Proyecto de Laboratorio de Prácticas

Paralelización OpenACC

En este apartado del proyecto, la compilación y ejecución del código se ha llevado a cabo utilizando un script de trabajo denominado job.sub. El script job.sub se envía a la cola de trabajo utilizando el comando: sbatch job.sub 4, el número 4 indica el nombre de k centroides que habrá.

Las secciones del código que pueden beneficiarse de ejecutarse en el acelerador i que, por lo tanto, justifican descargarlas en l'aceleradora són las siguientes:

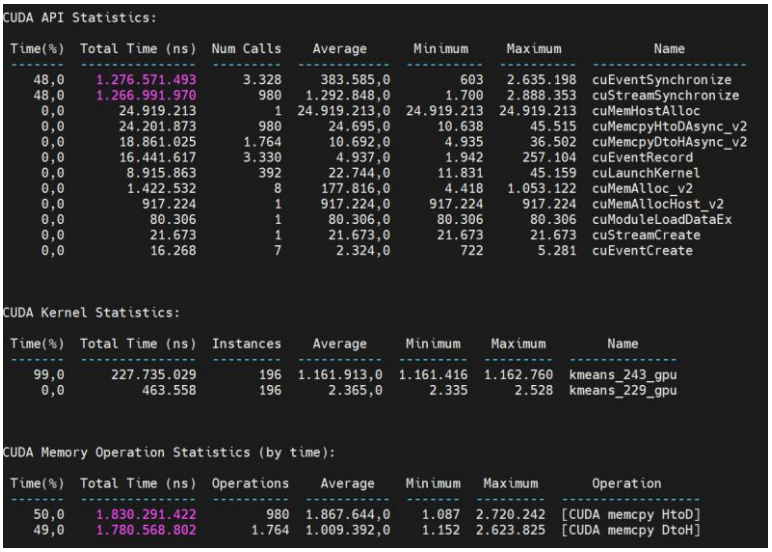
- **Cálculo del clúster más cercano**. Esta sección de código se encarga, mediante un bucle, de calcular por cada píxel el centroide más cercano. Esta parte del código realiza num_pixels*k veces debido a que por cada píxel se encuentra el centroide más cercano.
- **Actualización de la media de los colores por cada clúster**. Esta sección de código se encarga de actualizar los promedios de color y contadores de puntos de cada centroide con los valores de los píxeles más cercanos. Esta parte del código realiza k veces debido a que por cada centroide se actualizan los valores adequados.

Para mejorar el rendimiento del programa, hemos optado por utilizar la librería OpenACC. Hemos adaptado la paralelización de la función kmeans() de su versión original que usaba la directiva OpenMP a una versión utilizando la directiva OpenACC.

A continuación se explicaran los cambios realizados en el código. Primero de todo, se eliminan todas las directivas de OpenMP presentes en la función kmeans(). Posteriormente, añadimos las directivas de OpenACC que hemos considerado necessarias que són las siguentes:

- **#pragma acc data copy** : Copiamos los datos necessarios en la aceleradora antes de iniciar el càlculo. En concreto copiamos los siguientes datos: centroides[0:k], pixels[0:num_pixels], closest_centroide[0:num_pixels]. Es decir del vector centroides copiamos de la posición 0 hasta k, del vector pixels copiamos de la posición 0 hasta num_pixels i del vector closest_centroide copiamos de 0 a num_pixels.
- **#pragma acc parallel loop present(centroides[0:k])**: Esta directiva paraleliza la ejecución del bucle donde se inicializan los centroides, indicando que los datos de los centroides ya están presentes en la GPU. Esto evita la necesidad de realizar transferencias de datos adicionales desde la CPU a la GPU, ya que han sido previamente copiados mediante el copy.
- **#pragma acc parallel loop**: Se paraleliza el bucle de los promedios de color para cada centroide.

Al ejecutar con la siguiente comanda, sbatch job.sub 64 -prof, podemos ver que por cada pragma acc parallel se genera un kernel. EL primer kernel se llama kmeans_243_gpu, hace referencia a la línea 243, donde se encuentra el pragma *#pragma acc parallel loop*. El segundo kernel denominado kmeans_229_gpu hace referencia a la línea 229, donde tenemos el pragma *#pragma acc parallel loop present(centroides[0:k])*. Observamos que el porcentaje de tiempo en la línea 243 es mucho mayor que en la línea 229, es un noventa y nueve porciento, casi gran parte del tiempo. Esto se debe a que el primer kernel realiza num_pixels iteraciones, ya que debe recorrer todos los centroides por cada píxel, mientras que el segundo kernel es más rápido debido a que realiza solo k iteraciones.



```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average      Minimum     Maximum           Name
 -------  ---------------  ---------  -----------  ----------  ----------  --------------------
    48,0    1.276.571.493      3.328    383.585,0         603   2.635.198  cuEventSynchronize
    48,0    1.266.991.970        980  1.292.848,0       1.700   2.888.353  cuStreamSynchronize
     0,0       24.919.213          1 24.919.213,0  24.919.213  24.919.213  cuMemHostAlloc
     0,0       24.201.873        980     24.695,0      10.638      45.515  cuMemcpyHtoDAsync_v2
     0,0       18.861.025      1.764     10.692,0       4.935      36.502  cuMemcpyDtoHAsync_v2
     0,0       16.441.617      3.330      4.937,0       1.942     257.104  cuEventRecord
     0,0        8.915.863        392     22.744,0      11.831      45.159  cuLaunchKernel
     0,0        1.422.532          8    177.816,0       4.418   1.053.122  cuMemAlloc_v2
     0,0          917.224          1    917.224,0     917.224     917.224  cuMemAllocHost_v2
     0,0           80.306          1     80.306,0      80.306      80.306  cuModuleLoadDataEx
     0,0           21.673          1     21.673,0      21.673      21.673  cuStreamCreate
     0,0           16.268          7      2.324,0         722       5.281  cuEventCreate


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances    Average     Minimum    Maximum        Name
 -------  ---------------  ---------  -----------  ---------  ---------  --------------
    99,0      227.735.029        196  1.161.913,0  1.161.416  1.162.760  kmeans_243_gpu
     0,0          463.558        196      2.365,0      2.335      2.528  kmeans_229_gpu


CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations    Average    Minimum   Maximum       Operation
 -------  ---------------  ----------  -----------  -------  ---------  ------------------
    50,0    1.830.291.422         980  1.867.644,0    1.087  2.720.242  [CUDA memcpy HtoD]
    49,0    1.780.568.802       1.764  1.009.392,0    1.152  2.623.825  [CUDA memcpy DtoH]
```

Código paralelizado OpenACC

A continuación se muestra la función kmeans modificada:

```
void kmeans(uint8_t k, cluster* centroides, uint32_t num_pixels, rgb* pixels){
    uint8_t condition, changed, closest;
    uint32_t i, j, random_num;

    //Reservem memòria
    uint32_t* media_r = malloc(k * sizeof(uint32_t));
    uint32_t* media_g = malloc(k * sizeof(uint32_t));
    uint32_t* media_b = malloc(k * sizeof(uint32_t));
    uint32_t* num_puntos = malloc(k * sizeof(uint32_t));

    uint32_t* closest_centroide = malloc(num_pixels*sizeof(uint32_t));

    printf("STEP 1: K = %d\n", k);
    k = MIN(k, num_pixels);

    // Init centroids
    printf("STEP 2: Init centroids\n");
    for(i = 0; i < k; i++)
```

```
    {
        random_num = rand() % num_pixels;
        centroides[i].r = pixels[random_num].r;
        centroides[i].g = pixels[random_num].g;
        centroides[i].b = pixels[random_num].b;
    }

    // K-means iterative procedures start
    printf("STEP 3: Updating centroids\n\n");
     i = 0;
    do
    {
        // Reset centroids i vectores auxiliares
        #pragma acc data copy(centroides[0:k], pixels[0:num_pixels],
closest_centroide[0:num_pixels])
        {
         #pragma acc parallel loop present(centroides[0:k])
         for(j = 0; j < k; j++)
          {
           centroides[j].media_r = 0;
            centroides[j].media_g = 0;
           centroides[j].media_b = 0;
           centroides[j].num_puntos = 0;
              media_r[j] = 0;
              media_g[j] = 0;
              media_b[j] = 0;
              num_puntos[j] = 0;
        }

          // Find closest cluster for each pixel
          #pragma acc parallel loop
          for(j=0; j<num_pixels; j++)
        {  //Per cada pixel,
            int16_t diffR, diffG, diffB;
            rgb* p=&pixels[j];
            uint32_t min = UINT32_MAX, distancia, centroide_actual;

            for (centroide_actual=0; centroide_actual<k; centroide_actual++){
              diffR = centroides[centroide_actual].r - p->r;
              diffG = centroides[centroide_actual].g - p->g;
              diffB = centroides[centroide_actual].b - p->b;
              distancia = diffR*diffR + diffG*diffG + diffB*diffB;

               if (distancia < min){
                 min = distancia;
                 closest_centroide[j] = centroide_actual;
               }
             }
          }
```

```
        }
        //printf("Centroides encontrados");
        // Assign colors to each cluster
        //#pragma acc parallel loop
reduction(+:media_r[0:k],media_g[0:k],media_b[0:k],num_puntos[0:k])
       for(j = 0; j < num_pixels; j++)
       {
        uint32_t closest_cluster = closest_centroide[j];
        media_r[closest_cluster] += pixels[j].r;
        media_g[closest_cluster] += pixels[j].g;
        media_b[closest_cluster] += pixels[j].b;
        num_puntos[closest_cluster] += 1;
       }

        // Update structs
        for (j=0; j<k; j++){
         centroides[j].media_r = media_r[j];
         centroides[j].media_g = media_g[j];
         centroides[j].media_b = media_b[j];
         centroides[j].num_puntos = num_puntos[j];
       }

        // Update centroids & check stop condition
        condition = 0;
        for(j = 0; j < k; j++)
        {
            if(centroides[j].num_puntos == 0)
            {
                continue;
            }

            centroides[j].media_r = centroides[j].media_r/centroides[j].num_puntos;
            centroides[j].media_g = centroides[j].media_g/centroides[j].num_puntos;
            centroides[j].media_b = centroides[j].media_b/centroides[j].num_puntos;
            changed = centroides[j].media_r != centroides[j].r || centroides[j].media_g !=
centroides[j].g || centroides[j].media_b != centroides[j].b;
            condition = condition || changed;
            centroides[j].r = centroides[j].media_r;
            centroides[j].g = centroides[j].media_g;
            centroides[j].b = centroides[j].media_b;
        }

      i++;
   }while(condition);

   // Alliberem la memòria reservada previament
   free(media_r);
   free(media_g);
   free(media_b);
```

```
    free(num_puntos);
    free(closest_centroide);
    printf("Number of K-Means iterations: %d\n\n", i);
}
```

Análisis rendimiento

A continuación, analizaremos el rendimiento de la versión secuencial y paralela. Esta última, la ejecutaremos tanto en la GPU de GeForce RTX 3080 como en la GeForce RTX 1080Ti. Las siguientes tablas muestran los tiempos de ejecución según el número de clústers k.

Tiempos de referencia - Versión Secuencial						
Versión: Seq -Ofast	K=2	K=4	K=8	K=16	K=32	K=64
Elapsed time	0,27	1,06	4,26	7,32	15,32	121,85
Iteraciones	9	22	51	47	49	196
Checksum	6405336	12689895	22329779	42559141	73490886	123269810

Tiempos de referencia - Versión Paralela							
Nº threads	Versión: OpenACC	K=2	K=4	K=8	K=16	K=32	K=64
GeForce RTX 3080	Elapsed time	0,7	1,41	3,2	2,83	2,94	11,21
GeForce RTX 3080 Ti		1,3	2,11	4,4	3,91	4,25	15,54
Iteraciones		9	22	51	47	49	196
Checksum		6405336	12689895	22329779	42559141	73490886	124269810

Speedup						
Nº threads	K=2	K=4	K=8	K=16	K=32	K=64
GeForce RTX 3080	0,39	0,75	1,33	2,59	5,21	10,87
GeForce RTX 3080 Ti	0,21	0,50	0,97	1,87	3,60	7,84

Tal y como podemos observar, el Checksum no varia por cada clúster creado. Es decir, en ambos programas se obtiene el mismo resultado por cada k. De igual forma, el número de iteraciones se mantiene constante. Por lo tanto, podemos afirmar que la paralelización mediante OpenACC es eficaz (se obtiene el resultado deseado).

Si analizamos el rendimiento de la GPU GeForce RTX 3080, el speedup es menor de 1 (speedup<1) cuando hay menos de 8 clústers (k<8). Esto significa que la aceleradora no aprovecha de manera eficiente su capacidad de procesamiento paralelo. En cambio, cuando k es mayor o igual a 8 (k>=8), se produce una aceleración (speedup>1), por lo que hay una cierta eficiencia en la ejecución de tareas paralelizables. Como resultado, observamos que la aceleración augmenta conforme incrementamos el número de clústers. Addicionalmente, se aprovecha la paralelización al incrementar k, concretamente cuando k>=8.

Si analizamos el rendimiento de la GPU GeForce RTX 1080Ti, vemos que no se produce una aceleración (speedup<1) al haber menos de 16 clústers (k<16). Por lo tanto, necesita una mayor k para aprovechar la paralelización. No solo eso, los valores de la aceleración son

menores respecto a la anterior GPU. Concluimos que, de igual forma que GeForce RTX 3080, la aceleración augmenta conforme incrementamos el número de clústers. Addicionalmente, se aprovecha la paralelización al incrementar k, concretamente k>=16. No obstante, esta segunda GPU obtiene una menor eficiencia en la ejecución de tareas paralelizables respecto la primera.

Si comparamos los valores obtenidos al evaluar el rendimiento de nuestras dos propuestas en los diferentes modelos de GPU, observamos como estos son menores que los tiempos de referencia dados en la práctica. Esto es debido a que hemos realizado más paralelismo, por lo tanto, obtenemos una solución más óptima, con una mayor aceleración.

Obstáculos y soluciones

Al realizar la práctica, tuvimos diversos problemas que finalmente pudimos solucionar.

El primero de todos fue al realizar la transferencia de datos entre la CPU i GPU. Al haber programado con OpenMP, olvidamos que la sintaxis de OpenACC requiere del 0 en el intérvalo de elementos al declarar la copia de un array. Nuestro pragma inicial era: #pragma acc data copy(centroides[:k], pixels[:num_pixels], closest_centroide[:num_pixels]). Al final, escribimos: #pragma acc data copy(centroides[0:k], pixels[0:num_pixels], closest_centroide[0:num_pixels]).

El segundo inconveniente sucedió al declarar el #pragma acc parallel loop al reiniciar els centroides. Debimos añadir la cláusula present para indicar a la GPU que no debía de volver a copiar los vectores previamente copiados. Queremos destacar su importancia ya que la transferencia tiene una latencia considerablemente mayor que no el cálculo de operaciones simples. Por lo tanto, queremos reducir al máximo su uso: #pragma acc parallel loop present(centroides[0:k]).

Para acabar, intentamos paralelizar la reducción de la media de colores de cada clúster. Sin embargo, como había que modificar mucho código y no dispusimos del tiempo necesario, no la realizamos. Creemos que es un buen punto de partida para la continuación de la práctica.

Conclusiones

Al paralelizar el código kmeanslib.c hemos aprendido cómo usar correctamente las cláusulas de la librería OpenAcc. No solo eso, hemos sido conscientes de que a veces no basta con paralelizar los bucles añadiendo pragmas y cláusulas. Muchas veces es necesario reescribir el código, como pasa al intentar paralelizar la reducción.

Además, hemos ejecutado el código paralelo modificando los recursos disponibles y el tamaño del problema. Al analizar los resultados, hemos verificado que la paralelización es correcta y hemos determinado que el mismo programa no es igual de eficiente en las dos aceleradoras disponibles: GeForce RTX 3080 y GeForce RTX 1080Ti.

En GeForce RTX 3080, kmeanslib.c paralelizado con OpenAcc es eficiente al establecer que los clústers creados son mayores o igual a 8. A diferencia de esta, en GeForce RTX 1080Ti sucede cuando se forman como mínimo 16 clústers. El motivo de este suceso es debido a que la GPU GeForce RTX 3080 ofrece características y rendimiento superiores en comparación con la GeForce RTX 1080Ti. La primera GPU tiene una arquitectura más reciente, más núcleos especializados y un mayor ancho de banda. Es por eso que aprovecha mejor la capacidad de procesamiento paralelo y obtiene una aceleración mayor para todos los valores de k.

Por último, hemos comprendido que un alto speed-up indica que la aceleradora obtiene una eficiencia significativa en la ejecución de tareas paralelizables, permitiendo completar las tareas de forma más rápida. En cambio, un speed-up bajo indica que la tarea no es adecuada para la ejecución en la GPU. El motivo podría ser la sobrecarga asociada con la transferencia de datos entre la CPU y la GPU o la sincronización entre hilos de ejecución de la aceleradora. Esto se puede observar en nuestro análisis de rendimiento, ya que la GPU con un menor ancho de banda obtiene una menor aceleración, por lo tanto, un peor rendimiento.

Concluimos que contra más “pragmas” añadamos, se realizará más paralelización y más óptimo será el programa, siempre y cuando las características del hardware (recursos) lo permita.