Arquitecturas Avanzadas

Grupo AA-2-1

25/11/2019

Julio César Velasquez Cardenas 1397896
Sergio Prada Maeso 1459122
Juan Carlos Bermúdez Rodríguez 1455486

1-

- How many GPUs are available in this node?

Para el Aolin 23 hay 1 que es GeForce GTX 1080.
Para el Aolin 24 hay 1 que es la GeForce 1080 Ti.
Para el Aolin Master 2 que son GeForce GTX 750 Ti ambas.

- Are they all the same?

No, ya que en los primeros nodos solo hay una, y en la Master hay 2.
Solo en la Master ambas son la misma, pero respecto a las demás, ningún nodo comparte la misma.

- How many SMs (multiprocessors) does each GPU have?

Aolin23 tiene 20.
Aolin24 tiene 28.
Aolinmaster tiene 5 cada una (Son la misma).

2-

-Does one loop iteration affect other loop iterations?

En el primer loop la iteración x depende del resultado de x+1, así que una iteración tiene impacto en las otras. En el segundo y tercer loop, el for englobado en el kernels los datos del vector y[i] se usan para recalcular este valor, y se afectan a ellos mismos.

-Are there any data dependencies that prevent parallelization of loop iterations?

- Como podemos observar, tenemos una dependencia en la variable x que no deja paralelizar

According to the information generated by the compiler, which loops were parallelized? What's the reason that prevented parallelization of the other loops?

Solo el tercer bucle ha podido ser paralelizado, puesto que en los otros dos bucles se ha encontrado dependencias el variables que impiden a este ejecutarse en paralelo.

```
loop_1:
        18, Intensity = 0.0
        Generating implicit allocate(x[:1000])
        Generating implicit copyin(x[1:999])
        Generating implicit copyout(x[:999])
        19, Intensity = 0.50
        Loop carried dependence of x-> prevents parallelization
        Loop carried backward dependence of x-> prevents vectorization
        Accelerator serial kernel generated
        Generating Tesla code
        19, #pragma acc loop seq
loop_2:
```

```
        32, Intensity = 0.0
        36, Intensity = 0.0
        Generating implicit copyin(x[:1000])
        Generating implicit copy(y[:1000])
        37, Intensity = 0.67
        Complex loop carried dependence of x-> prevents parallelization
        Loop carried dependence of y-> prevents parallelization
        Loop carried backward dependence of y-> prevents vectorization
        Accelerator serial kernel generated
        Generating Tesla code
        37, #pragma acc loop seq
loop_3:
        49, Intensity = 0.0
        53, Intensity = 0.0
        Generating implicit copyin(x[:1000])
        Generating implicit copy(y[:1000])
        54, Intensity = 0.67
        Loop is parallelizable
        Generating Tesla code
        54, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Modify the second loop appropriately, so that the loop is parallelized.

```
void loop_2(){

   int     N=1000;
   float     a = 3.0f;
   float  * __restrict__ x = (float*)malloc(N * sizeof(float));
   float  * __restrict__ y = (float*)malloc(N * sizeof(float));

   for (int i = 0; i < N; ++i) {
          x[i] = 2.0f;
          y[i] = 1.0f;
   }
   #pragma acc kernels
   for (int i = 0; i < N; ++i) {
          y[i] = a * x[i] + y[i];
   }
}

loop_1:
        18, Intensity = 0.0
        Generating implicit allocate(x[:1000])
        Generating implicit copyin(x[1:999])
        Generating implicit copyout(x[:999])
```

```
        19, Intensity = 0.50
        Loop carried dependence of x-> prevents parallelization
        Loop carried backward dependence of x-> prevents vectorization
        Accelerator serial kernel generated
        Generating Tesla code
        19, #pragma acc loop seq
loop_2:
        32, Intensity = 0.0
        36, Intensity = 0.0
        Generating implicit copyin(x[:1000])
        Generating implicit copy(y[:1000])
        37, Intensity = 0.67
        Loop is parallelizable
        Generating Tesla code
        37, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
loop_3:
        49, Intensity = 0.0
        53, Intensity = 0.0
        Generating implicit copyin(x[:1000])
        Generating implicit copy(y[:1000])
        54, Intensity = 0.67
        Loop is parallelizable
        Generating Tesla code
        54, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

4-Modify the code of addition.c and add additional clauses to parallelize the main loops and
use eventually explicit data directives. You may use either kernels or parallel loop directives (or both, if you have time).

Hemos añadido #pragma acc data copy(m_v[0:MAXN],alpha,NB_ITER,size) previo al bucle while que recorre los tamaños de problema, para copiar una vez las variables del host al device (GPU), y no hacerlo una vez ya que lo tendríamos que copiar una vez por tamaño de problemas, ya que las operaciones de copia de memoria son costosas y se pierde mucho tiempo.

También hemos añadido #pragma acc data copy(m_gpu[0:MAXN]) porque se necesita copiar por cada tamaño de problema la variable m_gpu, debido a que si no se hace, por mucho que se ahorra tiempo evitando operaciones de copia de memoria a la GPU, el resultado de las operaciones es incorrecto.

Por último, la cláusula #pragma acc kernels loop auto (añadida en el bucle más interno) del cálculo de m_gpu, hace que la GPU ejecute la región de este mismo bucle, , especificando como se ha de ejecutar en paralelo (gangs, workers...), indicando al compilador que busque y aplique medidas para mejorar la ejecución del bucle automáticamente.

Código:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

#include "util.h"

#define MAXN 135000000
#define NB_ITER 10


float m_cpu[MAXN];
float m_gpu[MAXN];
float m_v[MAXN];

long sizes[] = {
  20,
  44,
  123,
  1024,
  2048,
  4092,
  32768,
  1048756,
  2097152,
  4194304,
  8388608,
 16777216,
 33554432,
 67108864,
 134217728,
  0      /* End marker! do not remove */
};

int main(int argc, char **argv)
{
  float alpha = 2.0f;
  long   i, j, k;
  int   size ;
  int   seed = 2;

  for( i = 0 ; i < MAXN ; i++ ) {
      m_v[i]   = (float)(((int)(i+seed) % 5) + 2) / 2.0f ;
    m_cpu[i] = (float)(((int)(i+seed) % 5) + 2) / 2.0f ;
    m_gpu[i] = (float)(((int)(i+seed) % 5) + 2) / 2.0f ;
  }
```

```
  i = 0;
  #pragma acc data copy(m_v[0:MAXN],alpha,NB_ITER,size)
  {
  while( sizes[i] != 0 )
  {
    size = sizes[i++];

    printf("BEFORE(%4d): %8.4f %8.4f (...) %8.4f %8.4f \n", size,
m_cpu[0], m_cpu[1], m_cpu[size-2], m_cpu[size-1]);

    // CPU version
    double time_gpu, time_cpu, t0, t1;

    t0 = wallclock();
    for(j = 0; j<NB_ITER; j++) {
      for( k = 0 ; k < size ; k++ ) {
        m_cpu[k] = m_cpu[k] + m_v[i] + alpha;
      }
    }

    t1 = wallclock();
    time_cpu = (t1-t0)/NB_ITER;

    printf( "CPU: [%4d] %10lf ms\n", size,  (double)time_cpu * 1e3 );

      // GPU version
      t0 = wallclock();
      #pragma acc data copy(m_gpu[0:MAXN])
      {
            for(j = 0; j<NB_ITER; j++)
            {
                  #pragma acc kernels loop auto
                  for( k = 0 ; k < size ; k++ )
                  {
                        m_gpu[k] = m_gpu[k] + m_v[i] + alpha;
                  }
            }
      }
      t1 = wallclock();
    time_gpu = (t1-t0)/NB_ITER;

    printf("GPU: [%4d] %10lf ms\n", size,  (double)time_gpu * 1e3 );
    printf("AFTER_CPU(%4d) : %8.4f %8.4f (...) %8.4f %8.4f \n", size,
m_cpu[0], m_cpu[1], m_cpu[size-2], m_cpu[size-1]);
    printf("AFTER_GPU(%4d) : %8.4f %8.4f (...) %8.4f %8.4f \n", size,
m_gpu[0], m_gpu[1], m_gpu[size-2], m_gpu[size-1]);
    printf("\n") ;
```

```
  }
  }

  return 0;
}
```