Adrià Muro Gómez (1665191), David Morillo Massagué (1666540)

Exercicis Pràctics OpenACC (Part II)

L'objectiu principal d'aquests exercicis és que experimenteu amb les capacitats d'OpenACC i CUDA fent servir casos simples, facilitant d'aquesta manera la transició entre els continguts discutits en les classes de teoria i la seva aplicació al cas pràctic (més complex) treballat en el laboratori.

El plantejament dels exercicis i la mecànica de treball per resoldre'ls consisteixen en:

1) Per cada apartat, es proporcionen un conjunt de fragments de codi que caldrà executar i analitzar (els programes corresponents els trobareu a /home/alumnos/pp//Avaluats-problemes/Cuda-OpenACC/sessio2/).
2) Per cada apartat, es fa un conjunt de preguntes sobre cada fragment de codi presentat. Heu de respondre cada pregunta, justificant sempre la vostra resposta. En alguns casos la justificació serà molt curta, en altres més complexa i, en altres, potser consistirà en un nou fragment de codi.
3) Juntament amb els codis a analitzar, trobareu un script (*job.sub*) per demanar que el gestor de cues (*SLURM*) enviï el nostre programa a ser executat en la màquina del laboratori on hi ha instal·lada l'acceleradora. Aquest script s'assegura que la configuració per generar codi per l'acceleradora estigui activa (a través de la utilitat *module*), compila el nostre programa (*nvc o nvcc*) donant-nos tota la informació possible sobre la paral·lelització feta i l'executa. El nom del codi font a compilar es passa com a paràmetre a l'script. Si només passem aquest argument, el programa serà executat sense fer servir cap eina d'anàlisi de rendiment, però si afegim l'opció "-prof" quan cridem l'script, aleshores executa el nostre codi fent l'anàlisi amb *nvprof*. La sortida produïda pel programa, així com la dels potencials anàlisis de rendiment, es desa en un arxiu de nom *slurm-<idjob>.out*

Exercicis

a) Reducció en CUDA:

En primer lloc, us proporcionem el codi seqüencial per fer una reducció (suma) d'un vector de 2^{27} elements (reduct-seq.c).

A més, a classe vam desenvolupar un kernel per fer una reducció (de suma) en paral·lel en la que cada thread sumava dos elements consecutius de l'array. El codi de la funció que feia la reducció era bàsicament el següent (teniu tot el programa a l'arxiu reduct-0.cu):

```
__global__ void reduce (int *V_d, int *res_d, unsigned long n){
  __shared__ int partialSum[2*BLOCKSIZE];
  unsigned int t = threadIdx.x;
  unsigned int start = 2*blockIdx.x*blockDim.x;

  partialSum[2*t] = V_d[start + 2*t];
  partialSum[2*t+1] = V_d[start + 2*t + 1];

  for (unsigned int stride=1; stride<=blockDim.x; stride*=2) {
     __syncthreads();
    if (t % stride == 0) partialSum[2*t] += partialSum[2*t+stride];
  }
  if (t == 0) atomicAdd(res_d, partialSum[0]);
}
```

i. Abans de començar, feu els càlculs teòrics de quina és l'acceleració (speedup) potencial que podríem obtenir en aquest cas particular (sumar 2^{27} elements) aplicant un algorisme de reducció si disposéssim dels recursos necessaris.

Per un algoritme de reducció en paral·lel genèric, assumint que l'estructura a reduir és un array, i apliquem qualsevol de les operacions (suma, resta, mínim, màxim...):

- S'opera amb els elements de l'array en parelles d'operands consecutius, és a dir, s'opera l'element 0 amb l'1, el 2 amb el 3, etc. I guardem el resultat (convencionalment) a la posició del primer dels operands de cada parella.
- Un cop fet aquest primer pas, es repeteix, però doblant l'espai entre operands. És a dir, operem l'element 0 (on es guarda el resultat del primer pas entre 0 i 1) amb l'element 2 (resultat del primer pas entre 2 i 3).

Així, en cada pas de la reducció fem la meitat d'operacions (i de resultats obtinguts), fins arribar a una operació simple de dos operands.

Cal recordar que assumim que tenim recursos suficients i, per tant, podrem fer N operacions a la vegada en el temps de 1. Llavors, el que ens interessa saber, és el nombre de passos que farem.

Necessitarem tant passos com per a arribar a una operació amb dos operands finals. És a dir, necessitarem **prous passos per a dividir el nombre d'elements entre 2, com per a arribar a 1**. Per a obtindre aquest nombre, calculem el logaritme en base 2 de N ($\log_2 N$).

Si en el càlcul seqüencial fèiem servir N passos per a obtindre el mateix resultat que amb la reducció, fem el càlcul del speedup:

$\text{Speedup} = T_{\text{seqüencial}}/T_{\text{paral·lel}} = N/\log_2 N$

Amb $N = 2^{27}$:

$\text{Speedup} = (2^{27})/\log_2(2^{27}) = (2^{27})/27 =$ **4971026x**

ii. Executeu el programa seqüencial i la versió paral·lela que us hem proporcionat, quina és l'acceleració que obteniu?

$T_{\text{seqüencial}} = 0{,}406044$ s
$T_{\text{paral·lel}} = 2.105.707$ ns $= 0.00210$ s

Speedup = **192.8302x**

iii. L'algorisme de reducció que us hem és ineficient, ja que:

- Cada thread accedeix a dos elements veïns
- A partir de la 2a iteració del for tenim divergència en tots els warps.

Programeu un nou algorisme de reducció en el que es solucionin aquestes ineficiències, fent:

- Que cada thread accedeixi a dos elements a distància BLOCKSIZE en el moment de fer la còpia (veure fig. 1)

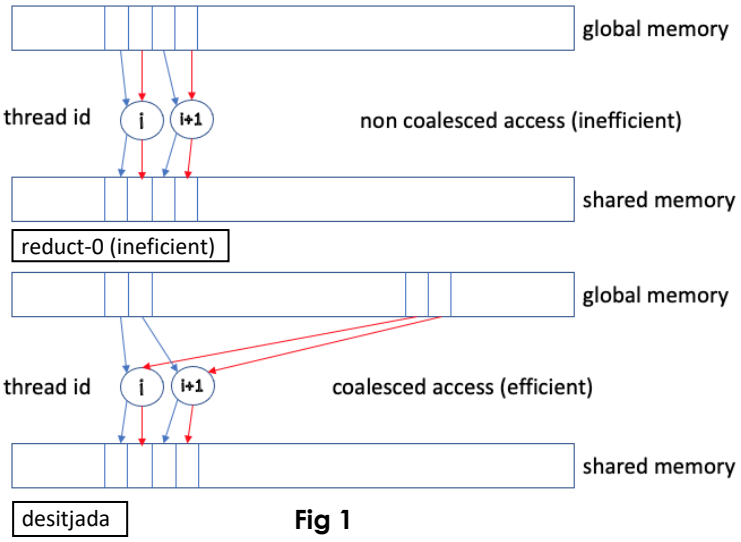

Fig 1

- Que en el for es minimitzi la divergència en cada warp (veure fig. 2).

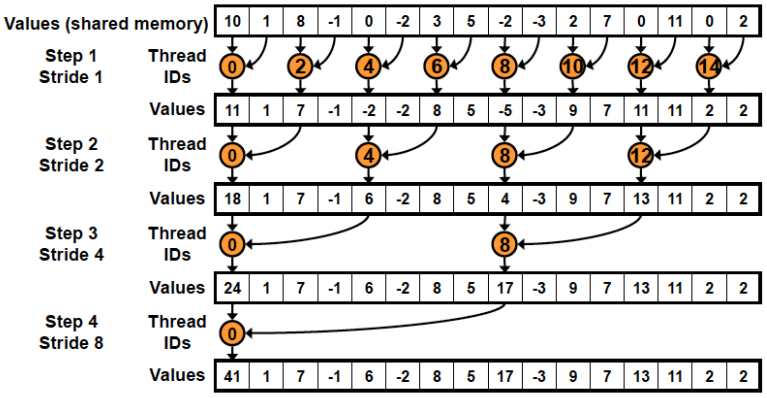

Fig 2. Reduct-0 (ineficient)

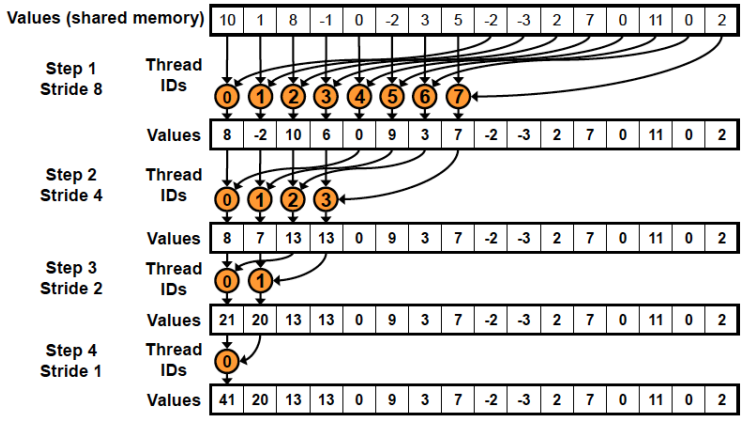

Fig 2. Desitjada

Proporcioneu el codi de la rutina que heu programat (només cal el codi del kernel, la resta és igual que pel reduct-0.cu):

```
__global__ void reduce_v2(int *V_d, int *res_d, unsigned long n){
    __shared__ int partialSum[2*BLOCKSIZE];
    unsigned int t = threadIdx.x;
    unsigned int start = 2*blockIdx.x*blockDim.x;

    partialSum[t] = V_d[start + t];
    partialSum[t+blockDim.x] = V_d[start + blockDim.x+ t ];

    for (unsigned int stride=blockDim.x; stride>0; stride/=2){
    __syncthreads();
    if (t< stride) partialSum[t] += partialSum[t+stride];
    }
    if (t == 0) atomicAdd(res_d, partialSum[0]);
}
```

iv. Quina acceleració obteniu per aquesta versió del codi?

$T_{seqüencial}$ = 0,397660 s
$T_{paral·lel}$ = 1.465.319 ns = 0.00147 s

Speedup = **271.3812x**