,2).eq.1) THEN !判斷是哪類顆粒
              oome = oomeAA;  beta = betaAA
           ELSE
             oome = oomeBB;  beta = betaBB
           END IF

           DO id=1,3
```

```
            relp(id) = pos(id,i)-pw(id,3)
            relv(id) = vel(id,i)-vw(id,3)
         END DO
        CALL unitvector(relp,relv,normV,tangentV,perpendV)
        ForceN = -m(i)*oome*eta+2*beta*dot_product(relv,normV)
        ForceT = -m(i)*miuk*dot_product(relv,tangentV)
        ForceP = 0
       DO id =1,3
         fos(id,i) =fos(id,i)+ (ForceN*normV(id)+ForceT*tangentV(id)+ForceP*perpendV(id))
       END DO
    END IF
END SUBROUTINE
```

到此，我們已經完成了這個程式的主要架構，剩下是即時動畫和數據儲存的部分就看個人想要如何呈現。由於IFQWIN繪圖模組只能呈現二維，因此這裡我們提供的副程式是用俯視與側視同時呈現作為例子。

```
SUBROUTINE drawing()
USE Global
USE IFQWIN
IMPLICIT REAL*8(a-h,k-z)
l = 1.5*rp
          iresult=setcolorrgb(#000000)
          iresult=rectangle_w($GFILLINTERIOR,-l,l,l,-l)  ! DRAW BLACK TO CLEAR SCREEN
          iresult=setcolorrgb(#00ff00)

          ired= 100 ;igreen=155;iblue=0
            icolor=rgbtointeger(ired,igreen,iblue)
            iresult=setcolorrgb(icolor)
          DO ib=1,ipn
            iresult=setcolorrgb(#ff0000)
            IF (MOD(ib,2).eq.0) THEN
               iresult=setcolorrgb(#0000ff)
            END IF
          iresult=ellipse_w($GFILLINTERIOR,pos(1,ib)-0.5,pos(2,ib)+0.5,pos(1,ib)+0.5,pos(2,ib)-0.5)
          iresult=ellipse_w($GFILLINTERIOR,pos(1,ib)-0.5,pos(3,ib)+0.5-l,pos(1,ib)+0.5,pos(3,ib)-0.5-l)
          END DO
iresult=setcolorRGB(#00ffff)
iresult=ellipse_w($GBORDER,-rp,rp,rp,-rp)  ! DRAW CIRCULAR PLATE
END SUBROUTINE
```

第三章 ：Cuda fortran 平行化MDS

我們將以第一章的2dMDS來講解如何運用PGI fortran的Cuda模組將程式平行化。關於cuda fortran的安裝和設定，請參考附錄(B)。

在進入程式前，我們需要先講解一些cuda平行運算的基本概念。不是所有的程式都可以被平行化，一般上，只有那些沒有因果關係的部分程式才可以被平行化。在我們的MDS程式中，t時刻時，每個粒子受力的計算是可以被平行化的，因為第i個粒子的受力並不會影響同一t時刻下其它粒子的受力。因此，每個粒子的受力計算是相互獨立的，可以同時被計算，即平行化。然而，一般的硬體架構只有一個計算中心(CPU)，程式只能按照順序一行一行執行，每一瞬間只能處理一個指令，因此必須做完第一顆粒子的受力計算後才能計算下一個粒子的受力，我們稱這樣的演算流程為串行運算。

GPU的架構原本是用來處理圖形影像的硬體，影像一般由螢幕上的畫素組成，每個畫素要表現的顏色亮度的訊息是相互獨立的，因此最適合使用平行運算。如果單靠CPU來處理這些影像，畫素太高時，影像就會顯得不流暢。GPU硬體中就有許多獨立的簡單計算中心，可以同時處理螢幕上的每個畫素，而達到快速流暢的畫質效果。Nvidia在2007年正式推出了CUDA的架構平台，通過高階C語言可以用呼叫副程式的方式把CPU的運算分配到個人電腦的顯示卡上工作，讓顯示卡變成了個人層級的叢集運算平台，大大提高了許多科學研究的速度。

CUDA 將CPU稱為主機端(Host)，而GPU稱為(Device)。整個程式仍由Host主導，決定如何分配記憶體，何時讓Device進行平行工作。CUDA Fortran的程式流程架構一般如下:

```
Program cuda
   宣告Host上的記憶體
   宣告Device上的記憶體
   .........................
   串行程式碼
   .........................
   將Host的資料複製到Device上
   CALL <<<Grid,Block,Thread >>>(arguments)    呼叫CUDA 核心副程式(Kernel)進行平行運算
   將Device上的資料複製回Host上以便輸出或進行其他串行運算
   .........................
   串行程式碼
   .........................
End Program
```

在Kernel程式中，會有三個層次。第一層叫做網格Grid，這只有在多張GPU顯示卡中會使用到。若只有一張顯卡時，此參數可以省略不寫。每個Grid中會分成許多區塊(Block)，Block

的個數不限，但受限於卡的能力，而每一個Block中會有執行緒(Thread)，每一個執行緒就是一個可以獨立處理基本運算的單元。例如 <<< 512, 256>>>表示我要使用512個blocks，每個blocks中安排256個threads，也就是說，我可以有512x256共131072個運算單元進行平行運算。目前不管GPU能力多少，每一個Block最多只能有1024個threads。很重要的一點是，同一個block中的執行緒可以共用一些記憶體，目前大概是48kb，但不同block中的threads是不能共用記憶體的。例如2號block中14號thread計算的某個x值不能直接被5號block中的任何一個thread使用。除非先把x的值放到block以外的一個全域記憶體中，這樣大家才可以共用。其它細節與技巧請參考cuda網頁的說明或書籍[4]。

在開始把第一章的程式平行化之前，我們先來舉個簡單的例子。我要使用512 X 256=131072個亂數x分別代入方程式$x_{n+1} = \sin(x_n)$中計算出疊代100次後的512x256個數值。一般的串行程式碼為

```
PROGRAM series
REAL*4 x(512*256)
CALL random_number(x)  !產生131072個初始亂數
DO i = 1, 512*256      !每個數字的迴圈
   DO j=1,100          !疊代100次迴圈
     x(i) = sin(x(i))
   END DO
END DO
WRITE(*,*) x(:)        !結果輸出
END PROGRAM
```

在這個例子中i迴圈是可以平行化的，因為計算x(5)的值時會用到這個時候x(5)的值，而不會用到或影響到其他x(6)，x(100)等等的值，然而j迴圈就無法平行化，因為計算第j步的x值是與j-1步時的x值有關的，而且sine函數是非線性的，我們沒有辦法將j迴圈拆成100個來各自計算後再用線性的方法疊加起來得到正確的答案。

接下來我們就把上面的i迴圈平行化。首先我們需要宣告兩套變數Hx(512*256)和Dx(512*256)，分別代表主控端Host和裝置端Device上的記憶體位置。並宣告Block和Thread的數目，NB和NT

```
PROGRAM parallel
USE cudafor !記得要使用 cuda函式庫
REAL*4 Hx(512*256)
REAL*4 Dx(512*256)
INTEGER NB,NT
NB = 512 ; NT = 256
idevice = 0
istat   = cudaSetDevice(idevice)  !啟動具有cuda計算能力的裝置，並編號為0，因為目前只有一個裝置。
!接下來產生初始亂數，存在Host上，並把他複製一份放到Device上
CALL random_number(Hx) !產生131072個初始亂數
```

```
Dx = Hx  !把Hx存到Dx中
CALL iteration<<<NB,NT >>> (NT) !呼叫iteration副程式，並傳入NT參數，這時每一個Device上的Thread都會執行同樣的iteration副程式，但是會根據每個執行緒內定好的編號去讀取對應的數值，這相當於把i迴圈拆開成131072個獨立的計算。
```

Device計算完畢後，數據還存在Device上的記憶體，主控端無法直接讀取，因此必須先把它複製回到主控端的記憶體上

```
Hx = Dx  !把Dx存回到Hx中
WRITE(*,*) Hx(:)   !結果輸出
END PROGRAM
```

在Device上運行的副程式的寫法如下，前面必須加上ATTRIBUTES(global)

```
ATTRIBUTES(global) SUBROUTINE iteration(NT)
INTEGER,VALUE :: NT
INTEGER  i,j
i =  (blockIdx%x -1)*NT+threadIdx%x !把第i迴圈做的事交給第blockIdx%x區塊中的第threadIdx%x執行緒來做
                                    ! blockIdx%x和threadIdx%x就是CUDA內定好的編號。在這個例子中，
                                    ! blockIdx%x = 1到NB 而threadIdx%x=1到 NT
  DO j=1,100  !疊代迴圈
     Dx(i) = sin(Dx(i))
  END DO
END SUBROUTINE
```

雖然在主程式中我們只呼叫一次iteration，但這一次呼叫是同時呼叫了NB個區塊中NT個執行緒。每一個執行緒會分別到Device上的記憶體中讀取各自的Dx值，然後各自執行j迴圈。疊代100次的j迴圈所需的時間是很短的，用串行的方式就要重複這樣的迴圈131072，累積起來的時間就相當可觀。而平行化之後，除了一開始記憶體複製時花掉的時間，其它運算的時間其實就只相當於疊代100次的時間，理論上就會省了10萬倍的時間! 當然實際上並無法達到這麼高的的倍數。原因有好幾個，一個是剛剛說的記憶體的複製會花去很多時間，因此如何有效的安排資料的讀取與交換是很關鍵的。另外Device上每個Thread的運算時脈一般都比CPU慢，而且我們使用10萬個Thread運算時，並不會真正有10萬個同時在計算，還是會有一些先後的差異。更詳細的資料請參閱PGI官網或書籍<高效能運算之CUDA>[4]。

現在讓我們開始來寫平行化的2dMDS程式，我們把它稱為parallel_2dMDS。

首先我們還是先有一個Global的模組，先把變數和參數設定好。大致與第一章的Global模組相同，但少了unitvector的副程式及多了兩個變數NT和NB，分別是每個block中Thread的數目和Block的個數，最好取成2^n，如32，64，128。因為我們要把每個顆粒的運算分給不同的thread去執行，因此NTxNB要剛好等於粒子數ipn。

```
MODULE Global
  REAL*8,ALLOCATABLE,DIMENSION(:,:) :: fos,opos,pos,vel ! 這些變數名稱是設定在主控端Host上的
  REAL*8 vw,pw,pwh
  REAL*8 dtsq,dt2,dt
  REAL*8 widthx,height
```

```
REAL*8 A,f,gama,normg
REAL*8 rd,m
REAL*8 betap,betaw, oomew, oomep,ome,mius,miuk,epp,epw
INTEGER ipn,idraw,inbox,igrid(2)
INTEGER NB,NT ! Block的數目NB 和 每個block中Thread的個數NT
PARAMETER (pi = 2.0*ASIN(1.0))

CONTAINS

     SUBROUTINE setconstant(icycle)
     IMPLICIT REAL*8(a-h,k-z),INTEGER(i,j)

     OPEN (unit=20,file='input.txt')
     READ(20,*) widthx,height
     READ (20,*) gama,f,epp,epw,mius,miuk
     READ (20,*) ipn,iperiod,idraw
     READ (20,*) rs,ms  !實際顆粒半徑(cm)與質量(gram)
      READ (20,*) igrid(1),igrid(2),NB,NT

     d0 = 0.6                   ! 長度參考單位 cm
     m0 = 0.2                   ! 質量參考單位 g
     w=2*pi*f                   ! 時間參考單位 1 / w
     m = ms / m0                ! 歸一化的質量
     rd = rs / d0               ! 歸一化的半徑
     g=980                      ! 重力加速度 cm/s^2
     tc    = w*50.0e-5          ! 歸一化的碰撞時間
     ome   = pi/tc              ! 歸一化的彈簧振動頻率
     betap=-LOG(epp)/tc         ! 歸一化的顆粒彈簧損耗
     betaw=-LOG(epw)/tc         ! 歸一化的牆壁彈簧損耗
     oomep = betap*betap+ome*ome !歸一化的顆粒彈簧自然頻率
     oomew = betaw*betaw+ome*ome !歸一化的牆壁彈簧自然頻率
     normg = g/d0/w**2          ! 歸一化的重力加速度
     A     = gama*normg         ! 歸一化的振幅
     div   = 35
     dt    = tc/div             ! 積分時間間隔
     dtsq  = dt*dt              ! verlet方法會用到的
     dt2   = 2.0*dt             ! verlet方法會用到的
     icycle= div*2*pi/tc*iperiod  ! 時間迴圈總數
     inbox = igrid(1)*igrid(2)
```

```
      RETURN
      END SUBROUTINE

END MODULE
```

接下來我們要寫一個命名為gpu的模組設定裝置端Device上的變數與需要執行的副程式。

```
MODULE gpu
USE cudafor   !使用cuda fortran api
INTEGER,DEVICE,ALLOCATABLE,DIMENSION(:) :: Dbox,Dboxcount ! 以下這些變數都是開在GPU上的全域記憶體中，變數的
INTEGER,DEVICE,ALLOCATABLE,DIMENSION(:,:) :: Dipd         ! 意義都和Global模組中對應。
REAL*8,DEVICE,ALLOCATABLE,DIMENSION(:,:) :: Dpos,Dvel,Dfos,Dopos
REAL*8,DEVICE :: Ddt2,Ddtsq,Dbetap,Dbetaw,Dmius,Dmiuk,Doomep,Doomew
REAL*8,DEVICE :: Dm,Drd,DA,Dnormg,Dwidthx,DHeight,Dpw

  CONTAINS

     ATTRIBUTES(global) SUBROUTINE neighborlist(NT,grx) !我們把neighborlist副程式和integration副程式放在一
     INTEGER,VALUE ::  NT, grx                          !起，都交由GPU處理，也就是說
     REAL*8 posnew(2)                                   !當呼叫這個副程式時，以下運算將由GPU執行
     INTEGER iwpx,iwpy,ibt,ibx,istate,id

        i =  (blockIdx%x-1)*NT+threadIdx%x  ! 這一行是重點，NT是Thread的總數，blockIdx%x和threadIdx%x是cuda
                                   !內定的變數，分別代表block和thread的編號。所以這一行的意思是我們把關於第i
                                   !編號粒子的計算交由(blockIdx%x,threadIdx%x)的執行緒來處理
        id = 1
                posnew(id)=2.0*Dpos(id,i)-Dopos(id,i)+Ddtsq*Dfos(id,i)
                Dvel(id,i)=(posnew(id)-Dopos(id,i))/Ddt2 +0.5*Ddtsq*Dfos(id,i)
                Dopos(id,i)=Dpos(id,i)
                Dpos(id,i)=posnew(id)
       id = 2
                posnew(id)=2.0*Dpos(id,i)-Dopos(id,i)+Ddtsq*(Dfos(id,i)-Dnormg)
                Dvel(id,i)=(posnew(id)-Dopos(id,i))/Ddt2 +0.5*Ddtsq*(Dfos(id,i)-Dnormg)
                Dopos(id,i)=Dpos(id,i)
                Dpos(id,i)=posnew(id)
      Dfos(1,i)=0.0 ;Dfos(2,i)= 0.0

      iwpx = INT(Dpos(1,i)); iwpy = INT(Dpos(2,i)+DA)
     Dbox(i) = 1+iwpx+iwpy*grx
      ibx = Dbox(i)
```

```
      istate = atomicadd(Dboxcount(ibx),1) !atomicadd(x,1)是CUDA的累加函數，它相當於x = x+1，cuda 3.2以下
                                           !版本下只能用在整數的累加，4.0才能做浮點數累加。Atomic的意思是每一次
                                           !只允許一個執行緒去更動x的值，不這樣設計的話，平行運算時可能有
                                           !好幾個執行緒同時去讀x的值就會出錯。
      ibt=istate+1
      Dipd(ibt,ibx) = i
   END SUBROUTINE

   ATTRIBUTES(global) SUBROUTINE collision(inbox,NT,grx,Dpw,Dpwh,Dvw) !計算受力的部分也是交由GPU處理
   INTEGER,VALUE :: inbox,NT,grx
   REAL*8, VALUE :: Dpw,Dpwh,Dvw
   REAL*8 relp(2),vn(2),normV(2),tangV(2)
   REAL*8 pi(2),vi(2),fi(2)
   REAL*8 eta,dis,dd,vndotnormV,FosN,vndottangV,friction
   INTEGER ibt,ibx,istate,id,ik,i,j,ii,ibxc
   INTEGER ib(9)

   i =  (blockIdx%x -1)*NT+threadIdx%x !把關於第i編號粒子的計算交由(blockIdx%x,threadIdx%x)的執行緒來處理
    pi(1)=Dpos(1,i);i(2)=Dpos(2,i) !因為要多次用到，所以先把全域記憶體中i顆粒的位置，速度和力
    vi(1)=Dvel(1,i);vi(2)=Dvel(2,i) ! 放到GPU上的暫存器變數中
    fi(1)=Dfos(1,i);fi(2)=Dfos(2,i)
         !###############  WALL  collision ################## 先檢查是否與邊界碰撞
 !iw=1 與底盤是否碰撞
                 eta = (Dpi(2)-Dpw) - Drd
                     IF (eta < 0.0) THEN
                           Dfi(1) = Dfi(1) - Dmiuk* Dvi(1) !摩擦力
                           Dfi(2) = Dfi(2) - Doomew*eta-Dbetaw*(Dvi(2)-Dvw) !碰撞力
                     END IF
!iw= 2 與頂盤是否碰撞
                 eta = (Dpwh-Dpi(2)-Drd)
                     IF(eta < 0.0) THEN
                            Dfi(1) = Dfi(1)- Dmiuk* Dvi(1) !摩擦力
                           Dfi(2) = Dfi(2)+ Doomew*eta-Dbetaw*(Dvi(2)-Dvw) !碰撞力
                     END IF
!iw= 3 與左邊是否碰撞
                 eta = Dpi(1)-Drd
                  IF (eta < 0.0) THEN
                          Dfi(1) = Dfi(1)-Doomew*eta-Dbetaw*Dvi(1)
                          Dfi(2) = Dfi(2)-Dmiuk* (Dvi(2)-Dvw)
                  END IF
```

```
!iw= 4 與右邊是否碰撞
                eta=(Dwidthx-Dpi(1))-Drd
                   IF (eta < 0.0) THEN
                           Dfi(1) = Dfi(1)+Doomew*eta-Dbetaw*Dvi(1)
                           Dfi(2) = Dfi(2)-Dmiuk* (Dvi(2)-Dvw)
                   END IF
! #######################################################################
 ibx=Dbox(i)
 ib(1)=ibx;ib(2)=ibx+1 ;ib(3)=ibx+(grx-1);ib(4)=ibx+grx;ib(5)=ibx+(grx+1)
 ib(6)=ibx-1 ;ib(7)=ibx-(grx-1);ib(8)=ibx-grx;ib(9)=ibx-(grx+1)
        DO ik = 1, 9
        ibx = ib(ik)
        IF( ibx>0 .and. ibx < inbox+1 ) THEN
             ibxc = Dboxcount(ibx)
             IF(ibxc >0) THEN
                  DO ii = 1,ibxc
                      j = Dipd(ii,ibx)
                      IF(j.ne.i) THEN
                            relp(1)=Dpi(1)-Dpos(1,j);relp(2)=Dpi(2)-Dpos(2,j)
                            dis = relp(1)*relp(1)+relp(2)*relp(2)
                            dd = dSQRT(dis)
                            eta = dd - 1.0
                            IF (eta < 0.0) THRN
                                 vn(1)=Dvi(1)-Dvel(1,j);vn(2)=Dvi(2)-Dvel(2,j)
                                 normV(1)=relp(1)/dd ;normV(2)=relp(2)/dd
                                 tangV(1)=-normV(2) ;tangV(2)=normV(1)
                                 vndotnormV = vn(1)*normV(1)+vn(2)*normV(2)
                                 vndottangV = vn(1)*tangV(1)+vn(2)*tangV(2)
                                 FosN=-Doomep*eta-Dbetap*vndotnormV
                                 friction = -min(Dmius*abs(FosN),Dmiuk*abs(vndottangV))
                                 Dfi(1) = Dfi(1)+FosN*normV(1)+friction*tangV(1)
                                 Dfi(2) = Dfi(2)+FosN*normV(2)+friction*tangV(2)
                            END IF
                      END IF
                  END DO
             END IF
        END IF
        END DO
       Dfos(1,i) =Dfi(1) ; Dfos(2,i)=Dfi(2)
```

```
  END SUBROUTINE
END MODULE gpu
```

然後是我們的主程式

```
PROGRAM Parallel_2dMDS
USE cudafor
USE Global
USE gpu
IMPLICIT REAL*4(a-h,k-z),INTEGER(i,j)

idevice = 0
istat   = cudaSetDevice(idevice)  !啟動具有cuda計算能力的裝置，並編號為0
CALL setconstant(icycle)

!把Host上的參數都複製到Device上
Ddtsq = dtsq ;Ddt2 = dt2
Dbetap = 2.0*betap ; Dbetaw = 2.0*betaw; Doomep = 0.5*m*oomep; Doomew = m*oomew
Dmius =mius ; Dmiuk =miuk
Dnormg = normg ; DA = A
Dwidthx =widthx; DHeight = Height
Dm = m; Drd =rd
!決定Host上變數陣列的大小
ALLOCATE(pos(2,ipn),vel(2,ipn),fos(2,ipn),opos(2,ipn))
!決定Device上變數陣列的大小
Allocate(Dpos(2,ipn),Dvel(2,ipn),Dfos(2,ipn),Dopos(2,ipn))
Allocate(Dboxcount(inbox),Dbox(ipn))
Allocate(Dipd(5,inbox))
OPEN (unit=14,file='pla.txt')
OPEN (unit=15,file='pos.txt')
OPEN (unit=16,file='vel.txt')

CALL initiate ()   ! 呼叫initiate設定初始位置與速度
!把Host上的陣列的初始值都複製到Device上
Dopos = opos(1:2,1:ipn)
Dpos = pos(1:2,1:ipn)
Dvel = vel(1:2,1:ipn)
Dfos = fos(1:2,1:ipn)
Dbox = 0; Dipd=0
```

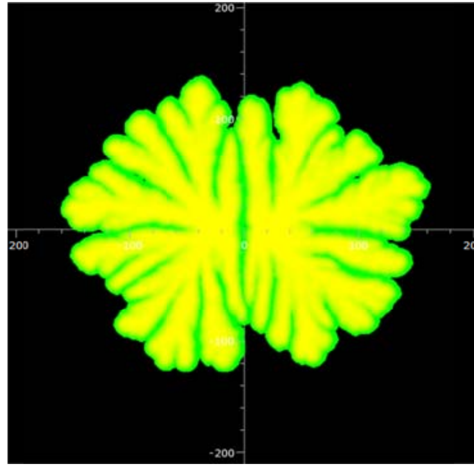
```
t=0.0
     DO it=1,icycle     !!!!!!!!!!!! 開始時間迴圈 !!!!!!!!!!!!

        Dboxcount=0  !歸零
        CALL neighborlist<<<NB,NT>>>(NT,igrid(1))!呼叫Device上的 neighborlist 積分後順便建立鄰居資料
        istat = cudaThreadSynchronize()   ! Cuda fortran的api 表示等待每個Thread都做完工作後才往下
        t = t + dt         !時間前進一步
        vw = A * dSIN(t)  !容器t時速度
        pw = -A * dCOS(t) !容器t時位置
        pwh = pw+Height   !頂盤t時位置
        CALL collision<<<NB,NT>>>(inbox,NT,igrid(1),pw,pwh,vw) !呼叫Device上的collision計算碰撞力
        istat = cudaThreadSynchronize(); ! 等待每個Thread都做完工作後才往下
        it1 = MOD(it,idraw)
        IF(it1.eq.0) THEN
             pos(1:2,1:ipn) = Dpos !每idraw步，才去把Device上的數據複製回Host上
             vel(1:2,1:ipn) = Dvel
             CALL save_data(t)      !然後儲存起來
        END IF
     END DO  ! END OF time COMPUTATION
     close(14);close(15);close(16)
END PROGRAM
```

到此我們已經把第一章的二維顆粒系統程式平行化了。剩下兩個副程式initiate和save並不需要更動。但要注意的是，PGI fortran目前沒有專門的繪圖模組，所以第一章的Drawing副程式在此不能使用，所有使用到ifqwin模組的繪圖指令記得要把它刪除。但因為我們已經有了顆粒每個時刻的位置，如果有需要，我們可以用另外的程式來做動畫。

下一章我們要介紹一個更實用的平行化例子。

第四章 ： 二維偏微分聯立方程組的平行化

我們要解一組反應擴散方程，描述二維平面上，兩群菌落的生長，在適當的參數下，你會發現兩群菌落之間會形成一道鴻溝，互不相交。



圖五：模擬結果以Vpython繪圖呈現

我們採用的模型方程如下:

$$\frac{\partial b}{\partial t} = d\nabla\left(bn\nabla b\right) + \frac{nb}{1+\gamma n}$$

$$\frac{\partial n}{\partial t} = \nabla^2 n - \frac{nb}{1+\gamma n} \qquad (4\text{-}1)$$

b和n分別為細菌數和養分濃度，它們都為空間(x,y)和時間t的函數。方程式中有兩個可調參數d和γ。方程式已經無因次化，其他細節請參考文獻(1)。我們採用四方晶格的解法，即把空間分成等間格格點，每一個格點上都遵照方程()隨時間演化，以Euler法求得。而空間的微分用下列差分方程近似

$$\frac{\partial b}{\partial x} \to \frac{b_{i+1,j} - b_{i-1,j}}{h}, \qquad \frac{\partial b}{\partial y} \to \frac{b_{i,j+1} - b_{i,j-1}}{h}$$

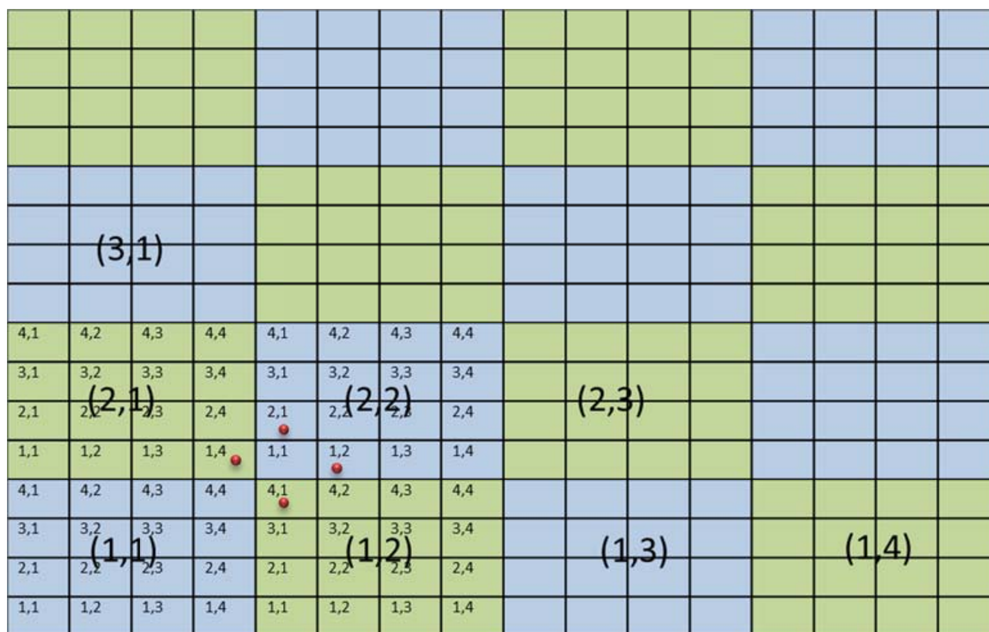
$$\frac{\partial^2 b}{\partial^2 x} \to \frac{b_{i+1,j} + b_{i-1,j} - 2b_{ij}}{2h}, \qquad \frac{\partial^2 b}{\partial^2 y} \to \frac{b_{i,j+1} + b_{i,j-1} - 2b_{ij}}{2h} \qquad (4\text{-}2)$$

我們已經把空間座標(x,y)換成晶格點的整數下標(i,j)，而h就是晶格點之間的基本單位。所以現在應該很簡單了，我們只要把每一個晶格點上需要解的方程式交給GPU去做就可以了，這是一個很適合用平行化來解的問題。讀者完全可以仿造第三章的例子去寫。但是就如前面所說，全域資料的讀取是很費時的，因此在本章將介紹一個使用shared memory的技巧讓平行程式更有效率，當然，付出的代價就是程式結構會變得比較複雜。

假設我們將空間分成512x512個格點，表示每一時刻，我們需要在Device的全域記憶體上開設262144個位址已存放b的值。當第(i,j)個執行緒在執行時就要到全域記憶體上的

(i,j),(i+1,j),(i-1,j),(i,j-1),(i,j+1)的位址讀取b的資料已做計算，做完之後再把結果存回適當的位址。這個過程很耗時間。如果我們可以把這些重複需要用到資料事先放到block裡面的shared memory上，這樣這個block中的thread就不需要返回到全域記憶體中去讀取資料，就能省下更多時間。然而因技術上的問題，目前每個block中只有48kb的記憶體容量，因此沒有辦法將262144個8bytes的數字(大約8Mb)全部放到一個block中，但適當的使用這48kb的shared memory絕對還是提升平行化效率的關鍵。

我們的策略是這樣的。以16x16格點為例子，我們將它分為4x4個blocks，每個block中有4x4個threads。如下圖。



圖六: 如何將空間格點對應到GPU上區塊與執行緒的示意圖

如此一來，當我們在計算block(1,1)中的threads之前，就可以先把這16個threads要用到的資料先一次過從全域記憶體中取出來放到這個block中的shared memory中，這樣就不必每次要用到時都到全域記憶體中取資料。因為只需有16*8bytes，對48kb而言是綽綽有餘。然而這還不是故事的全部，因為在每個block的邊界上，例如block(2,2)的第(1,1)個thread在做空間微分時需要用到block(1,2)，thread(4,1)和block(2,1)，thread(1,4)的資料(圖六中紅點區域)，但不同block中shared memory的資料不能共用，當然你還是可以到全域記憶體中去取資料，不過更好的方式是我們可以讓每一個block中有6x6個threads，而不是4x4，把相鄰block邊界的資料也一起抓進到shared memory中，只要總數不超48kb就好。這就是我們使用的技巧。

現在讓我們開始來寫這個程式。

一樣的，我們先寫一個模組宣告Device上需要的變數和執行的工作。

```
MODULE gpu
USE cudafor !使用cuda fortran api
    REAL*4,DEVICE,ALLOCATABLE,DIMENSION(:,:):: Db,Dn,Dd !宣告Device上的b,n,d陣列。
                                                          !b:細菌濃度，n養分濃度，d是擴散係數。
```

```
CONTAINS

    ATTRIBUTES(global) SUBROUTINE solve(h2,hsq,dt,g,NT) !執行平行運算的副程式
    INTEGER,VALUE :: NT
    REAL*4,VALUE :: h2,hsq,dt,g
    REAL*4 :: bf,nf,Lpb,Lpn,grad_bx,grad_by,grad_nx,grad_ny,reaction
    !bf和nf為等式(4-1)右邊的部分，Lpb為b的laplacian，grad_bx為b對x的一次微分，reaction為反應項。
    INTEGER :: i, j, tx, ty
    REAL*4,SHARED :: b_sh(32,32),n_sh(32,32)  !宣告每個區塊上用來暫存b和n的shared memory

    tx = threadidx%x  ! 令執行緒x的指標為tx
    ty = threadidx%y  ! 令執行緒y的指標為ty

    i = (blockidx%x-1) * (NT-2) + tx  ! 第blockidx%x區塊上，第tx個執行緒對應第i網格點
    j = (blockidx%y-1) * (NT-2) + ty  ! 第blockidx%y區塊上，第ty個執行緒對應第j網格點

    b_sh(tx,ty) = Db(i,j)  ! 把(i,j)網格上的b值從Device的全域區搬到第blockidx%x的共享區
    n_sh(tx,ty) = Dn(i,j)  ! 把(i,j)網格上的n值從Device的全域區搬到第blockidx%x的共享區

    CALL syncthreads()   ! 等待編號為blockidx%x的block中所有threads都執行到這裡時才繼續往下

    IF ((tx>1.and.tx<NT).and.(ty>1.and.ty<NT) ) THEN !除了邊界的網格外，要做以下計算
        grad_bx= (b_sh(tx+1,ty)-b_sh(tx-1,ty))/h2
        grad_by= (b_sh(tx,ty+1)-b_sh(tx,ty-1))/h2
        grad_nx= (n_sh(tx+1,ty)-n_sh(tx-1,ty))/h2
        grad_ny= (n_sh(tx,ty+1)-n_sh(tx,ty-1))/h2

        Lpb= ( b_sh(tx+1,ty) + b_sh(tx-1,ty) + b_sh(tx,ty+1) + b_sh(tx,ty-1) - 4.0*b_sh(tx,ty)) / hsq
        Lpn= ( n_sh(tx+1,ty) + n_sh(tx-1,ty) + n_sh(tx,ty+1) + n_sh(tx,ty-1) - 4.0*n_sh(tx,ty)) / hsq

        reaction = n_sh(tx,ty)*b_sh(tx,ty)/(1.0+g*n_sh(tx,ty))

        bf = Dd(i,j)*( b_sh(tx,ty)*n_sh(tx,ty)*Lpb + b_sh(tx,ty)*(grad_bx*grad_nx+grad_by*grad_ny)&
              +n_sh(tx,ty)*(grad_bx*grad_bx+grad_by*grad_by) ) +reaction
        nf = Lpn - reaction
   !注意，從IF開始到此的計算要用到很多次b和n的值，但這些值已經暫存到共享記憶體中，因此省了很多到全域記
   !憶體中讀取資料的時間，這就是shared memory的用意。
        Db(i,j) = b_sh(tx,ty) + bf * dt
        Dn(i,j) = n_sh(tx,ty) + nf * dt   ! 最後才把新的資料放回到全域記憶體中
```

```
        END IF
    END SUBROUTINE solve
END MODULE
```

接下來是主程式的部分。要注意的事是，上一章的例子，我們是把區塊和執行緒排成一維陣列來使用，而這裡顯示，我們也可以把它們排成二維陣列來使用，所以我們會使用CUDA內定的型態TYPE(dim3) dimGrid,dimBlock 定義區塊成NB X NB的陣列，而每一區塊中的執行緒排成 NT X NT。

```
PROGRAM Bact_Ecoli
USE cudafor    !使用cuda fortran api
USE gpu
REAL*4,ALLOCATABLE,DIMENSION(:,:) :: Hb,Hn  !宣告Host上的b和n陣列，在Host端用不到擴散係數
REAL*4 h,h2,hsq,dt,g  !h:空間間隔，h2:兩倍的h值，hsq:h的平方值，dt:積分的時間間隔，g:參數gamma值
REAL*4 rn,time_begin,time_end !rn是亂數，time_begin和time_end只是用來計算程式花的時間，可有可無
INTEGER i,j,ii,grid,istep ! grid是每一邊網格點的數目，istep是時間迴圈的總數
INTEGER NB,NT ! NB是每一邊使用的區塊數，NT是每一區塊中每一邊執行緒的數目
TYPE(dim3) dimGrid,dimBlock !Device上grid和block的維度
idevice = 0
istate = cudaSetDevice(idevice) !啟動有CUDA計算能的的裝置

OPEN(11,file='ini512.txt')
OPEN(12,file='bact512.txt')
OPEN(14,file='time512.txt')
NB = 17  !準備把空間分成 17 x 17的區塊
NT = 32  !而每一區塊中都有 32 x 32的執行緒

dimGrid  = dim3(NB,NB,1)  ! 表示Device上Grid的維度是 NB x NB 個區塊
dimBlock = dim3(NT,NT,1)  ! 表示Device上Block的維度是 NT x NT 個執行緒

grid = 512 ! 如內容所描述，每個區塊的邊界會重疊，
           !所以間中實際的網格點數會是  (NB -1)*NT x (NB -1)*NT = 512 * 512
           !注意:這裡的grid不是指Device的Grid。
ALLOCATE (Hb(grid,grid),Hn(grid,grid))    !決定Host變數的維度
ALLOCATE (Db(grid,grid),Dn(grid,grid),Dd(grid,grid)) !決定Device變數的維度

h = 0.5 ;  h2 = 2.0*h ; hsq = h*h
dt = 0.005 ; g = 0.5

CALL random_seed()
! 設定格點上的初值
```

```
DO i=2,grid-1,2
     DO j=2,grid-1,2
          Hb(i,j) = 0.0          !  Host端的細菌濃度先都設為零
          CALL random_number(rn)
          Hn(i,j) = 1.0 + 0.1*(2.0*rn-1.0)   !  亂數決定養分濃度，存放在Host端
          CALL random_number(rn)
          Dd(i,j)=30*0.025*(1.0+0.4*(2.0*rn-1.0))   !亂數決定格點四周的擴散係數，存放在Device端
          CALL random_number(rn)
          Dd(i,j+1)=30*0.025*(1.0+0.4*(2.0*rn-1.0))
          CALL random_number(rn)
          Dd(i+1,j)=30*0.025*(1.0+0.4*(2.0*rn-1.0))
          CALL random_number(rn)
          Dd(i+1,j+1)=30*0.025*(1.0+0.4*(2.0*rn-1.0))
    END DO
END DO

!  在空間中央左右兩側的格點上加入等量的細菌
DO i=96,112
   DO j = 120,136
       Hb(i,j) = 0.7
   END DO
END DO

DO i=144,160
   DO j = 120,136
       Hb(i,j) = 0.7
   END DO
END DO

!設定邊界條件，為No flux條件，即微分等於零
DO i=1,grid
   Hb(i,1) = Hb(i,3)
   Hb(i,grid) = Hb(i,grid-2)
   Hb(1,i) = Hb(3,i)
   Hb(grid,i) = Hb(grid-2,i)

   Hn(i,1) = Hn(i,3)
   Hn(i,grid) = Hn(i,grid-2)
   Hn(1,i) = Hn(3,i)
   Hn(grid,i) = Hn(grid-2,i)
```

```
END DO

Db = Hb    !  把host端上的細菌濃度複製到Device上
Dn = Hn    !  把host端上的養分濃度複製到Device上

istep =40000
CALL CPU_TIME(time_begin)
DO it = 1, istep    !時間迴圈開始

   CALL solve<<<dimGrid,dimBlock>>> (h2,hsq,dt,g,NT)   !呼叫Device上的solve副程式
   ii = cudathreadsynchronize()  !  等待所有執行緒都做完工作才往下執行

   IF(mod(it,500)==0) THEN  !  每５００步  紀錄數據一次
     Hb = Db    !記得要先從Device上把要的數據複製回Host端記憶體上
      DO i=1,grid
        DO j=1,grid
          bb = Hb(i,j)
          IF(bb > 0.005)  WRITE(12,*) i,j,bb      !儲存(i,j)格點上的細菌濃度值
       END DO
     END DO
   END IF

END DO    !時間迴圈結束

CALL CPU_TIME(time_end)
WRITE(14,*) time_end-time_begin

END PROGRAM
```

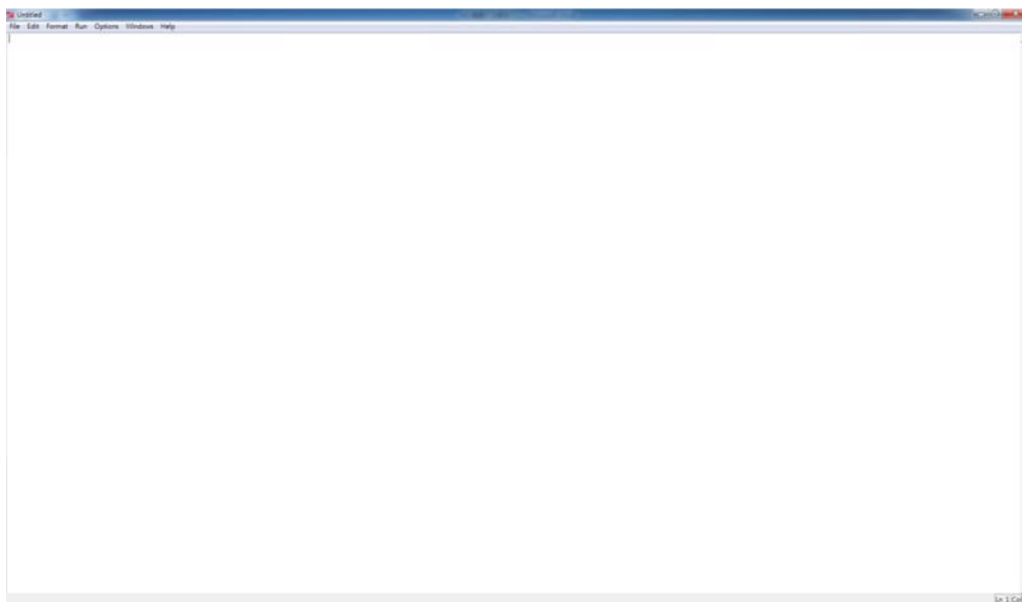
第五章: VPython 動畫編程介紹

Intel fortran雖然附有繪圖模組ifqwin，但只適用於沒有宣染效果的平面繪圖，使用上雖然簡單但卻不很人性化。這一章我們將簡單的介紹以Python語言為基礎的繪圖模組，將前面fortran計算出來的數據變成動畫。

首先，電腦必須安裝以下軟體，都是免費開放軟體，用google搜尋就能找到。

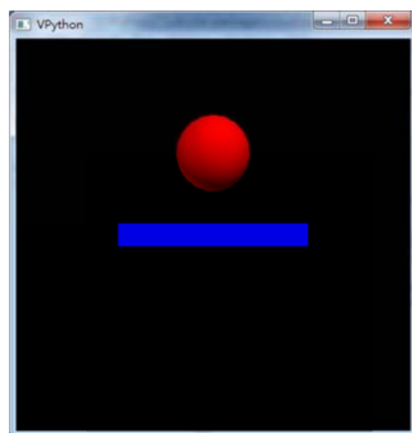
1. python 2.7
2. numpy
3. Vpython

安裝完畢後，可點選桌面上Vpython 的捷徑開啟Vpython 介面如下。



你可以從File的open中看見許多現成的範例，例如第一個範例是bounce.py

```
from visual import *
floor = box(length=4, height=0.5, width=4, color=color.blue)
ball = sphere(pos=(0,4,0), color=color.red)
ball.velocity = vector(0,-1,0)
dt = 0.01
while 1:
    rate(100)
    ball.pos = ball.pos + ball.velocity*dt
    if ball.y < 1:
        ball.velocity.y = -ball.velocity.y
    else:
        ball.velocity.y = ball.velocity.y - 9.8*dt
```



按下F5就可以看見如右圖的動畫結果。

橘色文字是Python指令。第一行是使用vpython的visual模組，然後畫一個盒子box命名為floor。這個盒子的長度，高度和寬度分別為length=4,height=0.5和width=4，顏色是藍色color=color.blue。

接下來畫一顆球sphere並命名為ball，這球的圓心在pos=(0,4,0)位置，顏色是紅色color=color.red。球的速度是一個向量ball.velocity=vector(0,-1,0)。dt是時間間隔。然後就進入動畫迴圈。Rate是畫面更新的頻率，數字越小更新的越慢。接著是球的位置更新，就是原來的位置加上速度乘以一小段的時間間隔。條件判斷，若球的y座標小於1，則球的速度反向，否則求的速度就因重力加速度而做改變。這樣短短的12行指令，就可以做出立體的動畫，所以我非常推薦使用Vpython來呈現fortran計算後的結果。那為何不直接在python上做計算?python畢竟是個直譯式高階語言，對一些複雜的科研計算還是比較慢的。除非是使用pycuda，即python的平行運算。

接下來我們看一個例子，如何將2dMDS程式輸出的數據以Vpython做成動畫。

```
from visual import *
posfile   = open('pos.txt','r')     #開啟文件pos.txt，命名為posfile
platefile = open('pla.txt','r')
inputfile = open('input.txt','r')
inpp=inputfile.readline()         #讀取inputfile的一行資料，存放在inpp
inp=inpp.split()                  #由於讀取是一整行的，所以再將一整行的資料以空格分開，存在inp陣列中

ballnum =int(inp[0])              #已知input.txt中，第一行第一個數字是粒子數目。Int()是整數的意思

widthx = int(inp[1])              #第二個數字是容器寬度
heighty = int(inp[2])             #容器高度
widthz = 2.0                      #二維模擬沒有厚度，所以自己指定

scene = display(title="Vibrating grains", width=800, height=400, x=0, y=0,
                center=(0.5*widthx,0.5*heighty,15),forward=(0,0,-1), background=(1,1,1))
#用display開啟一個繪圖視窗，命名為scene。他的屬性有標題title，寬高width,height，左上角座標x,y，攝影機中
#心center，攝影機方向forward及背景顏色background。

ppl=platefile.readline()
pl=ppl.split()
ctnx = 0.5*widthx ; ctny=0.5*heighty+ float(pl[1]); ctnz=0.0   #容器中心位置座標
container = box (pos=( ctnx,ctny ,ctnz ), size=(widthx,heighty,widthz),   color = color.red, opacity=0.5)
#劃一個盒子命名為container，其中的屬性有，盒子的中心位置pos，大小size，顏色color，透明度opacity
ps=[]   #設ps為一個空list
for i in range(ballnum):        #迴圈i從0到ballnum-1
    ppb = posfile.readline()   #讀取posfile中的一行資料
    pb = ppb.split()           #以空格把資料作分隔放到pb陣列中，每一筆資料預設為字串型態
    x = float(pb[0]) ; y = float(pb[1]);     z = 0   #  資料的第一行是粒子的x座標，第二行是y座標，z自設為零。
    ps = ps+[(x,y,z)]   #把第i顆粒子的空間座標(x,y,z)存放到ps中。
pset=points(pos=ps,size=1,color=color.red)   #把所有ps的座標點以點的方式畫出來，並把這些點集命名為pset
```

```
# def  是python中副程式的意思，這裡寫了個暫停的副程式命名為pause。
def pause():
    while True:
        rate(10)      #更新速率
        if scene.mouse.events:      #當視窗偵測到滑鼠事件時
            m = scene.mouse.getevent()    #記下這個滑鼠事件，持續迴圈
        elif scene.kb.keys:          #當視窗偵測到鍵盤事件時
            k = scene.kb.getkey()   #記下這個鍵盤事件
            return                  #返回主程式
while True:
  if scene.mouse.clicked:       #當視窗偵測到滑鼠按下事件
      pause()                   #呼叫pause副程式
  rate(50)           #畫面更新速率
  ppl=platefile.readline()    #讀取容器位置的資料
  pl=ppl.split()
  ctny= float (pl[1])+0.5*heighty
  container.pos=(ctnx,ctny,ctnz)     #  更新容器的位置
  ps = []
  for i in range(ballnum):
    ppb = posfile.readline()   #讀取粒子的位置資料
    pb = ppb.split()
    x = float (pb[0]);       y = float (pb[1]);      z = 0
    ps.append((x,y,z))
  pset.pos=ps     #  更新點集pset的位置
```

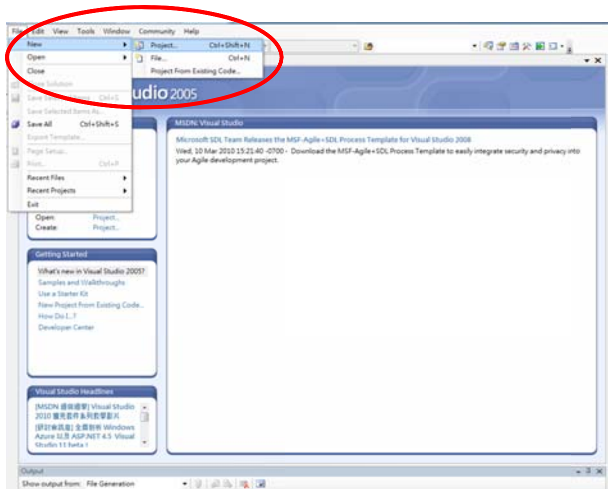
附錄A: INTEL Fortran的安裝與設定

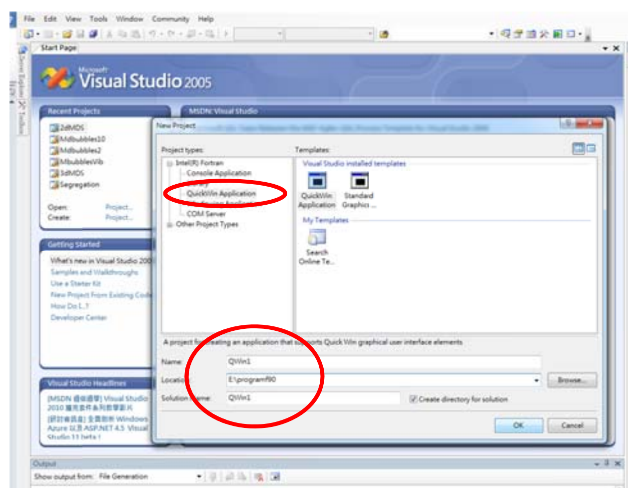
取得光碟按照一般程序安裝。Intel fortran 10.01版在win7上有相容性問題，還需要下載VS SP1(第一次使用時電腦會提示)，安裝後才能正常使用Microsoft Visual Studio 2005編譯。安裝好後就可以到 ”開始” 程式集中開啟Microsoft Visual Studio 2005。按照以下步驟建立專案

(project)。

1. File ->New->project

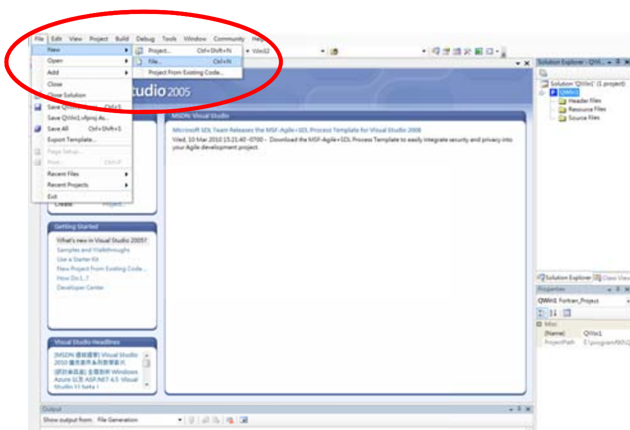
2.選Quickwin Application，和給定專案名稱。





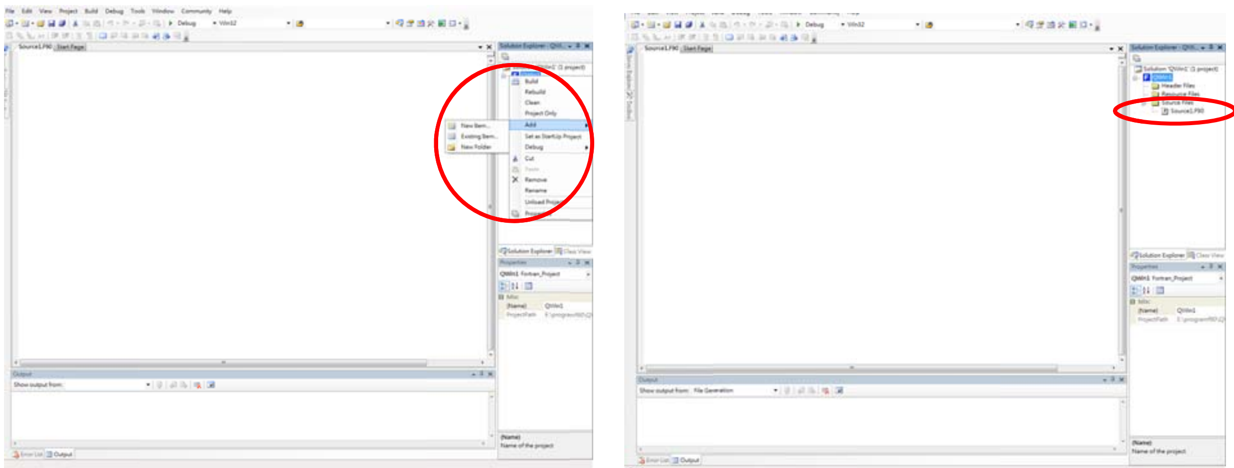
3.再到File ->New->File。

4.選Intel Fortran

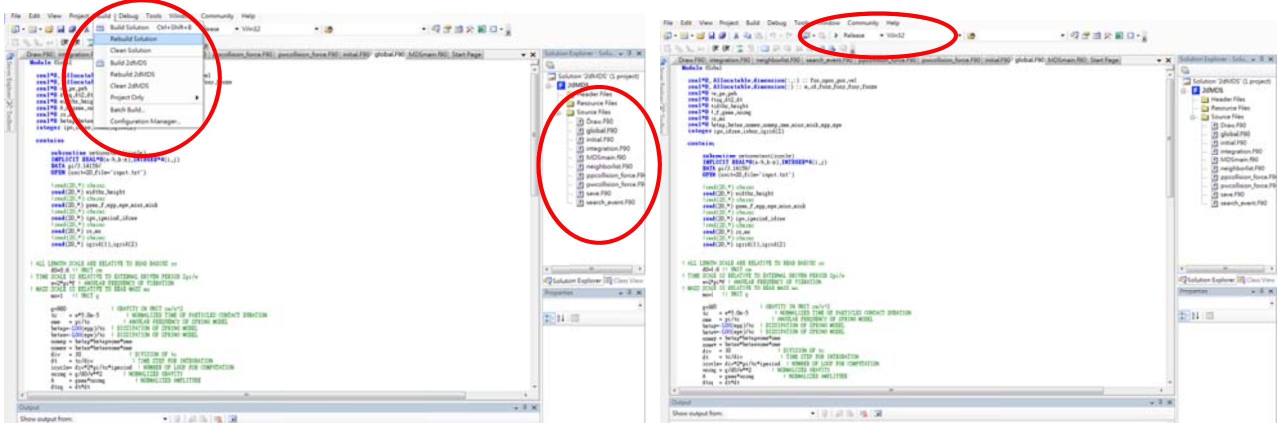


5.這時程是會自動開啟一個source.f90的空白臨時檔案，必須再到file案save file as..將檔案命名儲存到project 的資料夾下。

6.然後再到右手邊(有時是在左手邊)的solution explorer中以右鍵點選project，選add加入剛剛存的檔案就可以開始寫程式了。



如果是有已經寫好的程式，像是本文提供的範例，將第一章的範例程式資料夾中所有檔案複製到專案的資料夾下，在visual studio 下把所有.f90檔案加到source files下，到上方Build中選取rebuild solution。如果下方沒有錯誤顯示，就可以按執行按鈕開始執行。若一切正常可以切換到release模式，這樣會讓程式執行速度加快。



附錄B: PGI Fortran的安裝與設定

首先你的電腦當然需要可以執行CUDA的硬體，目前Nvidia Geforce系列都配有Cuda核心。接下來需要準備以下軟體:

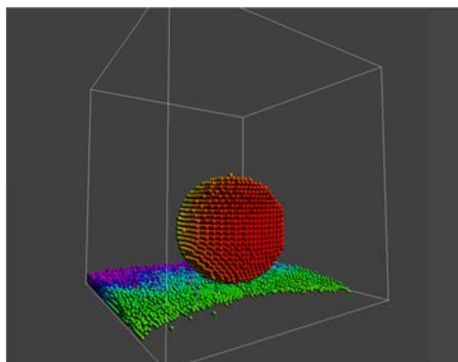
1. PGI Visual Fortran(PVF) 2010 11.1 以上版本
2. Microsoft Visual Studio 2010 (一般上已附在PGI Visual Fortran中，但也可以另外安裝)
3. Cuda toolkits 4.0 64bit (到nvida網站下載http://developer.nvidia.com/cuda-downloads)
4. dedriver 4.0 64bit(一樣到nvida網站下載)
5. gpucomputing sdk 4.0 64bit(一樣到nvida網站下載)

由於些資訊會不斷在更新，請自行留意自己需要的版本。目前我使用的系統是配合Win 7 64bit intel i7 4cores, 8G記憶體。而事實上PVF 11.1版本已附有CUDA toolkits 3.2 ，PVF11.4版才支援

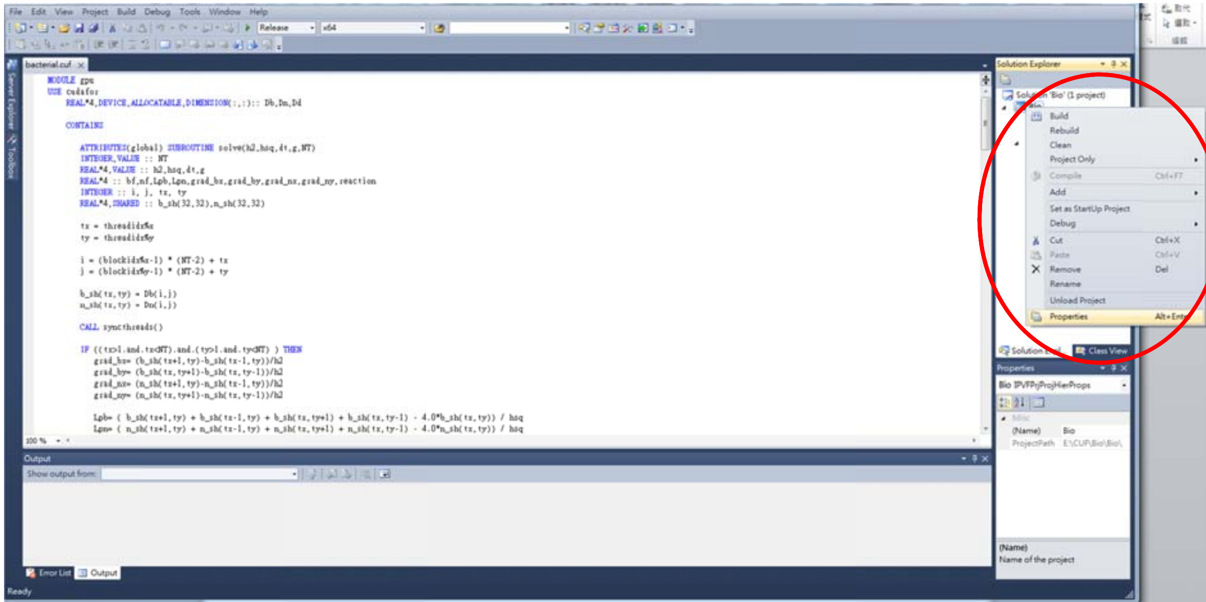
CUDA4.0的一些新功能。按照以上順序安裝之後可以先打開 GPU Computing SDK，裡面有許多用C語言寫好的範例，找一個叫做 Device Query的範例執行，看看你的安裝是否成功。如果成功，他就會顯示GPU的基本訊息，如記憶體大小，網格，區塊和執行緒的數目等等，如下圖。

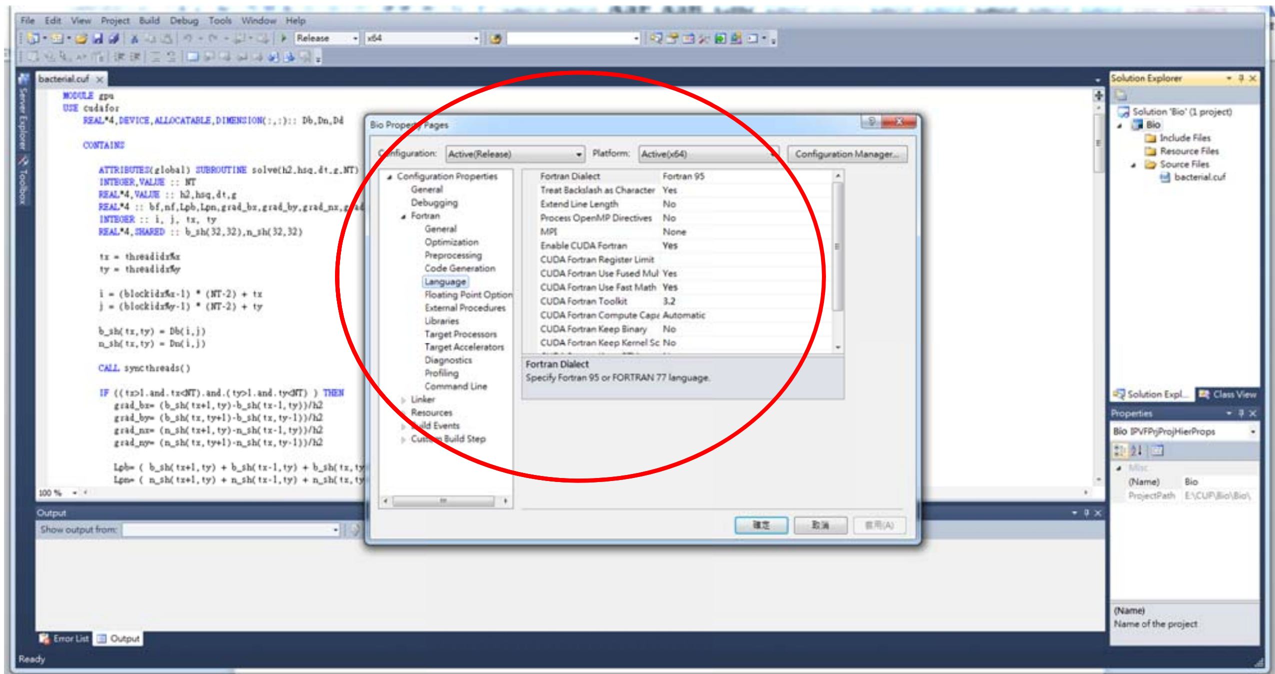
```
C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.0\C\bin\win64\Release...
[deviceQuery.exe] starting...
C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.0\C\bin\win64\Relea
se\deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "GeForce GTX 465"
  CUDA Driver Version / Runtime Version          4.0 / 4.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 993 MBytes (1041694720 bytes)
  (11) Multiprocessors x (32) CUDA Cores/MP:     352 CUDA Cores
  GPU Clock Speed:                               1.22 GHz
  Memory Clock rate:                             1603.00 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3
D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16
384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simu
ltaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.0, CUDA Runtime Versi
on = 4.0, NumDevs = 1, Device = GeForce GTX 465
[deviceQuery.exe] test results...
PASSED

Press ENTER to exit...
```

接下來你可以再找一個Particles的範例執行。這是一個用C和OPENGL寫的MD模擬。裡面的範例都附有程式碼和一些簡單的解說，有興趣C語言的人可以參考。



如果上面的步驟都已成功，那就可以打開PVF建立一個64位元的Project和開啟一個.CUF的空白檔案。再到Project的Property中選取fortran清單下的Language，其中有一個Enable CUDA Fortran的選項，把它設成yes就可以編譯含有CUDA API的程式碼了。


File Edit View Project Build Debug Tools Window Help
Release
x64
bacterial.cuf
MODULE gpu
USE cudafor
REAL*4,DEVICE,ALLOCATABLE,DIMENSION(:,:):: Db,Dn,Dd
CONTAINS
ATTRIBUTES(global) SUBROUTINE solve(h2,hsq,dt,g,NT)
INTEGER,VALUE :: NT
REAL*4,VALUE :: h2,hsq,dt,g
INTEGER :: i, j, tx, ty
REAL*4,SHARED :: b_sh(32,32),n_sh(32,32)
tx = threadidx%x
ty = threadidx%y
CALL syncthreads()
Solution Explorer
Build
Rebuild
Clean
Project Only
Compile
Add
Set as StartUp Project
Debug
Cut
Paste
Remove
Rename
Unload Project
Properties
Output
Show output from:
Error List
Output
Ready


File Edit View Project Build Debug Tools Window Help
Release
x64
bacterial.cuf
Bio Property Pages
Configuration: Active(Release)
Platform: Active(x64)
Configuration Manager...
Configuration Properties
General
Debugging
Fortran
General
Optimization
Preprocessing
Code Generation
Language
Floating Point Options
External Procedures
Libraries
Target Processors
Target Accelerators
Diagnostics
Profiling
Command Line
Linker
Resources
Build Events
Custom Build Step
Fortran Dialect Fortran 95
Treat Backslash as Character Yes
Extend Line Length No
Process OpenMP Directives No
MPI None
Enable CUDA Fortran Yes
CUDA Fortran Register Limit
CUDA Fortran Use Fused Mul Yes
CUDA Fortran Use Fast Math Yes
CUDA Fortran Toolkit 3.2
CUDA Fortran Compute Capa Automatic
CUDA Fortran Keep Binary No
CUDA Fortran Keep Kernel Sc No
Fortran Dialect
Specify Fortran 95 or FORTRAN 77 language.
Solution Explorer
Solution 'Bio' (1 project)
Bio
Include Files
Resource Files
Source Files
bacterial.cuf
Properties
(Name) Bio
Name of the project
Output
Show output from:
Error List
Output
Ready

附錄C 串行和平行程式效率的比較

使用硬體 CPU: Intel i3 2.93Ghz

RAM: 2.0GB

OS: 64bits Win 7

GPU: NVIDIA GTX 465 CUDA 4.0

模擬100震動週期

	2dMDS(f90)	2dMDSParallel(cuf)	ratio
N = 2048	275s	123s	2.2
N = 4096	571s	150s	3.8

模擬時間: 70000個dt，模擬尺寸: 512 x 512格子

Bacterial.f90	Bacterial.cuf	ratio
687s	16s	43

參考資料:

1. Computer Simulation of Liquids. M.P. Allen and D.J. Tildesley. Oxford University Press.
2. Sands, Powders, and Grains-An introduction to the physics of Granular Materials. Jacques Duran, Springer.
3. Horizontal segregation of mono-layer granules coordinated by vertical motion. S.-Y. Liaw, F.F. Chung, and S.-S. Liaw. Eur.Phys. J. E34(2011).
4. GPU 高效能運算之CUDA。作者:張舒，諸豔利，趙開勇，張鈺勃。中國水利水電出版社。
5. Vpython線上參考: http://vpython.org/contents/docs/visual/index.html
6. Pgi cuda fortran: http://www.pgroup.com/
7. Nvidia Cuda Zone: http://developer.nvidia.com/category/zone/cuda-zone