Blai Ras, Albert Morales i Àlex Campos

Programació Paral·lela

Pràctica 1b: Conversió a espai de color amb CUDA

Objectius

A la pràctica 1a vam ens van introduir a la programació paral·lela amb *OpenMP*. Per fer-ho, se'ns va proposar paral·lelitzar un algorisme que convertia una imatge en format de color *bgr* a *rgba*, on 'a' correspon al canal 'alpha'. Bàsicament aquest algoritme recorre la imatge amb dos fors niats reasignant posicions de memòria.
Ara, l'objectiu és paral·lelitzar el mateix algoritme però fent ús de CUDA, una plataforma d'Nvidia que permet fer ús de la nostre GPU per dur a terme operacions en paral·lel mitjançant una mescla de llenguatge C i C++.
Finalment, es demana una comparació en termes d'optimització i eficiència entre *OpenMP* i CUDA.

Implementeu una versió bàsica de l'algorisme en CUDA, amb el doble bucle for

Per a començar la pràctica se'ns proporciona un arxiu en format CUDA (.cu) molt similar al main.cpp de la pràctica anterior. De fet, un .cu no és res mes que un *.cpp* amb crides a funcions pròpies de CUDA, de control i gestió de la GPU.
El codi proporcionat té fetes les *allocations* de memòria, les comprovacions de si la conversió s'ha realitzat correctament i els *free's*. També té la crida a la funció convertBGR2RGBA juntament amb una predisposició d'una *grid* i blocs inicials. Ara bé: aquesta funció convertBGR2RGBA no té calculada la posició de cada *thread*.
Aleshores, per a poder completar l'objectiu, vam començar pensant en les dimensions de la nostra graella. Si volem realitzar un doble bucle, hem de treballar en la dimensió 'x' i 'y'. Ara bé, de quina mida han de ser aquestes coordenades? Doncs si sabem que depenen de la manera en com hem repartit els blocs, agafem l'amplada de la imatge (WIDTH) i la dividim entre la coordenada 'x' del bloc, mentre que en la segona coordenada (eix de les 'y'), agafem l'alçada de la imatge (HEIGHT) i la dividim per la coordenada 'y' del bloc. Fem ús de la funció *ceil* per agafar el número enter més proper al resultat de la divisió (les dimensions de la graella no poden ser decimals, clar!)
A continuació, per tant, vam repassar les dimensions del nostre bloc. En el codi base estan definides com (512,1,1). Ara fem ús de les dos dimensions, per tant, canviarem segur el valor de la segona coordenada. Llegint en la teoria del curs, sabem que una GeForce GTX 1060 té un màxim de 1024 *threads* per *block*. Ara estem treballant amb una de semblant, la GPU de l'Àlex, una 1070. El número màxim és de 1024 també. Per tant, el valor de la primera coordenada multiplicat pel valor de la segona coordenada multiplicat pel valor de la tercera coordenada no pot superar 1024. Això vol dir que màxim podem establir 32 en l'eix de les 'x' i 32 més en l'eix de les 'y'. Així doncs, establim una mida de *block* de (32,32,1).

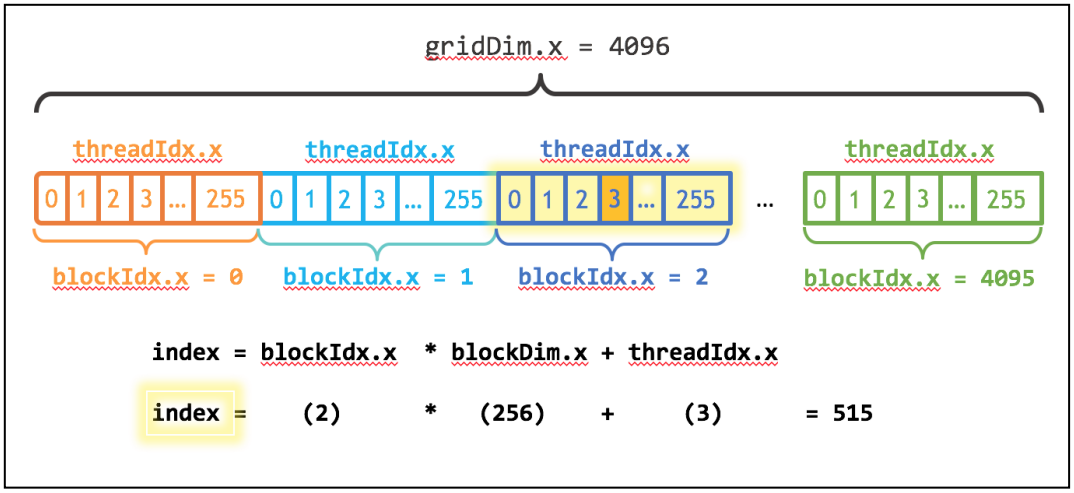
En tercer lloc ens movem a la funció convertBGR2RGBA. Fins ara hi ha un únic enter '*position*' a zero, que sabem que s'ha de canviar. Volem realitzar dos fors, per tant, sabem segur que hi haurà d'haver dues posicions. Sabem gràcies a la teoria que la posició de cada *thread* està determinada a CUDA com *threadIdx.x* + *blockIdx.x* * *blockDim.x*. Així doncs, creem un enter *positionx* igualat a aquesta operació i un segon enter '*positiony*' igualat a la mateixa operació però amb la coordenada 'y', és a dir, *threadIdx.y* + *blockIdx.y* * *blockDim.y*. Ara tenim dos enters que indiquen la posició de cada *thread* de cada coordenada, però nosaltres necessitem un enter que ens indiqui la posició del vector RGBA!
Podem obtenir aquesta posició en relació a l'alçada o l'amplada de la imatge. Per exemple, agafem l'amplada. Tirarem "alçada" vegades l'enter *positiony* i després ens desplacem *positionx* vegades. El resultat serà la posició del vector RGBA correcte. Podem fer el mateix en relació l'alçada: multiplico *positonx* per l'altura i em desplaço *positony* vegades. La funció queda així:

```
49  __global__ void convertBGR2RGBA_2for(uchar3 *bgr, uchar4* rgba, int width, int height) {
50
51      int positionx = threadIdx.x + blockIdx.x * blockDim.x;
52      int positiony = threadIdx.y + blockIdx.y * blockDim.y;
53      //int position = positiony * WIDTH + positionx;
54      int position = positionx * HEIGHT + positiony;
55
56      //printf("GPU - i = %d, j = %d\n", positionx, positiony);
57      // Protection to avoid segmentation fault
58      if (positionx < width ||  positiony < height) {
59          rgba[position].x = bgr[position].z;
60          rgba[position].y = bgr[position].y;
61          rgba[position].z = bgr[position].x;
62          rgba[position].w = 255;
63      }
64  }
```

Podem canviar la mida de bloc sempre i quan no ens passem de 1024, com per exemple, (8,8,1): 16 *threads* per *thread block*.

Implementeu una versió bàsica de l'algorisme en CUDA, amb un sol bucle:

Ara ens plantegem modificar el codi anterior per realitzar la iteració amb un sol bucle, és a dir, en una sola dimensió (la de les 'x'). En aquest cas el codi proporcionat ens és més útil: la *grid* sabem que ha de tenir les coordenades de les 'y' i 'z' en 1 i la primera coordenada anirà en proporció de la altura i amplada de la imatge. Concretament, tal i com surt al codi base, multipliquem altura per amplada i la dividim per la coordenada 'x' del *block*.
Aquest bloc sabem també que la coordenada 'y' i 'z' han de ser 1, i que la coordenada 'x' no pot superar els 1024 *threads* per *thread block*. Inicialment ens ve amb 512, un número que la majoria de GPU's poden suportar.
La funció *convertBGR2RGBA* només pateix una modificació: canviar la igualació de '*position*' a *threadIdx.x* + *blockIdx.x* * *blockDim.x*, és a dir, agafar la posició de cada *thread* tal i com diu CUDA que s'ha de fer o la nostra teoria.



La funció *convertBGR2RGBA* queda així llavors:

```
__global__ void convertBGR2RGBA_1for(uchar3 *bgr, uchar4* rgba, int width, int height) {

    int position = threadIdx.x + blockIdx.x * blockDim.x;
    //printf("GPU - i = %d, j = %d\n", positionx, positiony);
    // Protection to avoid segmentation fault
    if (position < width * height) {
        rgba[position].x = bgr[position].z;
        rgba[position].y = bgr[position].y;
        rgba[position].z = bgr[position].x;
        rgba[position].w = 255;
    }
}
```

**Hem creat dues funcions convertBGR2RGBA, una que realitza la iteració en dos fors (convertBGR2RGBA_1for) i l'altre que ho fa en dos (convertBGR2RGBA_2for).*

Intenteu optimitzar les accessos a memòria d'alguna manera bàsica, sense utilitzar shared memory.

En aquest apartat hem realitzat una millora d'accés a memòria (millora temporal). Recordem que entenem com a millora temporal com aquella modificació que permet reduir els accessos a memòria, un fet que comporta un major temps d'execució.
Concretament, hem creat dues variables temporals anomenades "*tempbgr*" i "*temprgba*". La primera és una copia d'un tros del vector "*bgr*", mentre que la segona és un vector del mateix tipus que el vector "*rgba*" (*uchar4*). La realització d'aquesta "duplicació" de variables és una millora temporal d'accés a memòria perquè són copies dels vectors reals però que es troben "a la memòria cache", una memòria que com sabem és de ràpid accés i d'emmagatzemat de dades temporal.
Aleshores, un cop tenim les variables creades, procedim a fer el mateix que fa "*convertBGR2RGBA*" però usant aquestes variables temporals, és a dir, assignem els valors "x", "y", "z" del vector "*tempgbr*" a la variable instanciada "*temprgba*". En aquesta última també li assignem el valor 255 corresponent a la *alpha*.
Ara el que tenim és una variable temporal amb el resultat de la conversió, així que només falta igualar el vector original a la posició en qüestió a ella.

Aquesta modificació l'hem reutilitzada de la pràctica anterior. El seu ús no es veu afectat pel fet de 'fer servir' CUDA, és a dir, ara la posició es calcula fent ús del *id* del *thread*, el *block* on està i la dimensió d'aquest, però la creació de variables temporals segueix sent possible:

```
64  __global__ void convertBGR2RGBA_optBasic(uchar3 *bgr, uchar4* rgba, int width, int height) {
65
66      int position = threadIdx.x + blockIdx.x * blockDim.x;
67      uchar3 tempbgr = bgr[position];
68      uchar4 temprgba;
69      // Protection to avoid segmentation fault
70      if (position < width * height) {
71          temprgba.x = tempbgr.z;
72          temprgba.y = tempbgr.y;
73          temprgba.z = tempbgr.x;
74          temprgba.w = 255;
75          rgba[position] = temprgba;
76      }
77  }
```

Finalment, hem realitzat una segona modificació que es podria considerar millora o no. El que hem fet és *unroll the loop* (https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf), és a dir, fer la conversió a RGBA "de dos en dos", causant així el doble de feina per a cada *thread* però reduint a la meitat el nombre d'iteracions. Ho veiem més clar en codi:

```
79  __global__ void convertBGR2RGBA_optBasic2(uchar3 *bgr, uchar4* rgba, int width, int height) {
80
81      int position = 2*(threadIdx.x + blockIdx.x * blockDim.x);
82
83      if (position < width * height) {
84          rgba[position+0].x = bgr[position+0].z;
85          rgba[position+1].x = bgr[position+1].z;
86
87          rgba[position+0].y = bgr[position+0].y;
88          rgba[position+1].y = bgr[position+1].y;
89
90          rgba[position+0].z = bgr[position+0].x;
91          rgba[position+1].z = bgr[position+1].x;
92
93          rgba[position+0].w = 255;
94          rgba[position+1].w = 255;
95      }
96  }
97
```

A l'últim apartat podem veure que els temps són majors, però.
**Hem creat dues funcions més convertBGR2RGBA, una que realitza la primera optimització (convertBGR2RGBA_optBasic) i l'altre que realitza la segona (convertBGR2RGBA_ optBasic2).*

Optimitzeu els accessos a memòria utilitzant shared memory

L´ús de CUDA ens obra les portes a la *shared memory*, una memòria localitzada a dins del *chip* (no com la memòria global o la de *Device*). Aquesta ens interessa perquè és molt ràpida: està a prop dels nuclis de càlcul i no té cap mecanisme d'organització de dades (la gran diferència amb les memòries *caché*). Tal i com el seu nom indica, aquesta memòria està compartida per *threads* d'un mateix *thread block*, de manera normalment s'usa per l'accés a memòria global de manera coalescent.

En aquest cas, nosaltres fem ús d'ella per guardar el vector *bgr* i el vector *rgba*. Concretament:

1. Es creen dues memòries compartides de mida *blockDim*.
2. Es calcula *l'id* de cada *thread*
3. Es calcula la posició segons *l'id* de cada *thread*, *l'id* de cada *block* i la dimensió d'aquest
4. Si aquesta posició no està fora de rang (altura multiplicat per amplada)
5. A la posició *threadIdx.x* de la memòria compartida *bgr*, guardo el contingut del vector *bgr* (situat en la memòria global) en la posició calculada prèviament.
6. Espero a que tots els *threads* acabin amb la funció *syncthreads();*
7. Faig la conversió a RGBA: guardo a la memòria compartida *d'rgba* en la posició *threadIdx.x* el contingut de la memòria compartida de *bgr* en la posició *threadIdx.x*. En la quarta posició, l'*alpha*, hi guardo 255, però.
8. Espero a que tots els *threads* acabin amb la funció *syncthreads();*
9. Guardo al vector *rgba* de la memòria global el contingut del vector *rgba* de la memòria compartida en la posició *threadIdx.x*

De manera que la funció convertBGR2RGBA_shared, en un cas de *blockDim* = 512, queda així:

```
87  __global__ void convertBGR2RGBA_shared(uchar3 *bgr, uchar4* rgba, int width, int height) {
88
89      extern __shared__ uchar3 shared_bgr[512];
90      extern __shared__ uchar4 shared_rgba[512];
91      int tid = threadIdx.x;
92      int position = threadIdx.x + blockIdx.x * blockDim.x;
93
94      if (position < width * height) {
95          shared_bgr[tid] = bgr[position];
96
97          __syncthreads();
98
99          shared_rgba[tid].x = shared_bgr[tid].z;
100         shared_rgba[tid].y = shared_bgr[tid].y;
101         shared_rgba[tid].z = shared_bgr[tid].x;
102         shared_rgba[tid].w = 255;
103
104         __syncthreads();
105
106         rgba[position] = shared_rgba[tid];
107     }
108 }
```

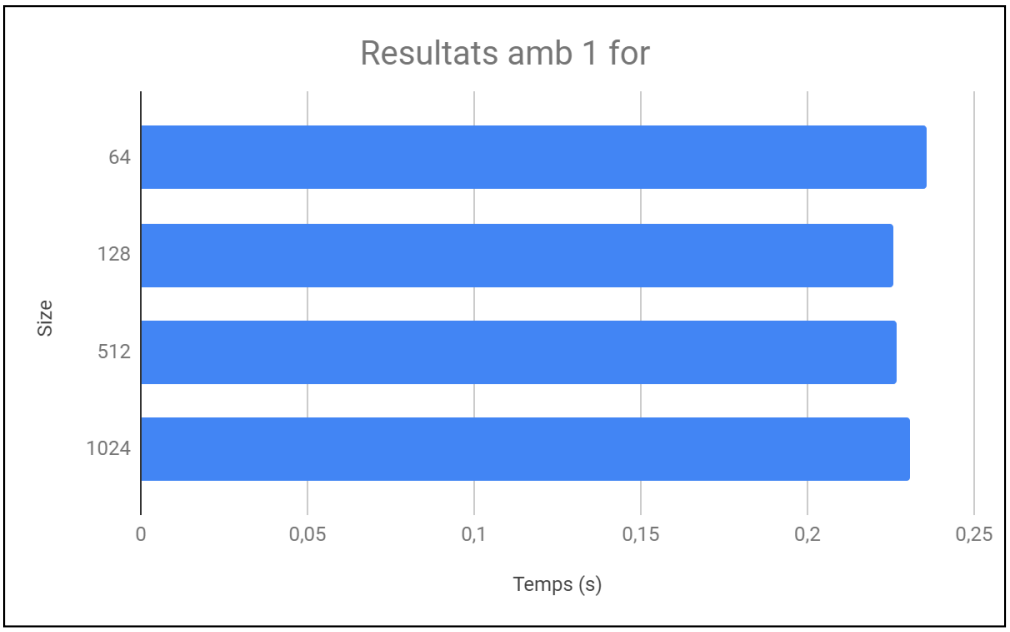
**Hem creat una funcions més convertBGR2RGBA per probar la memòria compartida: convertBGR2RGBA_ shared.*

Com a la pràctica anterior, creeu un informe, i en aquest cas, compareu el rendiment de la millor versió en serie i la millor versió amb OpenMP, amb les versions que feu amb CUDA.

A l'hora d'analitzar el temps d'execució del programa, hem decidit fer ús de la funció *multitime* (https://tratt.net/laurie/src/multitime/) la qual és una extensió de la funció *time* indicada a teoria on li pots indicar el número d'execucions que vols realitzar de cert executable (en aquest cas, el nostre codi). El resultat són 5 columnes: la mitjana de temps d'execució, la seva desviació mitjana, la execució que menys ha trigat, la mediana i la execució que més ha trigat, tan per temps "real" com d'usuari o de sistema. Aquí un exemple:

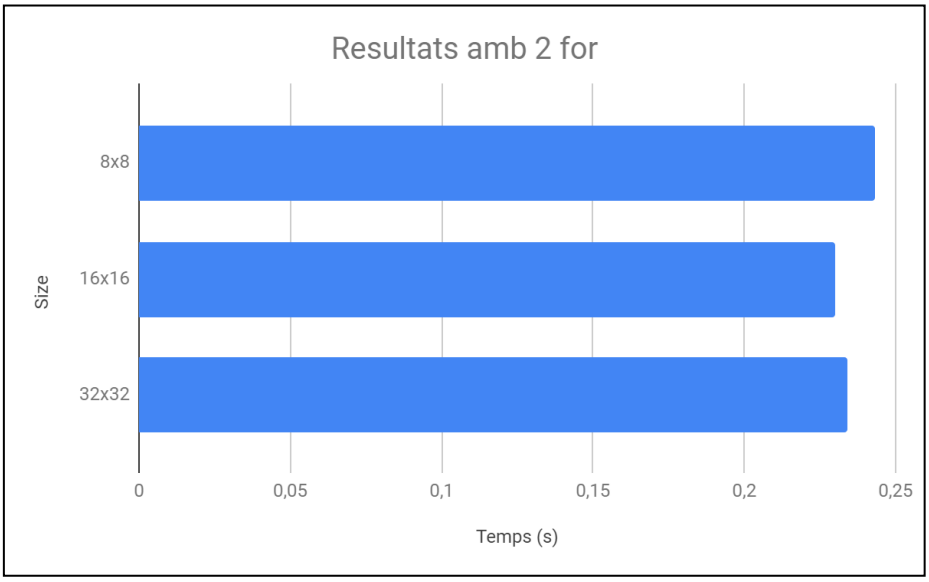
shared 512→	Mean	Std.Dev.	Min	Median	Max
real	0.234	0.020	0.202	0.239	0.295
user	0.131	0.017	0.103	0.129	0.187
sys	0.097	0.009	0.076	0.097	0.123

D'aquesta manera, podem tenir una millor aproximació del temps mitjà que triga el nostre codi sense haver d'executar-lo manualment. Concretament, totes les mitjanes de temps d'execució que veurem a continuació són extretes executant el programa 100 vegades.
En primer lloc, veiem els resultats de l'algoritme amb un *for*. Ho fem en relació a la mida de bloc sempre:

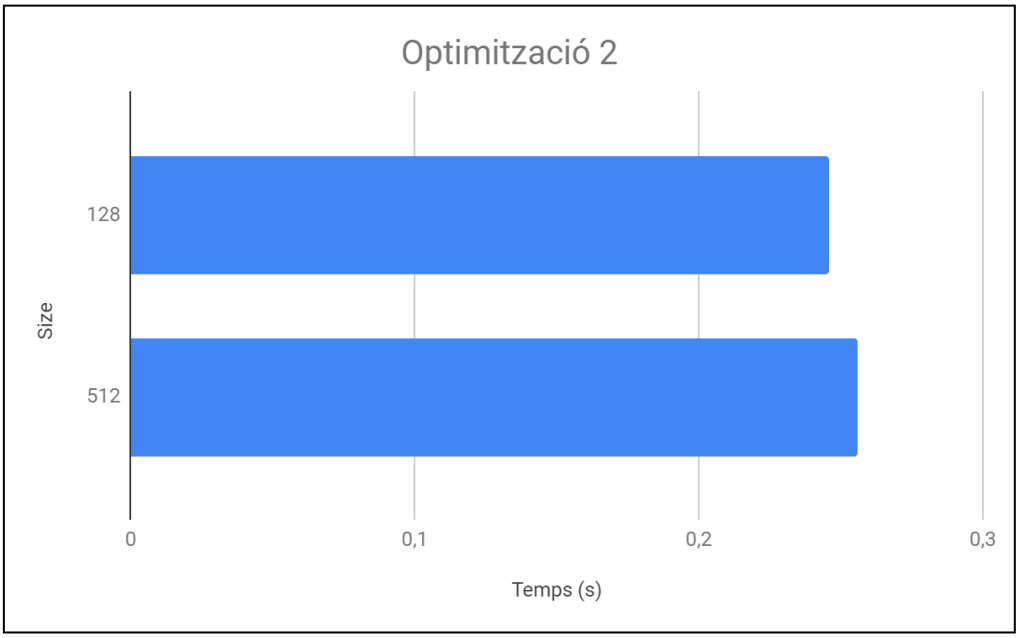


El temps mínim és de 0,226 segons i es troba amb una mida de bloc de 128. Tal i com podem veure, amb una mida massa petita (64) o massa gran (1024) el temps s'incrementa lleugerament.

En segon lloc, mirem la primera optimització realitzada (variables temporals) amb dos *for*:

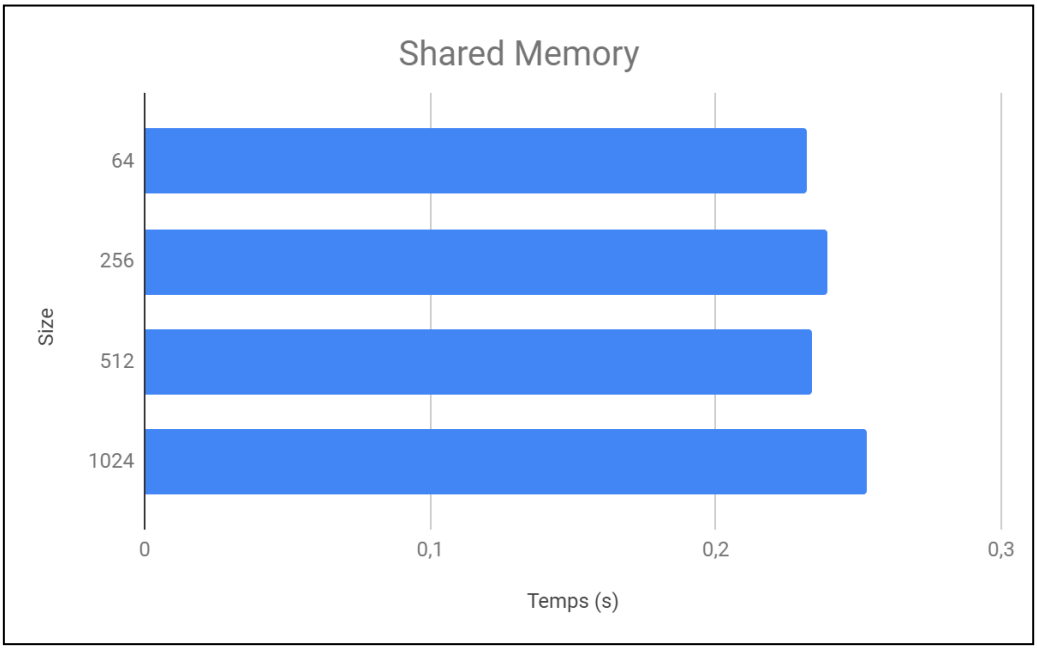


Curiosament, amb dos for ens triga més. Concretament, el mínim és de 0,23 i s'aconsegueix amb una mida de bloc de 16x16. És una diferència minúscula respecte al d'un *for*, però ens esperàvem una mica més de millora.
A continuació mirem la segona optimització, on fem la meitat d'iteracions amb el doble de feina:



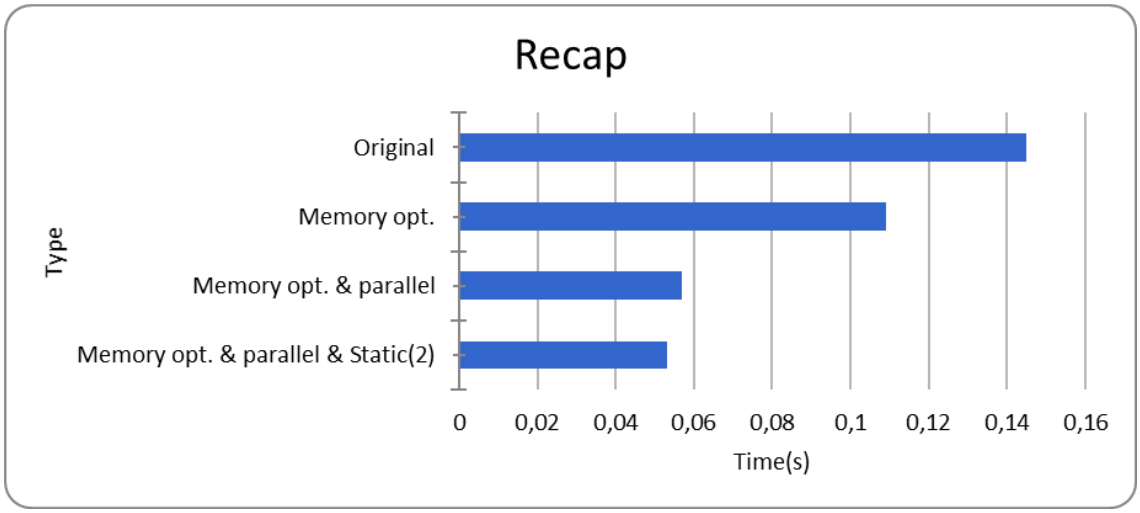
El temps mínim és de 0,246 i s'aconsegueix amb una mida de bloc de 128. És una diferència molt gran respecte als resultats anteriors, així que podem dir que convé més realitzar més iteracions on es treballa menys que fer-ne menys on es treballa més.

Ara passem a veure els resultats amb *shared memory*:



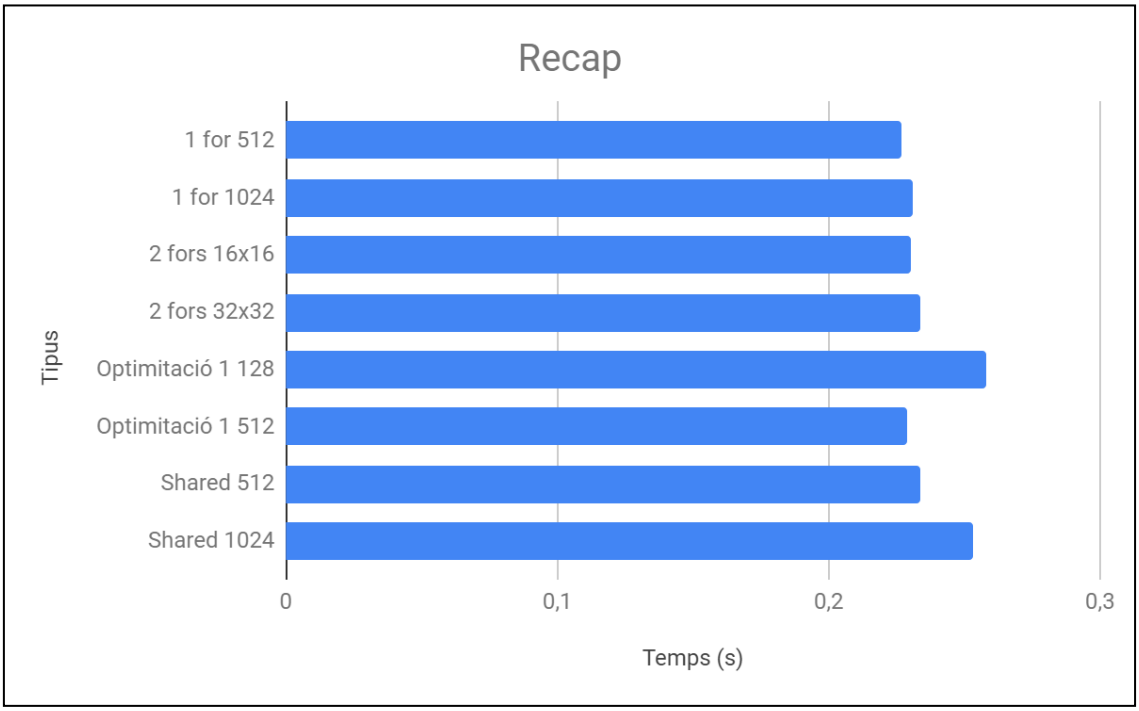
Fent ús de memòria compartida aconseguim un temps mínim de 0,232 segons amb un bloc de mida 64. La diferència amb la mateixa funció sense memòria compartida és nomes de 0,006 segons. Un altre cop, ens esperàvem més optimització.

Un cop hem vist els resultats de l'optimització de la funció amb CUDA, és hora de comprar-los amb la pràctica anterior, on fèiem ús *d'OpenMP* amb millores de localitat de dades i *scheduling*. En resum, els resultats de la pràctica anterior eren els següents:



On el temps més òptim aconseguit era de 0,053 segons.

Ara, ho podem comparar amb el resums de CUDA:



D'entre tots els tests, el mínim es troba en un for amb mida de bloc 512, on el temps mínim és de 0,227. La diferència de temps si ho comparem amb *OpenMP* és doncs de 0,173 segons. En definitiva, aquets mals resultats pensem que poden venir de:

1. Estem usant la GPU per fer una operació "poc complexa", és a dir, perdem més temps en "fer-la servir" que fen els càlculs en sí.
2. La realització de la memòria compartida no té sentit si no s'implementa la lectura coalescent de memòria. L'ús de la memòria compartida en el nostre cas no ens ofereix una millora significativa, així que si no s'implementa "amb un objectiu concret" potser no val la pena.

**Pots trobar tots els tests realitzats amb més valors dins del fitxer d'Excel adjuntat anomenat "tests"*