Introducción a Parallel toolbox de Matlab

Víctor M. García

Contenidos

- Descripción general de Parallel Toolbox: Preliminares
- Parfor
- SPMD
- pmode
- Computación en GPU

Computación en GPU desde MATLAB

Es posible utilizar Graphics Processing Units desde MATLAB, si se tiene disponible el Parallel toolbox.

-Tiene que ser GPUs NVIDIA, preferiblemente recientes.

-Hay que instalar un driver reciente (ultimo) de Nvidia para la tarjeta gráfica.

-Idealmente, para trabajar con Windows y dedicar la tarjeta gráfica sólo a cálculo, deberíamos tener dos tarjetas gráficas. Sin embargo, en la actualidad es complicado configurar un sistema así.

-Bastante más sencillo en Linux. Podemos usar los PCs del laboratorio o conectarnos a gpu.dsic.upv.es (o a knights, aunque da problemas)

Computación en GPU desde MATLAB

Para averiguar cuantas GPUs tenemos disponibles:

```
>> gpuDeviceCount
```

Para averiguar sus propiedades:

```
>> gpuDevice
```

O, si tenemos 2:

```
>>gpuDevice(1)
```

```
>>gpuDevice(2)
```

Computación en GPU desde MATLAB

Función gpuArray: Envía los datos a la GPU; lo que se haga con esos datos se hará en la GPU;

Restricción: Las matrices que se envíen deben ser DENSAS;
De momento, Matlab + GPU no trabajan con matrices dispersas.

Al acabar, se pueden recuperar los datos de la GPU con “gather”

Computación en GPU desde MATLAB: Posibilidad 1

Función gpuArray, combinado con funciones de MATLAB:

Ejemplo, calcular la factorización LU de una matriz densa

```
>> A=rand(5000);
>>tic, ,[l,u,p]=lu(A,'vector');toc
>> GA=gpuArray(A);
>> tic,[l,u,p]=lu(GA,'vector');toc
```

Ahora, repetimos con precisión simple:

```
>> A=rand(5000,'single');
>>tic, ,[l,u,p]=lu(A,'vector');toc
>> GA=gpuArray(A);
>> tic,[l,u,p]=lu(GA,'vector');toc
```

Ejercicio

Cálculo del número pi: Crea una nueva versión del programa para calcular pi por el método de Montecarlo, con gpuArray. A partir de compute_pi_Matlab (versión vectorizada)

Computación en GPU desde MATLAB: Posibilidad 2: Arrayfun

Arrayfun: es una forma “vectorizada” de llamar a funciones de Matlab, escritas por nosotros. Conceptualmente, es ejecutar muchas veces la misma función con diferentes datos.

Ejemplo: Función que devuelva una de las dos raíces de una ecuación de segundo grado:

```
function  x=miraiz (a,b,c)
  sol=roots([a,b,c]);
  x=sol(1);
end
```

Computación en GPU desde MATLAB: Arrayfun

A la función miraiz la podemos llamar (con números) de diferentes formas:

>>x=miraiz(1,2,3);

>>x=feval('miraiz',1,2,3)

>>x=feval(@miraiz,1,2,3)

En lugar de llamarla con números podemos pasarle vectores (todos de la misma dimensión) usando arrayfun:

Computación en GPU desde MATLAB: Arrayfun

Para ejecutar con Arrayfun

```
>>N=800;A=rand(N,1);B=rand(N,1);C=rand(N,1);
>>sol=arrayfun(@miraiz,A,B,C)
```

Arrayfun asume que las dimensiones de los arrays de entrada y de salida son las mismas; si la función "miraiz" devuelve un vector con las dos soluciones de la ecuación de segundo grado:

```
function  x=miraiz2 (a,b,c)
  x=roots([a,b,c]);
End
```

Entonces hay que llamarla con la opción UniformOutput a 0:

```
>>sol=arrayfun(@miraiz2,A,B,C,'UniformOutput',0)
```

(Ojo no va con gpus, por culpa de la función "roots")

Computación en GPU desde MATLAB: Arrayfun

Si los datos están en la CPU, arrayfun se ejecuta en la CPU;

Si los datos están en la GPU, arrayfun se ejecuta en la GPU

Esto permite extraer una gran parte del potencial de las GPUs

```
function [ x ] = miraizg( a,b,c )
x=(-b+sqrt(complex(b*b-4*a*c,0)))/(2*a)
end
```

```
>>Ag=gpuArray(A);
>>Bg=gpuArray(B);
>>Cg=gpuArray(C);
>>sol=arrayfun(@miraizg,Ag,Bg,Cg)
>>solgpu=gather(sol);
```

Computación en GPU desde MATLAB: Posibilidad 2

Ejercicio (3):

1) Crea una nueva versión del programa para calcular pi por el método de Montecarlo, usando arrayfun para calcular el vector sal, tal que sal(i)=1 si (x(i)^2+y(i)^2<1), y sal(i)=0 si (x(i)^2+y(i)^2>=1).

Partimos de compute_pi_for.m, pero quitando el bucle.

Computación en GPU desde MATLAB: Posibilidad 2

Arrayfun con matrices:
Arrayfun funciona también si tenemos un doble bucle que recorre una matriz

```
N=10; M=20;A=gpuArray.rand(N,M);B=gpuArray.rand(N,M);
C=gpuArray.rand(N,M);
for i=1:N
    for j=1:M
        sal(i,j)=miraizg(A(i,j),B(i,j),C(i,j));
     end
end
```

El doble bucle se puede sustituir por:

```
sal=arrayfun(@miraizg,A,B,C);
```

Computación en GPU desde MATLAB: Posibilidad 2

Caso especial: Arrayfun con matrices, accediendo dentro de la función a diferentes elementos de una matriz.

Arrayfun , en su forma básica, no es apropiado para acceder elementos de una matriz

```
function [ x ] = miraizg( a,b,c )
x=(-b+sqrt(complex(b*b-4*a*c,0)))/(2*a)
end
```

Las referencias a elementos de a,b,c,x deben ser escalares, sin subíndices. Así no podemos referenciar elementos de una matriz (por ejemplo, función difumina).

Computación en GPU desde MATLAB: Posibilidad 2

Es posible referenciar elementos de una matriz usando una técnica nueva, con dos detalles fundamentales:

1) Hay que usar “funciones anidadas” (Se pueden crear dentro de funciones, no dentro de scripts). Ejemplo: juego_vida_arrayfun.m

La variable **grid** (que es una matriz, de la cual queremos referenciar sus elementos) no se pasa como argumento a “Updateparent”, sino que es una variable de la función “juego_vida_arrayfun”, la cual contiene a “Updateparent”

Computación en GPU desde MATLAB: Posibilidad 2

2) Pasamos como argumentos vectores de índices.
La cabecera de la función updateparent:

function X = updateParent(row, col)

Dentro de updateparent, row funciona como índice de filas y col como índice de columnas.

X debe tener dimension M filas por N columnas. Entonces, en la llamada a updateParent,
-rows es un vector **columna** (dimension M por 1) con los índices de 1 a M (sería el vector columna con valores (1,2,3,...,M-1,M))
-cols es un vector fila (dimension 1 por N) con los índices de 1 a N (Igual que rows, pero como vector fila y de 1 a N)

Computación en GPU desde MATLAB: Posibilidad 2

Comparamos las opciones en arrayfun para ver las diferencias:

V1: Cuando llamamos a arrayfun pasándole vectores de la misma dimensión N , arrayfun realiza N llamadas a la función subyacente

V2: Cuando llamamos a arrayfun pasándole matrices de la misma dimensión M*N , arrayfun realiza M*N llamadas a la función subyacente (Como un doble bucle)
Pero no podemos acceder a los elementos de la matriz.

V3: Cuando llamamos a arrayfun pasándole un vector (rows en el ejemplo) **columna** de indices de fila (1…M) y un vector (columns en el ejemplo) **fila** de índices de columna(1..N), también hace M*N llamadas a la función subyacente. Se puede usar para acceder a los elementos de una matriz, pero la matriz debe ser “global”, no se puede pasar como argumento a la función subyacente.
También se pueden pasar variables “escalares” como argumentos de la función subyacente

Computación en GPU desde MATLAB: Posibilidad 2

Ejercicio: En Poliformat está la función difumina5, una versión simplificada de la función difumina que usamos en sesiones anteriores. Utiliza la técnica usada en juego_vida_arrayfun.m para ejecutar los cálculos en la GPU con arrayfun.

Vamos a sustituir los dos bucles mas internos por arrayfun, y dejamos el externo (ind=1:tam, tam es igual a 3) casi igual, debe quedar así:

```
for ind=1:tam
  imagen_out(:,:,i)=.... //llamada a arrayfun
End
```

Para cargar y visualizar la imagen, hacíamos esto:
matr=imread('ngc6543a.jpg'); imshow(matr);

Computación en GPU desde MATLAB: Posibilidad 3

Programar “kernels” usando CUDA + C, y llamarlo desde Matlab

Kernel CUDA para resolver una ecuación de segundo grado:

```
__global__ void ecuacion(double *solr, double * solim, const double *a, const double *b, const double *c )
{
 int id=threadIdx.x;
double tmp=b[id]*b[id]-4*a[id]*c[id];
 if (tmp>=0)
    {solr[id]=(-b[id]+sqrt(tmp))/(2*a[id]);
     solim[id]=0.0;
    }
else
  {solr[id]=(-b[id]/(2*a[id]));
   solim[id]=(sqrt(-tmp))/(2*a[id]);
  }
}
```

Observad el “const” para los argumentos de entrada

Computación en GPU desde MATLAB: Posibilidad 3

Para poder llamar al kernel desde MATLAB:

1) Guardar el kernel con extensión .cu; compilar con nvcc y flag –ptx:

nvcc –ptx ecuacion.cu (se genera ecuacion.ptx)

Se ponen los dos archivos (.cu y .ptx) en el directorio actual (o en un directorio accesible desde el path)

Computación en GPU desde MATLAB: Posibilidad 3

En MATLAB:

1) Enviar datos de entrada y de salida a GPU (Ag, Bg, Cg, solre, solim),Vectores de tamaño 800. Hay que dar memoria también a los argumentos de entrada
2) >>kern=parallel.gpu.CUDAKernel('ecuacion.ptx','ecuacion.cu')
3) Poner numero de “threads” que vamos a usar, uno por elemento de los vectores:
4) >>kern.ThreadBlockSize=800
5) Llamar al kernel con feval:
6) >>[solre,solim]=feval(kern,solg,solim,Ag,Bg,Cg)
7) Traer los resultados:gather(solre), gather(solim)

Computación en GPU desde MATLAB: Posibilidad 3

Si tenemos un tamaño mas grande que el tamaño de bloque:
N=5000 ➔
En el kernel, cambiar:

```
int id=threadIdx.x;
```

Por:

```
int id=threadIdx.x+blockIdx.x*blockDim.x;
if (id<N){...
        }
```

En el proceso de llamada, cambiar:

```
kern.ThreadBlockSize=800;
```

por

```
kern.ThreadBlockSize=512;
kern.GridSize=[ceil(N/512),1]
```

Computación en GPU desde MATLAB: Posibilidad 3

Ejercicio :

1) Crea una nueva versión del programa para calcular pi por el método de Montecarlo, usando un kernel cuda para calcular el vector sal, tal que sal(i)=1 si (x(i)^2+y(i)^2<1), y sal(i)=0 si (x(i)^2+y(i)^2>=1).

Computación en GPU desde MATLAB: Posibilidad 3

También es posible usar CUDA desde archivos Mex. Eso permitiría usar librerías como cublas, cufft, ...

https://es.mathworks.com/help/distcomp/run-mex-functions-containing-cuda-code.html

Ejemplo en Poliformat: mexGPUExample.cu