David Morillo Massagué (1666540)
Adrià Muro Gómez (1665191)

Pràctica de Laboratori 2 - OpenACC / CUDA

Anàlisi previ del codi:

Per a saber la part la qual hauríem de prioritzar a l'hora de realitzar millores al codi, vam seguir la llei d'Amdahl, i centrar-nos en la fracció de codi que més temps consumia en l'execució. Per a saber de quina es tractava, vam fer servir l'arxiu "job_gprof.sub", la qual executa i perfila el programa, i genera un informe ("informe_gprof.txt") indicant, per cada funció existent al codi, les vegades que es cridava i el temps total d'execució d'aquesta:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
74.10	4.72	**4.72**	**74649600**	0.00	**0.00**	**find_closest_centroid**
23.44	6.21	**1.49**	**1**	1.49	6.21	**kmeans**
2.05	6.34	**0.13**	**1**	0.13	0.13	**read_file**
0.63	6.38	**0.04**	**1**	0.04	0.04	**getChecksum**

Veiem en la taula generada que les funcions cridades un sol cop són les funcions de llegir l'arxiu, obtenir el checksum, i la funció principal de la llibreria (la qual crida a les anteriors citades).
La funció més cridada és "**find_closest_centroid**", que consumeix més d'un 74% de temps d'execució. Observem també que la funció, per cada crida, consumeix molt poc temps (4.72 segons/74649600 crides = 0.00000006 segons).
Determinem que el codi a canviar serà aquesta mateixa funció, o el bucle el qual la crida.
El codi de la funció **find_closest_centroid** és:

```
uint8_t find_closest_centroid(rgb* p, cluster* centroids, uint8_t num_clusters){
  uint32_t min = UINT32_MAX;
  uint32_t dis[num_clusters];
  uint8_t closest = 0, j;
  int16_t diffR, diffG, diffB;

  for(j = 0; j < num_clusters; j++)
  {
    diffR = centroids[j].r - p->r;
    diffG = centroids[j].g - p->g;
    diffB = centroids[j].b - p->b;
    dis[j] = diffR*diffR + diffG*diffG + diffB*diffB;

    if(dis[j] < min)
    {
      min = dis[j];
      closest = j;
    }
  }
  return closest;
```

Veiem que aquesta consta d'un bucle for per cada clúster, amb unes 3 restes, 3 multiplicacions, 2 sumes i un mètode simple per a determinar la distància mínima al clúster, per cada iteració. És important destacar que el número de clústers mai superarà els 64 en les nostres execucions. Tenint en compte això, distribuir i paral·lelitzar el treball del bucle entre fils per a cada crida (74649600 cops), no resultaria la millor opció per optimitzar la funció. Atès que la funció és relativament simple i amb poc marge de millora, decidim ajustar el codi (bucle) que la crida, i procedir a paral·lelitzar en la mesura del possible aquestes crides.

Estratègia de millora:

Paral·lelització de Bucles Amb OpenACC:

1. Per millorar el rendiment de l'algorisme K-Means, hem aplicat la paral·lelització del bucle principal mitjançant directrius OpenACC. Aquestes directrius permeten distribuir el càlcul de les iteracions del bucle entre diferents fils o processadors de la GPU, aprofitant així la capacitat de processament paral·lel. Amb aquesta paral·lelització, aconseguim una execució més ràpida i eficient de l'algorisme.

Atomic Update per Realitzar la Reducció:

2. Per a calcular les sumes totals de les variables **red**, **green**, **blue** i **points** per a cada clúster, hem utilitzat operacions atòmiques (atomic update) dins el bucle paral·lel. Això ens permet realitzar les operacions de suma de manera segura i coherent, evitant possibles conflictes de memòria quan múltiples fils accedeixen a les mateixes variables simultàniament.

Copia de Dades i Clàusules Per Informar la GPU de la Seva Ubicació:

3. Per assegurar que les dades necessàries per a l'algorisme es trobin a la memòria de la GPU i no a la memòria principal, hem utilitzat clàusules de transferència de dades com **present** o **create** en les directrius OpenACC. Aquestes clàusules indiquen a la GPU quines variables han de ser copiades a la seva memòria local abans de l'execució, garantint un accés més ràpid i eficient a les dades.

Amb aquestes estratègies de millora, hem aconseguit optimitzar significativament l'algorisme K-Means per a la seva execució en entorns paral·lels com ara GPUs, millorant el seu rendiment i escalabilitat.

David Morillo Massagué (1666540)
Adrià Muro Gómez (1665191)

Aplicació de la paral·lelització:

Present:

Aquesta clàusula s'utilitza per indicar al compilador que una variable ja es troba present a la memòria de l'accelerador. En el context de OpenACC, quan una variable està marcada com a present, el compilador no realitzarà cap còpia addicional de la variable a la memòria de l'accelerador abans de l'execució. En el cas de l'algorisme K-Means, la clàusula **present** s'utilitza per indicar que les variables **centroides** i **pixels** ja es troben a la memòria de la GPU abans de l'execució de les tasques paral·leles.

Create:

Aquesta clàusula s'utilitza per indicar al compilador que una variable ha de ser creada a la memòria de l'accelerador amb un valor inicial específic abans de l'execució. En l'algorisme K-Means, la clàusula **create** s'utilitza per crear noves variables **red**, **green**, **blue** i **points** a la memòria de la GPU abans de l'execució dels bucles paral·lels.

Copy:

Aquesta clàusula s'utilitza per indicar al compilador que una variable ha de ser copiada a la memòria de l'accelerador abans de l'execució. Durant la pràctica es copien les variables pixels i centroides perquè la GPU siguiu capaç d'utilitzar-les durant l'execució de la funció **kmeans.**

Atomic update:

Aquesta clàusula s'utilitza per realitzar operacions atòmiques sobre variables compartides entre fils en paral·lel. En l'algorisme K-Means, la clàusula **atomic update** s'aplica a les operacions de suma de les variables **red**, **green**, **blue** i **points** dins dels bucles paral·lels, garantint que les operacions de suma es realitzin de manera segura i coherent sense conflictes de memòria.

Parallel loop:

Aquesta clàusula s'utilitza per indicar al compilador que un bucle ha de ser paral·lelitzat i executat en paral·lel en l'accelerador. En l'algorisme K-Means, les clàusules **parallel loop** s'apliquen als bucles que iteren sobre els píxels de la imatge i els clústers, permetent una execució eficient i paral·lela del càlcul de la distància i l'assignació de píxels als clústers.

David Morillo Massagué (1666540)
Adrià Muro Gómez (1665191)

Amb aquestes clàusules i canvis, el codi resultant és el següent:

Per a crear i crear variables a la memòria de la GPU:

```
// K-means iterative procedures start
i = 0;
#pragma acc data create(red[0 : k], green[0 : k], blue[0 : k], points[0 : k])
copy(pixels[0 : num_pixels], centroides[0 : k])
   do
   {
         // Codi de les iteracions (while)
}
```

Per a canviar els valors de les variables (a la GPU) a 0 al principi de cada iteració:

```
// Reset centroids
#pragma acc parallel loop
      for (j = 0; j < k; j++)
      {
         red[j] = 0;
         green[j] = 0;
         blue[j] = 0;
         points[j] = 0;
      }
```

Implementant el codi de la funcio find_closest_centroid() directament en el bucle for per a afavorir la paral·lelització:

```
// Find closest cluster for each pixel
#pragma acc parallel loop present(red, green, blue, points, centroides, pixels)
      for (j = 0; j < num_pixels; j++)
      {
         uint32_t min_dist = UINT32_MAX;
         uint8_t closest = 0;
         for (uint8_t c = 0; c < k; c++)
         {

            int16_t diffR = centroides[c].r - pixels[j].r;
            int16_t diffG = centroides[c].g - pixels[j].g;
            int16_t diffB = centroides[c].b - pixels[j].b;
            uint32_t dist = diffR * diffR + diffG * diffG + diffB * diffB;
            if (dist < min_dist)
            {
               min_dist = dist;
```

```
                closest = c;
            }
        }

#pragma acc atomic update
        red[closest] += pixels[j].r;
#pragma acc atomic update
        green[closest] += pixels[j].g;
#pragma acc atomic update
        blue[closest] += pixels[j].b;
#pragma acc atomic update
        points[closest]++;
    }
```

Per a actualitzar centroides en paral·lel:

```
    // Update centroids & check stop condition
    condition = 0;
    int next_red, next_green, next_blue, changed;
#pragma acc parallel loop present(red, green, blue, points, centroides)
    for (j = 0; j < k; j++)
    {
        if (points[j] > 0)
        {
            next_red = red[j] / points[j];
            next_green = green[j] / points[j];
            next_blue = blue[j] / points[j];
            changed = (centroides[j].r != next_red || centroides[j].g != next_green ||
centroides[j].b != next_blue);
            if (changed)
                condition = 1;
            centroides[j].r = next_red;
            centroides[j].g = next_green;
            centroides[j].b = next_blue;
        }
    }
```

David Morillo Massagué (1666540)
Adrià Muro Gómez (1665191)

Mètriques obtingudes:

Hem elaborat unes taules per a visualitzar els valors de temps i speedup segons valors de k d'entre 2 i 64. Hem utilitzat el codi seqüencial modificat i compilat amb -Ofast per a medir el temps de la primera fila amb aquesta comanda:

```
gcc kmeans.c kmeanslib.c -Ofast -o executable
```

Per a la segona fila de valors, s'ha compilat amb:

```
nvc -acc=gpu -ta=tesla -Minfo=all -o executable kmeans.c kmeanslib.c -w
```

Per una millor comparació visual, hem canviat el color de les cel·les segons el valor d'aquestes:
Verd → Millor valor, Vermell → Pitjor valor:

Time elapsed (s) with RTX 3080						
Execució:	**Valor de K (clústers):**					
	2	4	8	16	32	64
Seqüencial	1.13	2.92	11.59	20.53	41.40	294.52
Accelerat	0.29	0.41	0.64	0.64	0.67	2.16

Observem, com era d'esperar, que com a més clústers, més temps execució obtenim. En el cas del codi seqüencial (modificat), veiem com el temps, és directament proporcional al nombre de clústers k, excepte en el cas de 64 clústers. Al fer servir la GPU veiem com el temps de k=8, k=16 i k=32 són pràcticament idèntics. Veiem també que el temps de la GPU no augmenta de forma lineal, com ho fa a la CPU, i creix molt lentament.

Speedup (respecte seqüencial modificat)					
Valor de K (clústers):					
2	4	8	16	32	64
3.90	7.12	18.11	32.08	61.79	136.35

En aquest gràfic podem observar realment l'acceleració que obtenim respecte al codi original. Veiem que, com a més clústers, el speedup es multiplica per 2 (o més en alguns casos). El que ens ha sorprès és que el speedup augmentava de manera considerable a mesura que augmenta el nombre de clústers. Això és un molt bon indicador de l'aplicació de la paral·lelització al nostre codi: com més augmentem el nombre de clústers, més speedup obtindrem respecte al temps que hauria trigat en seqüencial, i això dóna molta llibertat de cara a futures execucions amb imatges més grans, més quantitat d'elles o més clústers.