OpenMP coprocesadores

1. Consideraciones previas

- Se usará el compilador nvc de Nvidia, que se puede descargar de su página web. En atcgrid está instalado en el nodo atcgrid4.
- El objetivo de estos ejercicios es habituarse a la organización de la GPU y al compilador, y entender la sobrecarga que introduce el uso del coprocesador (GPU, en este caso).
- El compilador nvc espera que el código termine con un salto de línea

Ejercicios basados en los ejemplos del seminario

1. **(a)** Compilar el ejemplo `omp_offload.c` del seminario en el nodo atcgrid4::

```
sbatch -pac4 -Aac --wrap "nvc -O2 -openmp -mp=gpu omp_offload.c -o omp_offload_GPU"
```

(`-openmp` para que tenga en cuenta las directivas OpenMP y `-mp=gpu` para que el código delimitado con target se genere para un dispositivo gpu)

Ejecutar `omp_offload_GPU` usando:

```
srun -pac4 -Aac omp_offload_GPU 35 3 32 > salida.txt
```

CONTENIDO FICHERO: `salida.txt` (**destaque en el resultado de la ejecución con colores las respuestas a las preguntas (b)-(e))**

```
Target device: 1
Tiempo:0.141690016
Iteracción 0, en thread 0/32 del team 0/3
Iteracción 1, en thread 1/32 del team 0/3
Iteracción 2, en thread 2/32 del team 0/3
Iteracción 3, en thread 3/32 del team 0/3
Iteracción 4, en thread 4/32 del team 0/3
Iteracción 5, en thread 5/32 del team 0/3
Iteracción 6, en thread 6/32 del team 0/3
Iteracción 7, en thread 7/32 del team 0/3
Iteracción 8, en thread 8/32 del team 0/3
Iteracción 9, en thread 9/32 del team 0/3
Iteracción 10, en thread 10/32 del team 0/3
Iteracción 11, en thread 11/32 del team 0/3
Iteracción 12, en thread 12/32 del team 0/3
Iteracción 13, en thread 13/32 del team 0/3
Iteracción 14, en thread 14/32 del team 0/3
Iteracción 15, en thread 15/32 del team 0/3
Iteracción 16, en thread 16/32 del team 0/3
Iteracción 17, en thread 17/32 del team 0/3
Iteracción 18, en thread 18/32 del team 0/3
Iteracción 19, en thread 19/32 del team 0/3
Iteracción 20, en thread 20/32 del team 0/3
Iteracción 21, en thread 21/32 del team 0/3
Iteracción 22, en thread 22/32 del team 0/3
```

```
Iteracción 23, en thread 23/32 del team 0/3
Iteracción 24, en thread 24/32 del team 0/3
Iteracción 25, en thread 25/32 del team 0/3
Iteracción 26, en thread 26/32 del team 0/3
Iteracción 27, en thread 27/32 del team 0/3
Iteracción 28, en thread 28/32 del team 0/3
Iteracción 29, en thread 29/32 del team 0/3
Iteracción 30, en thread 30/32 del team 0/3
Iteracción 31, en thread 31/32 del team 0/3
Iteracción 32, en thread 0/32 del team 1/3
Iteracción 33, en thread 1/32 del team 1/3
Iteracción 34, en thread 2/32 del team 1/3
```

Contestar las siguientes preguntas:

(b) ¿Cuántos equipos (*teams*) se han creado y cuántos se han usado realmente en la ejecución?

RESPUESTA:

Como podemos observar, se han creado 3 equipos pero se han utilizado únicamente 2 (0, 1). Se ve en la columna cuya primera fila está marcada de rojo.

(c) ¿Cuántos hilos (*theads*) se han creado en cada equipo y cuántos de esos hilos se han usado en la ejecución?

RESPUESTA:

Podemos ver que para cada uno de los 2 equipos creados se han creado 32 hilos, de los cuales se han usado los 32 en el primer equipo y sólo 3 en el segundo equipo (1). Se observa en la columna cuya primera fila está marcada de morado.

(d) ¿Qué número máximo de iteraciones se ha asignado a un hilo?

RESPUESTA:

Vemos que para cada hilo se ejecuta una única iteración.

(e) ¿Qué número mínimo de iteraciones se ha asignado a un equipo y cuál es ese equipo?

RESPUESTA:

Como se aprecia en lo que se ha marcado de verde, el número mínimo de iteraciones asignadas a un equipo ha sido de 3, concretamente, al equipo 1.

2. Eliminar en `opp_offload.c` `num_teams(nteams)` y `thread_limit(mthreads)` y la entrada como parámetros de `nteams` y `mthreads`. Llamar al código resultante `opp_offload2.c`. Compilar y ejecutar el código para poder contestar a las siguientes preguntas:

(a) ¿Qué número de equipos y de hilos por equipo se usan por defecto?

RESPUESTA:

CAPTURA (que muestre el envío a la cola y el resultado de la ejecución)

Como podemos ver en la imagen, el número de hilos utilizados por defecto es de 1048 y utiliza 48 equipos.

```
srun -pac4 -Aac omp_offload_GPU2 35 > salida2.txt
nano salida2.txt
```

```
Target device: 1
Tiempo:0.130471945
Iteracción 0, en thread 0/1024 del team 0/48
Iteracción 1, en thread 1/1024 del team 0/48
Iteracción 2, en thread 2/1024 del team 0/48
Iteracción 3, en thread 3/1024 del team 0/48
Iteracción 4, en thread 4/1024 del team 0/48
Iteracción 5, en thread 5/1024 del team 0/48
Iteracción 6, en thread 6/1024 del team 0/48
Iteracción 7, en thread 7/1024 del team 0/48
Iteracción 8, en thread 8/1024 del team 0/48
Iteracción 9, en thread 9/1024 del team 0/48
Iteracción 10, en thread 10/1024 del team 0/48
Iteracción 11, en thread 11/1024 del team 0/48
Iteracción 12, en thread 12/1024 del team 0/48
Iteracción 13, en thread 13/1024 del team 0/48
Iteracción 14, en thread 14/1024 del team 0/48
Iteracción 15, en thread 15/1024 del team 0/48
Iteracción 16, en thread 16/1024 del team 0/48
Iteracción 17, en thread 17/1024 del team 0/48
Iteracción 18, en thread 18/1024 del team 0/48
Iteracción 19, en thread 19/1024 del team 0/48
Iteracción 20, en thread 20/1024 del team 0/48
Iteracción 21, en thread 21/1024 del team 0/48
Iteracción 22, en thread 22/1024 del team 0/48
Iteracción 23, en thread 23/1024 del team 0/48
Iteracción 24, en thread 24/1024 del team 0/48
Iteracción 25, en thread 25/1024 del team 0/48
Iteracción 26, en thread 26/1024 del team 0/48
Iteracción 27, en thread 27/1024 del team 0/48
Iteracción 28, en thread 28/1024 del team 0/48
Iteracción 29, en thread 29/1024 del team 0/48
Iteracción 30, en thread 30/1024 del team 0/48
Iteracción 31, en thread 31/1024 del team 0/48
Iteracción 32, en thread 32/1024 del team 0/48
Iteracción 33, en thread 33/1024 del team 0/48
Iteracción 34, en thread 34/1024 del team 0/48
```

(b) ¿Es posible relacionar este número con alguno de los parámetros, comentados en el seminario, que caracterizan al coprocesador que estamos usando? ¿Con cuáles?

RESPUESTA:

Podemos ver que el número por defecto de equipos se corresponde con el *Streaming Multiprocesor (SM)* del Nvidia Quadro RTX 5000 que utiliza el atcgrid. Los 1024 hilos por defecto se corresponden con el máximo de threads por *SM* (o *CUDA Block*).

(c) ¿De qué forma se asignan por defecto las iteraciones del bucle a los equipos y a los hilos dentro de un equipo? Contestar además las siguientes preguntas: ¿a qué equipo y a qué hilo de ese equipo se asigna la iteración 2? Y ¿a qué equipo y a qué hilo de ese equipo se asigna la iteración 1025, si la hubiera? (realizar las ejecuciones que se consideren necesarias para contestar a esta pregunta, en particular, alguna ejecución con un número de iteraciones de al menos 1025)

RESPUESTA:

Se asignan de forma iterativa, la primera iteración corresponde al primer hilo del primer equipo, la iteración 1024 será asignada al primer hilo del segundo equipo. Se puede comprobar en la siguiente imagen:

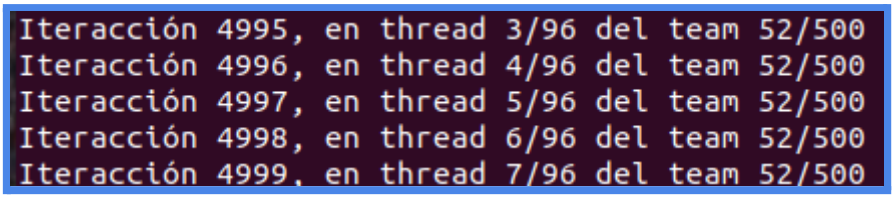
```
Iteracción 1022, en thread 1022/1024 del team 0/48
Iteracción 1023, en thread 1023/1024 del team 0/48
Iteracción 1024, en thread 0/1024 del team 1/48
Iteracción 1025, en thread 1/1024 del team 1/48
Iteracción 1026, en thread 2/1024 del team 1/48
Iteracción 1027, en thread 3/1024 del team 1/48
Iteracción 1028, en thread 4/1024 del team 1/48
Iteracción 1029, en thread 5/1024 del team 1/48
```

3. Ejecutar la versión original, `omp_offload`, con varios valores de entrada hasta que se pueda contestar a las siguientes cuestiones:

(a) ¿Se crean cualquier número de hilos (*threads*) por equipo que se ponga en la entrada al programa? (probar también con algún valor mayor que 3000) En caso negativo, ¿qué número de hilos por equipo son posibles?

RESPUESTA:

Podemos observar que para cada equipo podemos tener un número máximo de 96 hilos. En esta imagen podemos observar que el valor límite es de 96 hilos, habiendo puesto 100 hilos el máximo que nos deja es de 96.



(b) ¿Es posible relacionar el número de hilos por equipo posibles con alguno o algunos de los parámetros, comentados en el seminario, que caracterizan al coprocesador que se está usando? Indicar cuáles e indicar la relación.

RESPUESTA:

Hay 96 de límite ya que contamos con dos Nvidia Quadro RTX 5000, cada una de las cuales con un *SM* de 48, con lo cual el máximo es 96. El límite que se corresponde con la imagen. Por otra parte, al igual que para el anterior ejercicio, los 1024 hilos por defecto se corresponden con el máximo de threads por *SM* (o *CUDA Block*).

4. Eliminar las directivas `teams` y `distribute` en `omp_offload2.c`, llamar al código resultante `opp_offload3.c`. Compilar y ejecutar este código para poder contestar a las siguientes preguntas:

(a) ¿Qué número de equipos y de hilos por equipo se usan por defecto?

RESPUESTA:

Por defecto se utilizan 1024 hilos y un único equipo.

(b) ¿Qué tanto por ciento del número de núcleos de procesamiento paralelo de la GPU se están utilizando? Justificar respuesta.

RESPUESTA:

Podemos observar que para esta ejecución donde se han hecho 1030 iteraciones se han ejecutado en el mismo hilo varias iteraciones a la vez de tal manera que la última iteración sea asignada al último hilo. Si incrementamos el número de iteraciones el grado de paralelismo aumentará proporcionalmente, es decir, si ejecutáramos con 3500 iteraciones el grado de paralelismo sería de 3 o 4 en función del hilo, siendo en los primeros hilos de grado 4 mientras que en los últimos de grado 3.

5. En el código `daxpbyz32_ompoff.c` se calcula (a y b son escalares, x, y y z son vectores):

$$z = a \cdot x + b \cdot y$$

Se han introducido funciones omp_get_wtime() para obtener el tiempo de ejecución de las diferentes construcciones/directivas target utilizadas en el código.

1) t2-t1 es el tiempo de target enter data, que reserva de espacio en el dispositivo coprocesador para x, y, z, N y p, y transfiere del host al coprocesador de aquellas que se mapean con to (x, N y p).
2) t3-t2 es el tiempo del primer target teams distribute parallel for del código, que se ejecuta en paralelo en el coprocesador del bucle:

   for (int i = 0; i < N; i++) z[i] = p * x[i];

3) t4-t3 es el tiempo de target update, que transfiere del host al coprocesador p e y.
4) t5-t4 es el tiempo del segundo target teams distribute parallel for del código, que ejecuta en paralelo en el coprocesador del bucle:

   for (int i = 0; i < N; i++) z[i] = z[i] + p * y[i];

5) t6-t7 es el tiempo que supone target exit data, que transfiere los resultados de las variables con from y libera el espacio ocupado en la memoria del coprocesador.

Compilar `daxpbyz32_off.c` para la GPU y para las CPUs de atctrid4 usando:

sbatch –pac4 –Aac --wrap “nvc -O2 -openmp -mp=gpu daxpbyz32_ompoff.c -o daxpbyz32_ompoff_GPU”
sbatch –pac4 –Aac --wrap “nvc -O2 -openmp -mp=multicore daxpbyz32_ompoff.c -o daxpbyz32_ompoff_CPU”

En `daxpbyz32_off_GPU` el coprocesador será la GPU del nodo y, en `daxpbyz32_off_CPU`, será el propio host. En ambos casos la ejecución aprovecha el paralelismo a nivel de flujo de instrucciones del coprocesador. Ejecutar ambos para varios valores de entrada usando un número de componentes N para los vectores entre 1000 y 100000 y contestar a las siguientes preguntas.

CAPTURAS DE PANTALLA (que muestren la compilación y las ejecuciones):

```
Salida CPU:

Target device: 0
 *      Tiempo: ((Reserva+inicialización) host 0.000019073) + (target enter data 0.000000000) + (target1 0.001687050) + (host actualiza 0.000012875) + (target data update 0.000000000) + (target2
0.000010967) + (target exit data 0.000000000)= 0.001729965
 / Tamaño Vectores:5000
 / alpha*x[0]+beta*y[0]=z[0](2.000000*500.000000+3.000000*500.000000=2500.000000) /
 / alpha*x[4999]+beta*y[4999]=z[4999](2.000000*999.900024+3.000000*0.100000=2000.100098) /
```

```
Salida GPU:

Target device: 1
 *      Tiempo: ((Reserva+inicialización) host 0.000011921) + (target enter data 0.121566057) + (target1 0.000509977) + (host actualiza 0.000002146) + (target data update 0.000039816) + (target2
0.000035048) + (target exit data 0.000051022)= 0.122215986
 / Tamaño Vectores:5000
 / alpha*x[0]+beta*y[0]=z[0](2.000000*500.000000+3.000000*500.000000=2500.000000) /
 / alpha*x[4999]+beta*y[4999]=z[4999](2.000000*999.900024+3.000000*0.100000=2000.100098) /
```

(a) ¿Qué construcción o directiva target supone más tiempo en la GPU?, ¿a qué se debe?

RESPUESTA:

Vemos que la directiva target que más tiempo tarda es *“target enter data”*. Es el encargado de la entrada de datos del programa y, como suele suceder, las operaciones de entrada/salida son las más

costosas. Tarda más que la operación de salida ya que tiene que mapear todas las variables a un ámbito de datos coprocesador

(b) ¿Qué construcciones o directivas target suponen más tiempo en la GPU que en la CPU?, ¿a qué se debe?

RESPUESTA:

	CPU	GPU
host	0.000019073	0.000011921
target enter data	0.000000000	0.121566057
target1	0.001687050	0.000509977
host actualiza	0.000012875	0.000002146
target data update	0.000000000	0.000039816
target2	0.000010967	0.000035048
target exit data	0.000000000	0.000051022
TOTAL	0.001729965	0.122215986

Podemos ver en la anterior imagen que la diferencia más clara de tiempos corresponde a las operaciones de entrada y salida, tardando, aparentemente, un tiempo nulo en la CPU mientras que para la GPU es lo más costoso.
Esto es debido a que en la arquitectura Coprocesador-GPU se usa un bus de E/S para la comunicación entre el host y la GPU, mientras que para la CPU no es necesario este bus. Luego es por esto que existe esta gran diferencia de tiempos con las operaciones entrada/salida. Vemos que para el resto de valores son prácticamente idénticos.

2. Resto de ejercicios

6. A partir del código secuencial que calcula PI, obtener un código paralelo basado en las construcciones/directivas OpenMP para ejecutar código en coprocesadores. El código debe usar como entrada el número de intervalos de integración y debe imprimir el valor de PI calculado, el error cometido y los tiempos (1) del cálculo de pi y (2) de la trasferencia hacia y desde la GPU. Generar dos ejecutables, uno que use como coprocesador la CPU y otro que use la GPU. Comparar los tiempos de ejecución obtenidos en atcgrid4 con la CPU y la GPU, indicar cuáles son mayores y razonar los motivos.

CAPTURA CÓDIGO FUENTE: `pi-ompoff.c`

```
int main(int argc, char **argv)
{
  register double width;
  double sum;
  register int intervals, i;
  struct timespec cgt1,cgt2; double ncgt; //para tiempo

  //Los procesos calculan PI en paralelo
  if (argc<2) {printf("Falta número de intevalos");exit(-1);}
  intervals=atoi(argv[1]);
  if (intervals<1) {intervals=1E6; printf("Intervalos=%d",intervals);}
  width = 1.0 / intervals;
  sum = 0;
//#pragma omp target enter data map(to: width, sum, intervals)
  clock_gettime(CLOCK_REALTIME,&cgt1);
#pragma omp target teams distribute parallel for map(to: width, sum, intervals)
  for (i=0; i<intervals; i++) {
    register double x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }
  sum *= width;
//#pragma omp target exit data map(delete: width, intervals)
  clock_gettime(CLOCK_REALTIME,&cgt2);
  ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
       (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
  printf("Iteraciones:\t%d\t. PI:\t%26.24f\t. Tiempo:\t%8.6f\n", intervals,sum,ncgt);
```

```
 return(0);
}
```

CAPTURAS DE PANTALLA (mostrar la compilación y la ejecución para 10000000 intervalos de integración en atcgrid4 – envío(s) a la cola):

CPU:

```
Iteraciones:    1000000 . PI:    3.14159265358986239391 4700      . Tiempo:       0.002784
```

GPU:

```
Iteraciones:    1000000 . PI:    3.14159265358979323846 2640      . Tiempo:       1.407270
```

RESPUESTA:

Podemos observar que el tiempo de ejecución para *GPU* es bastante mayor que el de *CPU*, concretamente más de 500 veces mayor. Esto es debido a que en un coprocesador o GPU para cada iteración se ha de cargar el valor de la variable *sum* que es la encargada de ir almacenando el valor de las sumas de los intervalos. Consecuentemente, en la siguiente iteración que corresponda a otro *SM* se tendrá que acceder a memoria y cargar el valor guardado de *sum* de las anteriores iteraciones. En definitiva, la diferencia de tiempos es tan significativa ya que entre iteraciones los *Streaming Multiprocesor* han de acceder a memoria para modificar el valor de la variable, mientras que, por otro lado, la *CPU* no necesita realizar estos accesos a memoria ya que la memoria está compartida entre todos los *multicores*.