David Morillo Massagué (1666540)

Exercicis Pràctics OpenACC i CUDA (Part I)

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/problemes-practics/OpenACC-CUDA/sessio1).
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) Preliminars

```
//Arxiu hello.c
#include <stdio.h>
#ifdef _OPENACC
#include <openacc.h>
#endif

int main(void) {
#ifdef _OPENACC
        acc_device_t devtype;
#endif
        printf("Hello world from OpenACC\n");
#ifdef _OPENACC
        devtype = acc_get_device_type();
        printf("Number of available OpenACC devices: %d\n",
        acc_get_num_devices(devtype));
        printf("Type of available OpenACC devices: %d\n", devtype);
#else
        printf("Code compiled without OpenACC\n");
#endif
    return 0;
}
```

i) Quina és la sortida quan executem el programa havent-lo compilat amb *nvc -Minfo=all -o executable hello.c* (segona opció en el job.sub)?

Arxiu slurm.out:

```
Entro
Hello world from OpenACC
Code compiled without OpenACC
```

El codi detecta que no s'ha compilat fent servir la clàusula per a funcionar amb OpenAcc (`#ifdef _OPENACC`), i printa el missatge per indicar-ho.

ii) Quina és la sortida que es produeix quan executem el programa havent-lo compilat amb *nvc -acc=gpu -ta=tesla -Minfo=all -o hello hello.c*? (primera opció en el job.sub)

Arxiu slurm.out:

```
Entro
Hello world from OpenACC
Number of available OpenACC devices: 1
Type of available OpenACC devices: 4
```

Aquesta execució mostra informació dels dispositius que executen el programa i el seu tipus. La diferència entre aquesta compilació i l'anterior són les flags: "`-acc=gpu -ta=tesla`", que indica que l'acceleració es farà a la gpu, i el nom d'aquesta.

iii) Quina és la funcionalitat de la directiva #ifdef-[#else]-#endif?

S'utilitza per a executar una part del codi del programa segons es fa servir OpenAcc o no (macro _OPENACC). Comprovem amb "#ifdef _OPENACC" si el codi està compilat amb OpenAcc, i executa el codi. Si no és així, executa el codi on és el "#else". El "#endif" tanca aquest bloc condicional i continua executant el codi del programa, estigui compilat amb OpenAcc o no.

iv) Què ens indica la macro _OPENACC?

Ens indica si el codi està compilat amb suport d'OpenAcc o no (booleà).

b) Directives bàsiques. Per el següent codi, escriviu el bucle necessari per fer la suma dels vectors i genereu tres versions paral·leles, una fent servir la directiva kernels, una fent servir la directiva parallel (feu servir la directiva sense cap altra clàusula) i una darrera programant un kernel de CUDA per fer la suma (cada thread calcula un sol element del resultat). Executeu les tres versions obtenint dades d'anàlisi de rendiment. (per les 2 primeres versions heu de compilar amb la 1a opció present a job.sub, per la versió CUDA heu de fer servir la 3a opció)

```
//Arxiu sum.c
#include <stdio.h>
#ifdef _OPENACC
#include <openacc.h>
#endif

#define NX 102400
```

```
int main(void)
{
   double vecA[NX], vecB[NX], vecC[NX];
   double sum;
   int i;
   /* Initialization of the vectors */
   for (i = 0; i < NX; i++) {
      vecA[i] = 1.0 / ((double) (NX - i));
      vecB[i] = vecA[i] * vecA[i];
   }

   //Codi seqüencial:
   for (i = 0; i < NX; i++) {
         vecC[i] = vecA[i] + vecB[i];
   }

   //Fent servir kernels:
   #pragma acc kernels
   {
      for (i = 0; i < NX; i++) {
         vecC[i] = vecA[i] + vecB[i];
      }
   }

   //Fent servir parallel:
   #pragma acc parallel loop
   for (i = 0; i < NX; i++) {
      vecC[i] = vecA[i] + vecB[i];
   }

   sum = 0.0;
   /* Compute the check value */
   for (i = 0; i < NX; i++)
      sum += vecC[i];
   printf("Reduction sum: %18.16f\n", sum);
   return 0;
}
```

Kernel CUDA:

```
__global__ void vectorAdd(double *vecA, double *vecB, double *vecC, int n) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < n) {
      vecC[i] = vecA[i] + vecB[i];
   }
}
```

i) A partir de les dades obtingudes pel nvprof. Com és la graella de threads generada en cada cas (kernels i parallel)? Quants elements de l'array vecC calcula cada thread en cada cas? D'acord amb els temps de còmput (només de còmput), quina és la millor decisió?

Cas de kernels:

- Nom del kernel: main_22_gpu
- Número d'instàncies: 1

- Temps total d'execució del kernel: 3.712 ns
- Configuració de la graella i els blocs:
- Graella: dimensió X = 800, dimensió Y = 1, dimensió Z = 1
- Bloc: dimensió X = 128, dimensió Y = 1, dimensió Z = 1

Cas de parallel:

- Nom del kernel: main_21_gpu
- Número d'instàncies: 1
- Temps total d'execució del kernel: 3.648 ns
- Configuració de la graella i els blocs:
- Graella: dimensió X = 800, dimensió Y = 1, dimensió Z = 1
- Bloc: dimensió X = 128, dimensió Y = 1, dimensió Z = 1

Cada thread calcula 128 elements de l'array "vecC" (Mida del vector: 102400 entre la dimensió de la graella: 800, = dimensió del bloc: 128). Segons el temps de còmput, la millor desició seria fer servir el bucle amb la clàusula "parallel", encara que amb una diferència molt petita respecte a l'altre mètode. (Speedup: 1.0175x respecte a kernels)

ii) Genereu una nova versió del kernel de CUDA en la que cada thread sumi dos elements consecutius de l'array i un altre en la que cada kernel sumi dos elements a distància blockDimx.x. Compari les dades de rendiment de les 3 versions CUDA. Pot explicar les raons per les que obtenim aquests resultats?

Kernel CUDA (de 2 en 2 elements amb distància 1):

```
__global__ void vectorAdd(const double *vecA, const double *vecB,
double *vecC) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   vecC[tid] = vecA[tid] + vecB[tid];
   vecC[tid+1] = vecA[tid+1] + vecB[tid+1];
}
```

Kernel CUDA (Distància blockDim.x):

```
__global__ void vectorAdd(const double *vecA, const double *vecB,
double *vecC) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   vecC[tid] = vecA[tid] + vecB[tid];
   vecC[tid + blockDim.x] = vecA[tid + blockDim.x] + vecB[tid + blockDim.x];
}
```