**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/capmc/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**

1. **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;

**}**

* + 1. Quina és la sortida quan executem el programa havent-lo compilat amb *nvc -Minfo=all -o executable hello.c*?

La sortida és la següent:

cpu-bind=MASK - aolin23, task 0 0 [6128]: mask 0x1 set

Hello world from OpenACC

Code compiled without OpenACC

* + 1. 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*?

La sortida és la següent:

cpu-bind=MASK - aolin23, task 0 0 [6901]: mask 0x1 set

Hello world from OpenACC

Number of available OpenACC devices: 1

Type of available OpenACC devices: 4

* + 1. Quina és la funcionalitat de la directiva #ifdef-[#else]-#endif?

Controlar el que fa el programa segons estigui definida o no una certa variable (en aquest cas en concret, \_OPENACC). En cas que ho estigui, farà l’include de OpenAcc i els prints dins del #ifdef \_OPENACC. En cas contrari, imprimirà per pantalla el missatge “Code compiled without OpenACC”.

* + 1. Què ens indica la macro \_OPENACC?

Que s’està usant OpenAcc (en el cas que estigui definit) o no (en cas contrari).

1. **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.**

**//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];

}

**/\* TODO:**

**\* Implement vector addition on device with OpenACC**

**\* vecC = vecA + vecB**

**\*/**

sum = 0.0;

/\* Compute the check value \*/

for (i = 0; i < NX; i++)

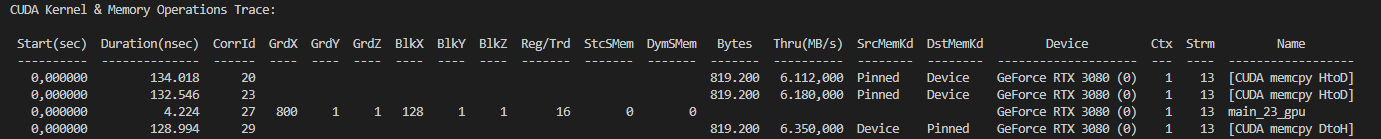
sum += vecC[i];

printf("Reduction sum: %18.16f\n", sum);

return 0;

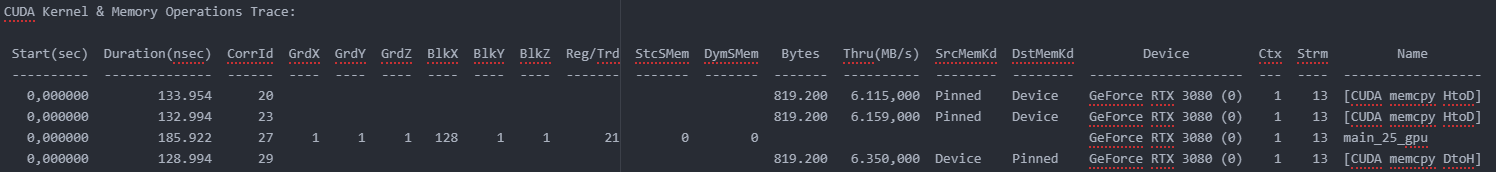
}

* + 1. 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?

Amb el cas kernels: 

Veiem que s’utilitzen 800 grids amb 128 blocs (800x128 = 102400). Com sabem que l’array vecC té 102400 elements, deduïm que cada thread calcula 1 element de l’array.

Amb el cas parallel:

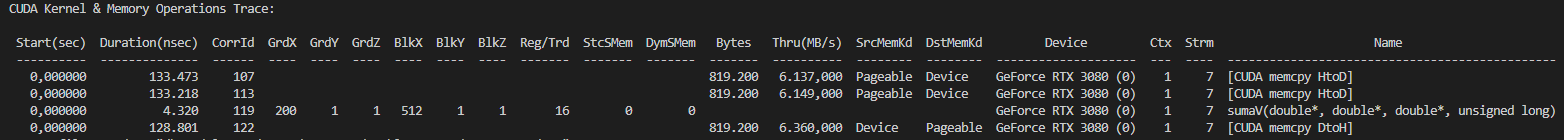


Veiem que s’utilitza 1 grid amb 128 blocs, pel que deduïm que 102400/128 = 800, cada thread calcula 800 elements de l’array.

D’acord amb el temps de còmput és molt millor l’opció kernels (4.2 segons) que parallel (185.9 segons).

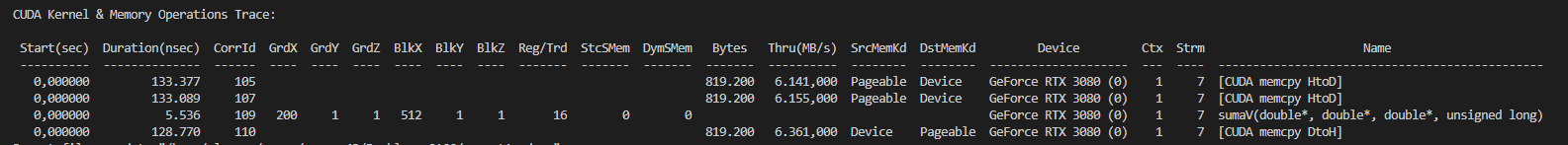
* + 1. Genereu una nova versió del kernel de CUDA en la que cada thread sumi dos elements consecutius de l’array i un altre en la que cada kernel sumi dos elements a distància blockDimx.x. Compari les dades de rendiment de les 3 versions CUDA. Pot explicar les raons per les que obtenim aquests resultats?

Versió distància blockDimx.x (fitxer sum.cu):



Veiem que s’utilitzen 200 grids amb 512 blocs, pel que deduïm que cada thread calcula 1 element de l’array.

Versió dos elements consecutius (fitxer sum2.cu):



Veiem que l’estructura dels threads en grids i blocs són igual al cas anterior.

Destacar també que, tot i que els temps són bastant semblants, la primera versió és la millor pel que fa a temps de còmput (4.3 segons davant 5.5 segons de la segona versió).