**Exercicis Pràctics OpenACC i CUDA (Part III)**

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/Avaluats-problemes/Cuda-OpenACC/sessio3/).
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*
4. **Múltiples loops.**

**// arxiu stencil.c**

**// En aquest cas, donada la llargària del codi, no el copiem en aquest enunciat.**

* + 1. Proveu una primera aproximació, fent servir les directives *kernels* i *parallel* abans del for extern (sobre iter). D’acord amb la sortida donada pel compilador, quina paral·lelització s’ha fet?
    2. Corrobora la sortida del nvprof la vostra resposta anterior? (expliqueu la raó)
    3. Canvia la situació si les directives es posen dins del for extern? És a dir, indicant-li al compilador que paral·lelitzi el codi intern del for sobre iter (fors sobre i).
    4. Proveu ara de paral·lelitzar els bucles interns creant una regió paral·lela per a cada un dels bucles dobles i afegint la clàusula acc loop abans del cada for. Aconsegueix el compilador fer ara la paral·lelització? Quants cops es copien dades entre la CPU i el dispositiu? Quants threads es creen? Quants elements computa cada thread?

1. **Fent servir l’algorisme de reducció.**

El codi d’una funció que realitza la multiplicació escalar de dos vectors pot semblar-se a:

**//escalar-seq.c**

float scalar\_mult(float vect1[], float vect2[], int n) {

float res = 0.0;

for( int i = 0; i < n; i++ )

res += vect1[i]\*vect2[i];

retorn res;

}

Programeu un kernel en CUDA que paral·lelitzi aquesta operació en la GPU:

1. Feu una versió en la que cada thread multiplica només un element del vector 1 amb el corresponent del vector 2. Quin és el codi del kernel resultant?
2. Feu una versió en la que cada thread multiplica 4 elements dels vectors 1 y 2 (la distància entre els elements multiplicats ha de ser blockdim.x per optimitzar l’accés a memòria). Quin és el codi de la nova versió del kernel?
3. Proveu les diferents versions dels vostres kernels amb vectors de 1 M posicions i mides de bloc de 128, 256 i 512 threads. Quina acceleració obteniu en cada cas?

**NOTA**: la reducció resultant de cada bloc de threads s’ha d’acumular en una variable global (resultat del producte escalar) fent servir la funció atomicAdd de Cuda.