Aránzazu Miguélez
Emma Juanico

INFORME PRÀCTICA Kmeans OpenMP

Anàlisi de rendiment

Per tal d'analitzar la funció *kmeans* i poder millorar-ne el seu rendiment hem hagut de fer un profiling d'aquesta. Per tal d'identificar les zones crítiques del codi i centrar els nostres esforços d'optimització en aquestes àrees hem començat calculant el temps d'execució.

Temps en seqüencial executant al wilma:

```
Time to Kmeans is 47 seconds and 445157 microseconds
Checksum value = 73490886

 Performance counter stats for './kmeans_OpenMP.exe test 32 imagen.bmp':

        46.749,97 msec task-clock:u                     #    0,983 CPUs utilized
                0      context-switches:u               #    0,000 /sec
                0      cpu-migrations:u                 #    0,000 /sec
            6.143      page-faults:u                    #  131,401 /sec
  138.773.965.116      cycles:u                         #    2,968 GHz
  302.531.090.479      instructions:u                   #    2,18  insn per cycle
   13.976.359.003      branches:u                       #  298,960 M/sec
       15.024.933      branch-misses:u                  #    0,11% of all branches

     47,576120633 seconds time elapsed

     46,316048000 seconds user
      0,146542000 seconds sys
```

Per tal de conèixer les zones crítiques del codi i poder optimitzar-lo hem seguit la següent estratègia. Primer de tot hem hagut d'analitzar l'execució del codi, és a dir veure quina part del codi consumeix la major part del temps i recursos.

Per a fer això hem utilitzat la comanda perf record. A continuació mesurem els recursos més utilitzats i veiem que la funció kmeans és la que triga el 99%.

```
Samples: 187K of event 'cycles:u', Event count (approx.): 138347779550
Overhead  Command          Shared Object      Symbol
  99,78%  kmeans_OpenMP.e  kmeans_OpenMP.exe  [.] kmeans
   0,09%  kmeans_OpenMP.e  libc-2.28.so       [.] _IO_getc
   0,06%  kmeans_OpenMP.e  kmeans_OpenMP.exe  [.] read_file
   0,04%  kmeans_OpenMP.e  kmeans_OpenMP.exe  [.] getc@plt
   0,02%  kmeans_OpenMP.e  [unknown]          [k] 0xffffffffae600e70
   0,01%  kmeans_OpenMP.e  [unknown]          [k] 0xffffffffae600230
   0,00%  kmeans_OpenMP.e  [unknown]          [k] 0xffffffffae600b60
   0,00%  kmeans_OpenMP.e  [unknown]          [k] 0xffffffffae600000
   0,00%  kmeans_OpenMP.e  libc-2.28.so       [.] _IO_file_read
   0,00%  kmeans_OpenMP.e  libc-2.28.so       [.] _IO_file_underflow@@GLIBC_2.2.5
   0,00%  kmeans_OpenMP.e  ld-2.28.so         [.] do_lookup_x
   0,00%  kmeans_OpenMP.e  libc-2.28.so       [.] __strstr_sse2
   0,00%  kmeans_OpenMP.e  ld-2.28.so         [.] memset
   0,00%  kmeans_OpenMP.e  ld-2.28.so         [.] _dl_check_map_versions
   0,00%  kmeans_OpenMP.e  ld-2.28.so         [.] _dl_map_object
   0,00%  kmeans_OpenMP.e  ld-2.28.so         [.] _dl_sysdep_start
   0,00%  kmeans_OpenMP.e  ld-2.28.so         [.] _start
```

Seguidament analitzem la funció kmeans. Si ens fixem kmeans té un bucle de *num_pixels* iteracions i a cada iteració fa una crida a la funció *find_closest_centroid*. Aquesta funció si l'analitzem podem observar que itera sobre tots els centroides per calcular la distància entre el píxel donat i cadascun dels centroides, és a dir, també està construida amb un bucle for de *k* iteracions, així doncs, el bucle principal s'executarà *num_pixels* x *k* vegades, fet que fa augmentar la complexitat de l'algorisme.

Per tant, aquesta funció es bottleneck ja que és la que està consumint més temps i la que no ens està permitint paral·lelitzar el codi.

Modificacions

Per tant hem realitzat les següents modificacions en el codi.

Com s'ha esmentat prèviament a la primera implementació, la funció 'find_closest_centroid' calcula la distància euclidiana al quadrat entre cada píxel i tots els centroides, i després tria el centroide més proper en funció d'aquesta distància. Aquesta funció és invocada en un bucle dins de la funció principal 'kmeans'. D'aquesta manera no és possible la seva paral·lelització.

Això ho hem modificat a la segona implementació, la cerca del centroide més proper es realitza directament dins del bucle principal 'for' dins la funció kmeans. En lloc de calcular la distància per a tots els centroides i després triar el més proper, cada píxel calcula la seva distància a tots els centroides en paral·lel utilitzant directives d'OpenMP per millorar-ne el rendiment.

Seguidament a la segona implementació, hem utilitzat diversos vectors auxiliars (red, green, blue i points) per acumular les sumes dels valors de píxels assignats a cada centroide.

Els arrays auxiliars red, green, blue i points s'utilitzen per acumular els valors dels components RGB i el nombre de punts assignats a cada centroide durant l'assignació de píxels a clusters. Per tant, aquests vectors es fan servir per calcular els nous valors mitjans dels centroides abans de l'actualització final.

Quan es realitza l'assignació de píxels als centroides més propers en paral·lel, diversos threads poden intentar actualitzar els mateixos valors als centroides alhora. Això pot conduir a condicions de carrera, on el resultat final depèn de l'ordre d'execució dels threads. Per evitar-ho, cada thread utilitza el seu propi conjunt de variables red, green, blue i points per acumular els valors corresponents, ja que utilitzem la directiva *#pragma omp parallel for reduction(+:red[:k],green[:k],blue[:k],points[:k])*. D'aquesta manera cada thread té el seu propi espai de memòria per treballar i no hi ha interferència entre els ells. Per

aplicar la clàusula *reduction* hem hagut de utilitzar aquesta estructura de vectors auxiliars ja que si no no funcionava.

Càlcul de temps

WILMA						
TEMPS DE REFERÈNCIA						
Versió	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
Seq + Ofast	1,47	4,53	19,23	34,74	69,88	529,95
Iteracions	9	22	51	47	49	196
Checksum value	6405336	12689895	22329779	42559141	73490886	124269810

TEMPS DE REFERÈNCIA (codi seqüencial modificat) amb 1 thread						
Versió	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
Seq + Ofast	1,48	4,88	19,32	34,12	69,19	527,34
Iteracions	9	22	51	47	49	196
Checksum value	6405336	12689895	22329779	42559141	73490886	124269810

versió	Temps de referència					
OpenMp -Ofast	wilma					
num threads	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
2	0,77	2,46	10,13	17,67	35,70	265,58
4	0,48	1,36	5,29	9,09	18,02	133,87
6	0,38	0,97	3,61	6,23	12,15	89,29
8	0,34	0,79	2,75	4,75	9,23	67,91
10	0,31	0,67	2,27	3,87	7,48	54,62

12	0,30	0,63	1,99	3,31	6,39	46,37
Iteracions	9	22	51	47	49	196
Checksum Value	6405336	12689895	22329779	42559141	73490886	124269810

Speedup						
num threads	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
2	1,91	1,84	1,89	1,97	1,95	1,99
4	3,06	3,33	3,63	3,82	3,87	3,95
6	3,87	4,67	5,32	5,57	5,75	5,93
8	4,32	5,73	6,99	7,31	7,57	7,80
10	4,74	6,76	8,47	8,98	9,34	9,70
12	4,9	7,19	9,66	10,49	10,94	11,43

Eficència						
num threads	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
2	0,96	0,92	0,95	0,99	0,98	1
4	0,77	0,83	0,91	0,96	0,97	0,99
6	0,65	0,78	0,89	0,93	0,96	0,99
8	0,54	0,72	0,87	0,91	0,95	0,98
10	0,47	0,68	0,85	0,90	0,93	0,97
12	0,41	0,60	0,81	0,87	0,91	0,95

AOLIN						
TEMPS DE REFERÈNCIA						
Versió	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64

Aránzazu Miguélez
Emma Juanico

Seq + Ofast	0,26	1,036	4,25	6,94	14,72	115,96
Iteracions	9	22	51	47	49	196
Checksum value	6405336	12689895	22329779	42559141	73490886	124269810

TEMPS DE REFERÈNCIA (codi seqüencial modificat)						
Versió	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
Seq + Ofast	0,32	1,13	4,30	7,07	15,13	118,42
Iteracions	9	22	51	47	49	196
Checksum value	6405336	12689895	22329779	42559141	73490886	124269810

versió	Temps de referència					
OpenMp -Ofast	AOLIN					
num threads	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
2	0,37	0,6	2,29	3,75	8,26	60,96
4	0,11	0,36	1,27	2,03	4,42	34,5
6	0,106	0,29	0,97	1,49	3,14	23,12
8	0,10	0,29	1,11	1,86	3,90	29,01
10	0,088	0,25	0,91	1,47	3,26	24,17
12	0,080	0,22	0,77	1,28	2,67	20,16
Iteracions	9	22	51	47	49	196
Checksum Value	6405336	12689895	22329779	42559141	73490886	124269810

Speedup						
num threads	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64

2	0,7	1,72	1,85	1,85	1,78	1,9
4	2,36	2,87	3,37	3,4	3,33	3,36
6	2,45	3,57	4,38	4,65	4,68	5,05
8	2,6	3,57	3,82	3,85	3,77	3,9
10	2,95	4,1	4,67	4,72	4,51	4,79
12	3,25	4,7	5,51	5,42	5,51	5,75

Eficència						
num threads	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
2	0,35	0,86	0,92	0,92	0,89	0,95
4	0,59	0,71	0,84	0,85	0,832	0,84
6	0,4	0,59	0,73	0,77	0,78	0,84
8	0,325	0,44	0,48	0,48	0,47	0,48
10	0,29	0,41	0,46	0,47	0,45	0,48
12	0,27	0,39	0,45	0,45	0,49	0,47

Analitzant els temps ontinguts veiem que tant a Wilma com a aolin els canvis que hem proposat fan una millora del temps d'execució del programa. El speedup màxim que s'ha assolit és de 10,94x, el qual podem afirmar que és una millora molt significativa. Tot i això, podem veure que a mesura que augmenta la k, va també ho fa el speedup i l'eficència, és a dir, els canvis realitzats al codi són més notables quan la mida de les dades augmenta.

Si comparem aolin amb Wilma, podem veure que Wilma té speedups i eficiències majors en relació amb aolin, això pot ser causat principalment per les diferències entre els dos entorns de treball i la càrrega de treball a la que estan sotmeses al moment d'execució.

Observant els resultats del speedup i eficiència proporcionats per a cada nombre de fils en les dues versions, podem determinar quina versió demostra un millor aprofitament dels recursos de processament amb l'augment del nombre de fils i així podem saber quin dels dos és més escalable.

Aránzazu Miguélez
Emma Juanico

A Wilma els valors de speedup augmenten de manera consistent i l'eficiència es manté relativament alta (aproximadament 0,9) amb l'increment dels fils utilitzats. En canvi, a Aolin l'eficència es troba sempre entre 0,4 i 0,5, per tant això indica una millor escalabilitat Wilma. En altres paraules, Wilma és la versió que manté un rendiment consistent i eficient en relació amb el nombre de fils utilitzats i aleshores es considera més escalable, ja que fa un ús dels recursos més eficient. Això és degut al fet que Wilma té 12 cores i, per tant, està utilitzant millor tots els seus recursos, ja que a Aolin només hi ha 6 cores.