Anàlisi d'acceleració d'algoritmes de clustering mitjançant targetes gràfiques

TREBALL FINAL DE GRAU

Joan Orrit Palau
DIRECTOR: MARC GONZÁLEZ TALLADA | DEPARTAMENT D'AC
DATA DEFENSA: 29 JUNY DEL 2017
ESPECIALITAT EN ENGINYERIA DE COMPUTADORS
FACULTAT D'INFORMÀTICA DE BARCELONA
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Índex de continguts

Índex de figures

Índex de taules

1 Context

1.1 Introducció

Avui en dia, amb l'aparició d'Internet i l'avenç de les tecnologies en general, hom està compartint i administrant les seves vides a través del món online, així com el món està incrementant el seu espai digital. Això ha provocat que es puguin fer recol·lectes massives de dades, el qual a generat grans conjunts massius de dades; el que avui en dia coneixem com a *Big Data*. Analitzar conjunts de dades tant grans és molt costós, i ha provocat la necessitat d'usar tècniques complexes de paral·lelització, per tal d'optimitzar els algoritmes i programes que tracten aquestes dades. Tècniques englobades dins el que és conegut com a *High Perfomrance Computing*, al qual ens referirem com a HPC, i que són un conjunt de tècniques que ens permetrà analitzar aquests grans conjunts de dades. Per tal d'utilitzar les tècniques d'HPC, és necessari disposar d'una infraestructura física molt potent i costosa. És aquí on va nàixer el que avui en dia coneixem com a *Cloud Computing*. Aquesta és una metodologia que permet a clients usar màquines similars a les que podem trobar en un *data center* per tal d'executar algoritmes costosos, amb la possibilitat de fer-ho de forma virtualitzada a través d'internet.

1.1.1 *Big Data*

Des de principis del segle passat *[1]*, amb l'entrada d'Internet al nostre dia a dia, i amb la invenció de l'emmagatzematge digital, l'anàlisi i l'emmagatzematge d'informació s'ha accelerat de forma exponencial. Hem de tenir en compte que si agafem totes les dades recollides fins a l'any 2000, no arribarien a igualar les recollides avui en dia en tant sols un minut *[2]*. Dades com poden ser compres que els usuaris fan a través d'internet, missatges de veu o text que enviem, cerques que fem en navegadors d'Internet o dades que generem a través de les xarxes socials, entre molts d'altres exemples. Aquesta suma de dades fa que, per exemple, moltes empreses o científics, tinguin moltes dades a analitzar, ja sigui per predir models o generar estadístiques, i necessitin d'eines per poder-ho fer. És aquí on entra el ja esmentat HPC

1.1.2 *High Performance Compuitng* (HPC)

Per tal d'analitzar aquests conjunts de dades massives s'han creat i s'estan millorant dia a dia les tècniques anomenades de *High Performance Compuntin* (HPC). Es tracta de la utilització de súper computadors i el tècniques de processament en paral·lel, per resoldre problemes computacionals complexes *[3]*. Aquesta

tècnica per tant, és molt adequada a l'hora d'analitzar grans magnituds de dades com és el cas del *Big Data*. En aquest projecte estudiarem una forma d'utilitzar la tècnica d'HPC, utilitzant una eina anomenada *CUDA*. Aquesta eina va ser desenvolupada per la casa *Nvidia* a l'any 2006 *[4]* i ens permet paral·lelitzar algoritmes complexes mitjançant targetes gràfiques (GPU) i obtenir un elevat *performance* en la seva execució. A part d'eines de paral·lelització per tal de tractar aquests conjunts massius de dades, també hi ha molts algoritmes (els quals s'intenta optimitzar a través de tècniques com són les de l'HPC) que ens permeten analitzar les mateixes dades. Per aquest projecte s'han triat els algoritmes de *Clustering.* Aquests algoritmes ens permeten, donat un *data-set* de punts a l'espai sense etiquetar, ordenar els mateixos, de forma automàtica, en diferents subgrups o clústers. Aquests són utilitzats tant en sectors com són el de la indústria tèxtil, com en d'altres com ho poden ser els de la medicina o la logística. Bàsicament ens permeten, donades moltes dades, analitzar-les i aconseguir treure estadístiques o fer prediccions sobre aquestes.

Un dels problemes principals d'aquests algoritmes és que, tal i com diu pròpia definició d'HPC, necessiten utilitzar súper computadors, i això és complicat d'aconseguir si no disposem d'accés a una màquina amb la potència suficient (targetes gràfiques potents entre d'altres) per tal d'executar aquests tipus d'algoritmes. És aquí on va nàixer el que coneixem com a *Cloud Computing*.

1.1.3 *Cloud Computing*

En el món interconectat d'avui en dia i un cop, no només era necessari que grans científic o centres de recerca treballessin amb grans magnituds de dades, sinó que també o necessitaven les més petites empreses, va sortir el que avui en dia anomenem com a *Cloud Computing*. Fa una dècada, proveïdors d'Internet, Google *[5]* o Amazon amb el seu *Amazon Web Services [6]* van començar aquest servei. Bàsicament el que oferien era que els clients poguessin usar un proveïdor extern per tal d'executar aquests algoritmes d'alt cos, sense necessitat de ser propietaris de la infraestructura d'aquestes màquines. Tant sols tenint com a cost el lloguer que comporta usar una màquina d'aquestes. El que ofereixen serveis com aquests han estat una autèntica revolució, ja que permeten fer tot allò que el client duria a terme des d'un *data center*, des de casa seva i només tenint el cost del servei que s'utilitzi en cada moment.

1.2 Motivació

Amb tots aquests canvis i revolucions que estem vivint i que, especialment, el món de la informàtica està vivint, no podíem més que fer una recerca sobre totes aquestes tecnologies i, a menys introduir-nos en la majoria d'aquests sectors vists fins ara. És per això que hem escollit estudiar el algoritmes de *clustering*, amb els quals podrem estudiar una part de les tècniques d'HPC; a més de sotmetre els mateixos a analitzar grans magnituds de dades, la qual cosa, ens permetrà simular que treballem en un projecte de *Big Data*.

Per fer-ho tindrem accés a màquines de la Facultat d'Informàtica de Barcelona, i no ens serà necessari contractar cap servei de *Cloud Computing*. Aquest, per tant serà un punt on no aprofundirem degut a aquesta disposició de màquines suficientment potent per tractar els algoritmes de *clustering* que estudiem en aquest projecte.

1.3 Actors implicats

1.3.1 Desenvolupador

Per tal d'analitzar com *CUDA* afecta als diferents algoritmes, s'hauran d'analitzar el comportament de *CUDA* en cada un dels codis, i treure o posar millores per tal d'analitzar el rendiment de les mateixes. El desenvolupador del projecte serà l'encarregat de dur a terme aquest anàlisi dels algoritmes de *Clustering* i d'implementar i fer els pertinents canvis a les diferents parts dels algoritmes per tal d'analitzar com *CUDA* influeix en el *performance* de les seves execucions. En ell recaurà la responsabilitat de de l'èxit o fracàs del projecte i, en qualsevol cas, la justificació de qualsevol d'aquests casos. A més d'aquesta implementació, haurà de testejar que els algoritmes segueixin funcionant de la mateixa manera en cada un del canvis que s'afegeixi per tal dur a terme l'anàlisi

1.3.1 Director

El director d'aquest projecte, Marc González Tallada, tindrà la tasca d'assessorar i guiar al desenvolupador del projecte, així com triar els diferents enfocaments i algoritmes que el projecte encararà i estudiarà. El director també tindrà la tasca de supervisar el correcte avenç del projecte que vagi duent a terme el desenvolupador d'aquest.

1.3.2 Usuaris

- **Directes:** Els usuaris que es beneficiaran i utilitzaran directament els resultats d'aquest projecte seran tots aquells que, per les seves investigacions, empreses, estudis de mercat; ja siguin metges, biòlegs, economistes... utilitzin els algoritmes de *Clustering,* i que puguin obtenir un estalvi considerable del seu temps, degut a l'anàlisi exhaustiu que farem de la paral·lelització dels algoritmes
- **Indirectes:** Els usuaris que es beneficiaran de forma indirecta dels resultats d'aquest projecte seran tots aquells usuaris que utilitzin aquells productes, serveis, estudis... Que es duguin a terme gràcies als algoritmes de *Clustering* i que, gràcies a aquest projecte, podran arribar a un millor ús de CUDA envers els algoritmes de *Clustering*; probablement de forma més acurada i amb millors resultats.

2 Estat de l'art

En aquest apartat es mirarà de revisar la literatura del tema objecte d'estudi. Es farà una cerca acurada de referències que ens aportin coneixement sobre el tema i, es farà una comprensió i síntesi del coneixement trobat.

2.1 Algoritmes de Clustering

Tal i com defineix *Andrew Ng*, professor de Ciències de la Computació per la Universitat d'*Stanford*, els algoritmes de *Clustering*, tenen la funcionalitat de; donat un *data-set* de punts, ordenar aquests en diferents clústers o grups que siguin coherents per aquests mateixos punts *[7]*.

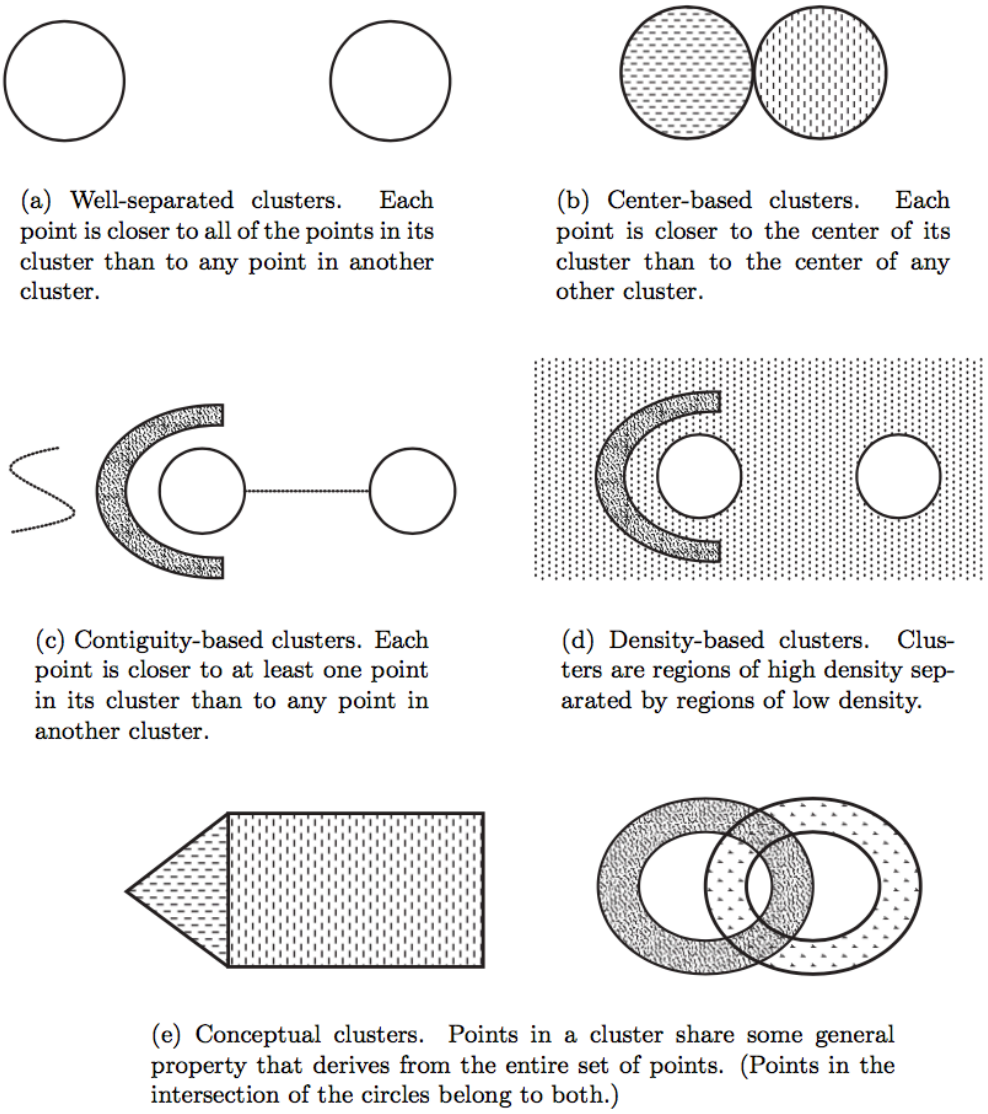
2.1.1 Diferents tipus d'algoritmes per tractar diferents tipus de clústers

En aquest apartat analitzarem els algoritmes de *Clustering* que s'han considerat més rellevants i estudiarem per quins clústers són més adients cada un d'ells. Aquests han sigut investigats i molt ben explicats pel professor de la Universitat de *Minnesota; Vipin Kumar [8].*

a) Tipus de clústers

Aquest punt fa referència a la figura 1 on podrem veure els tipus de clústers analitzats.

- ***Well-Separated clústers*:** Cada punt està a prop de qualsevol punt del seu clúster, abans que de qualsevol altre punt d'un altre clúster.
- ***Prototype-Based*:** Cada punt està més a prop del centre del seu clúster que del de qualsevol centre de qualsevol altre clúster.
- ***Graph-Based*:** Cada punt està, com a mínim, més a prop d'un punt del seu clúster que d'un punt de qualsevol altre clúster
- ***Density-Based*:** Aquests clústers compleixen la mateixa definició que els *Graph-Based* emperò, obviant aquells punts considerats menyspreables o *outliers.*
- ***Shared-Propierty*:** Els punts del clúster comparteixen alguna propietat amb els punts del seu clúster. Cal parar atenció i tenir en compte la seva similitud amb els clústers del tipus *Prototype-Based.*



(a) Well-separated clusters. Each point is closer to all of the points in its cluster than to any point in another cluster.

(b) Center-based clusters. Each point is closer to the center of its cluster than to the center of any other cluster.

(c) Contiguity-based clusters. Each point is closer to at least one point in its cluster than to any point in another cluster.

(d) Density-based clusters. Clusters are regions of high density separated by regions of low density.

(e) Conceptual clusters. Points in a cluster share some general property that derives from the entire set of points. (Points in the intersection of the circles belong to both.)

Figura 1. Diferents tipus de clústers analitzats al projecte

b) Tipus d'algoritmes de Clustering

En aquest apartat estudiarem els diferents tipus d'algoritmes de *Clustering* i analitzarem en quins clústers funciona millor cada un d'ells. Aquests també han sigut estudiats i són explicats de forma acurada i rigorosa pel professor *Vipin Kumar* de la Universitat de *Minnesota [8].*

- ***K-centers:*** Aquest algoritme ens agruparà els punts en K clústers essent K igual al número de clústers que elegim posar. L'algoritme, començarà, inicialment, agrupant tots els punts en un mateix clúster. Un cop fet això, prendrà el punt més llunyà al centre del primer clúster; aquest punt serà el centre del següents clúster. L'algoritme farà una reassignació de punts fins a obtenir els dos clústers. El *k-centers*, prosseguirà així fins a obtenir els k centres.

- ***K-means*:** El següent algoritme funciona de forma molt similar al k-centers però de forma més elaborada i complexa. Cosa que ens donarà millors resultats. Inicialment calcula k clústers de forma amb els centres aleatoris. Després calcularà la mitjana de tots els punts de cada clúster i, cada mitjana, serà un nou centre de cada clúster. Un cop fet aquest pas, el *k-means* assigna de nou els punts als clústers amb els centres més propers als mateixos punts. Aquest pas es va repetint les vegades que siguin necessàries fins aconseguir un resultat el més acurat a la realitat possible. És un bon algoritme a fer servir quan tenim els clústers de la forma *Well-Separtated* (veure figura 1).
- ***Agglomerative Hierarchical Clustering*:** Aquest algoritme ens calcula els diferents clústers que tenim en el nostre *data-set* de punts i ens acabarà construint un graf. És un algoritme potent quan ens interessa connectar clústers: aquest aspecte ens pot servir per analitzar distribucions de punts com les del *Prototype-Based* (veure figura 1).
- ***DBSCAN*:** Funciona de forma molt similar que l'*Agglomerative* Hierarchical vist amunt però amb la qualitat que podrà identificar *outliers*. Per tal de trobar els *outliers* segueix el següent procediment: defineix uns punts com a *core* del clúster; aleshores tots aquells punts que no s'acostin suficient a les normes del *core*, l'algoritme decideix que no és un bon punt. Un bon *data-set* per aquests punts seria el *Density-Based* (veure figura 1).

2.1.2 Algoritmes utilitzats

L'àmbit del *clustering* és de gran abast i ha sigut estudiat per moltes universitats. Professors com els citats anteriorment; Andrew Ng ***[7]***, de la Universitat de *Stanford* o el professor *Vipin Kumar [8]*, de la Universitat de *Minnesota*, han fet grans estudis i avenços en el món de la computació dels algoritmes de clustering i no tindria, per tant, sentit començar aquest projecte de recerca des de zero i sense aprofitar tota aquesta feina. Per fer-ho s'han utilitzat vídeos o cursos oberts *on-line* dels professors citats així com s'han buscat algoritmes ja fets sobre els mateixos.

Després d'una recerca acurada i rigorosa es va trobar, a través de la plataforma *Campaign*, alguns dels algoritmes de *Clustering* més importants ja fets i preparats per executar *[9]*. Aquesta plataforma és suportada pels professors: *Kai J. Kolhlhoff*, *Marc Sosnick*, *Bill Hsu*; de les Universitats de *Stanford* i de San Francisco. Els algoritmes trobats a la plataforma també havien estat provats d'accelerar a través de la GPU i la CPU i es podien utilitzar. S'ha decidit treballar, en aquest projecte, amb aquesta plataforma i provar primer d'entendre com han estat construïts i accelerats aquests algoritmes i, finalment, provar en quins punts són aquests més ràpids i, especialment, quines optimitzacions els fan més ràpids.

Aquesta plataforma consta de diversos algoritmes que ja, prèviament, han sigut accelerats mitjançant *CUDA*. Pel que fa aquest projecte, ens centrarem en els algoritmes que aglomeren els conceptes més importants en el món del *clustering*; el *k-centers* i el *k-means*. Cal dir que el *k-centers*, és un pas previ al *k-means* i que, per tant, ens centrarem especialment en el darrer.

Com conseqüència d'aquest apartat, podem dir que, en el problema que abastim en aquest projecte, analitzarem i adaptarem una solució ja existent per comptes de construir-ne una de nova.

2.1.3 *CUDA* i la programació amb *GPU*

CUDA és una plataforma de computació paral·lela i un model de programació que permet fer servir la GPU de forma simple, creada per *Nvidia* *[10]*. La plataforma de CUDA, ens permet tenir accés directe a la GPU i a les instruccions en paral·lel de la mateixa, per tal d'aconseguir, executar i computar els diferents *kernels* que li enviem.

L'eina CUDA, ha estat dissenyada per funcionar amb llenguatges tals com; C, C++ i *Fortran*. Aquests són els més utilitzats amb l'eina i, pel que ens envolta en aquest projecte, utilitzarem el llenguatge C++.

A continuació es descriuen tres dels conceptes més importants de CUDA: el *kernel*, l'anomenada *Shared Memory* i , finalment, els accessos *coalesced*.

a) Els *kernels* a CUDA

Els *kernels*, són una de les parts més importants que tenim en CUDA. És el punt de la nostra aplicació on s'executarà el codi de forma paral·lela a la nostra GPU. Bàsicament un *kernel* és un tros de codi que s'executa igual per tots el *threads* i de forma única per cada un d'ells *[11]*. Al *kernel*, li hem d'indicar les dimensions del número de blocs de *threads* que hi haurà, el nombre de threads per cada un dels blocs que voldrem i, finalment, la *Shared Memory*, la qual explicarem amb detall en els punts següents.

Per tal de declarar un *kernel* ho farem de la següent manera:

kernel<<<*grid*, TPB, SM>>>(*params*);

El *grid* seran les dimensions de la quadrícula de blocs que tindrem, TPB indicarà el nombre de threads que tindrem a cada un dels blocs i, finalment, SM indicarà la quantitat de *Shared Memory* , en bytes, que tindrem

per bloc. És important saber triar una bona relació entre blocs i *threads* per bloc segons el problema que ens escaigui.

Per últim, és important saber que cada un dels *threads* sap quin identificador de *thread* té, a més de saber l'identificador del bloc on està i les seves dimensions. En afegit, també sap la informació sobre les dimensions del tota la quadrícula de blocs (el *grid*).

b) La *Shared Memory* a CUDA

La memòria compartida és una de les optimitzacions que CUDA ens permet fer als nostres codis. Aquesta memòria és de molt més ràpid accés que la memòria global (la qual és visible per tots els *threads*). L'inconvenient que té, és que només és visible pels threads d'un mateix bloc i, la mateixa memòria, només existirà, mentre el mateix bloc existeixi . És per això que, al invocar al *kernel*, podem indicar la *Shared Memory* que necessitarem per cada bloc.

Quan vulguem fer servir memòria compartida haurem, abans de res, copiar les dades que ens interessin, la memòria compartida. Simplement haurem de declarar un espai de memòria com a *Shared Memory*, i copiar-hi la les dades que vulguem analitzar de la memòria global. Per a declarar un vector de *Shared Memory ho farem:*

```
__shared__ array[128];
```

El qual ens declararia un vector de 128 bytes de memòria compartida en cada bloc de threads on s'executi el kernel.

Un altre dels conceptes importants en memòria compartida, és la sincronització entre *threads* quan hi estem escrivint. Com que tots el *threads* del mateix bloc, accediran a la mateixa memòria compartida i, en la majoria de casos, cada *thread* escriu una part de la memòria global a la *shared;* quan un *thread* hagi escrit a la memòria compartida, haurà d'esperar que tots els altres *thread* també ho facin. Sense això podria succeir que hi haguessin *threads* que llegissin de posicions de memòria compartida en les que encara no hi hauria sigut copiada la global. Per dur a terme aquesta sincronització CUDA ens ofereix la següent crida que haurem d'utilitzar pràcticament sempre que utilitzem l'anomenada *Shared Memory [12]*:

```
__syncthreads();
```

Per acabar aquest apartat, dir que aquest serà un dels aspectes que estudiarem en aquest projecte; com la memòria compartida millora el nostre rendiment en els aquells codis que han sigut accelerats amb CUDA.

c) Els accessos *coalesced* a CUDA

La forma en com accedim a memòria és una de les part més importants de CUDA i que podrà determinarà el *performance* que obtindrem en l'execució dels nostres codis. Quan un thread accedeix a una posició de memòria, agafa la línia o bloc d'aquella posició de memòria. Això és un factor vital a l'hora de programa i que, si l'explotem; podrem obtenir uns guanys considerables.

La idea principal és aconseguir que quan un thread accedeix a una posició de memòria, els threads del mateix bloc, accedeixin a posicions contigües a aquesta primera posició. Si per comptes que cada *thread* accedeixi a posicions aleatòries, aconseguim que cada un d'ells ho faci tenint en compte les posicions on accediran els altres *threads*, obtindrem una elevada millora en el rendiment *[13]*.

És el que es coneix com a accessos *Coalesced* i, com hem dit, és clau en la programació de CUDA. Per tant el programador ha de muntar unes estructures de dades, a les quals sigui possible accedir-hi de forma *Coalesced*.

Aquest punt serà una altre, junt amb el de *Shared Memory*, que estudiarem en aquest projecte. Analitzarem de quina manera afecta el accedir o no de forma *Coalesced* en les execucions dels algoritmes de *Clustering*.

d) Les reduccions a CUDA

Les operacions de reducció ens permeten, partint d'un conjunt d'elements, reduir aquest en un sol element. Algunes d'aquestes operacions, són la suma, el producte, el màxim d'un conjunt d'elements... Aquest tipus d'operacions, si es paral·lelitzen, se'n pot obtenir una millora logarítmica respecte del que seria l'execució seqüencial. És en aquest punt on podem explotar aquest paral·lelisme tot usant CUDA.

S'ha de tenir en compte que no totes les operacions de reducció es poden paral·lelitzar, haurem de mirar aquelles que siguin associatives; com ho són la suma, el màxim d'un nombre o el producte; per contra, operacions com la resta o la divisió, no són operacions de reducció associatives i no es poden paral·lelitzar.

Quan aconseguim paral·lelitzar una operació de reducció s'obté un cost en l'execució de l'algoritme de*[14]*:

$$O(\log n)$$

En aquest projecte veurem algunes d'aquestes operacions de reducció paral·lelitzades amb CUDA i estudiarem, encara que no la pròpia operació directament, com hi afecta el fet d'utilitzar en elles memòria compartida o global.

e) Accions de la CPU quan programem amb CUDA

Hi ha 3 accions bàsiques que la CPU pot fer en relació amb la targeta gràfica. Aquestes són: reservar memòria a la GPU, per tal que aquest pugui analitzar les dades que li enviem, copiar memòria de la CPU a la GPU i, finalment, copiar-la de la GPU a la CPU.

Com forma d'exemple, suposem que volem operar a la GPU amb tres vectors; a[N], b[N] i deixar els resultats al vector c[N]. Per fer-ho, tal i com podem veure a la figura 2 *[15]* haurem de fer servir la funció *cudaMalloc()*, per tal de reservar espai en la GPU, la funció *cudaMemCopy()*, amb el paràmetre *cudaMemcpyHostToDevice*, per tal de copiar els vecotr a i b a la GPU i, finalment, amb la darrera funció, però aquest cop usant com a paràmetre, *cudaMemcpyDeviceToHost*, per copiar el vector c resultant a la CPU.

```
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;

// reservem espai a la GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

/* aquí inicialitzaríem els vector a i b */

HANDLE_ERROR( cudaMemcpy( dev_a,
                          a,
                          N * sizeof(int),
                          cudaMemcpyHostToDevice));
HANDLE_ERROR( cudaMemcpy( dev_b,
                          a,
                          N * sizeof(int),
                          cudaMemcpyHostToDevice));
HANDLE_ERROR( cudaMemcpy( dev_c,
                          a,
                          N * sizeof(int),
                          cudaMemcpyHostToDevice));

/* invoquem al kernel de CUDA que suma a i b */
add<<<128,128>>>(dev_a, dev_b, dev_c);

// copiem el vector resultant c cap a la CPU
HANDLE_ERROR( cudaMemcpy( dev_c,
                          a,
                          N * sizeof(int),
                          cudaMemcpyDeviceToHost));
```

Figura 2. Accions de la CPU quan programem amb CUDA.

3 Abast

3.1 Definició de l'abast

El projecte consistirà en agafar alguns dels algoritmes de la plataforma *Campaign* esmentada damunt, i en base a aquesta començar la nostra investigació. Els algoritmes elegits, degut a la utilització d'alguns dels *kernels* més típics en el món del *clustering* a l'hora de paral·lelitzar-se, han estat el *k-means* i el *k-centers*.

En primer lloc haurem de comprendre aquests dos algoritmes de la plataforma així com executar-los i provar-los. Un cop fet això tocarà entendre la paral·lelització que s'ha fet en els codis tot estudiant-les i entenent-ne la seva lògica.

Un cop fet aquesta recerca, podrem començar a analitzar amb quines millores CUDA funciona millor. És sabut que emprant la ja explicada *Shared* Memory i garantint els accessos *Coalesced*, de forma que els accessos siguin consecutius, CUDA funciona molt millor. El que ens proposem d'estudiar en aquest projecte, és quin impacte tenen en cada un dels algoritmes de la plataforma *Campaign,* aquestes optimitzacions de CUDA. Amb el fi d'aconseguir l'objectiu de provar quines són les optimitzacions més importants a tenir en compte quan paral·lelitzem algoritmes utilitzant la *API* de *CUDA.*

Com a conclusió, podem dir que: si s'aconsegueix comprendre i entendre els algoritmes, a més d'implementar els canvis necessaris en els algoritmes *k-means* i *k-centers* de la plataforma *Campaign*, per tal d'analitzar com els hi afecten les optimitzacions de memòria compartida i d'accessos *Coalesced;* els objectius del projecte seran complerts. Com a resultat podrem saber quin *performance* obtenen aquests dos algoritmes a l'hora d'utilitzar aquestes dues optimitzacions, a més de que, per part del desenvolupador, hi haurà un aprenentatge molt important en la utilització de CUDA.

3.3 Els algoritmes tractats: *k-centers* i *k-means*

Tant el *k-centers* com el *k-means*, tal i com és la característica dels algoritmes de *Clustering*, tenen la finalitat d'agrupar els algoritmes que li passem en els clústers que més els hi convinguin. A continuació veurem amb detall com funcionen aquests dos algoritmes ja que, com s'ha explicat amb anterioritat, seran els dos que tractarem en aquest projecte. Per fer-ho ens centrarem en els algoritmes ja accelerats mitjançant la GPU; ja que són els que estudiarem i la seva lògica, encara que funciona en paral·lel, i per tant més ràpid, és la mateixa que en l'execució seqüencial.

a) *K-centers*:

La idea principal de l'algoritme, és agafar els K punts, essent K el número de clústers amb els que vulguem agrupar el nostre *data-set* de punts, que més distin entre ells, i a aquests assigna'ls-hi els punts que hi siguin més propers.

D'aquesta manera tindrem, els centres del clústers el més separats possible i, a aquests, els hi assignarem els punts que els hi siguin més propers. Un cop ha fet això el *k-centers* para i no itera més vegades per tal d'obtenir un resultat més acurat. Això, fa que el *k-centers* sigui un algoritme amb no molta precisió.

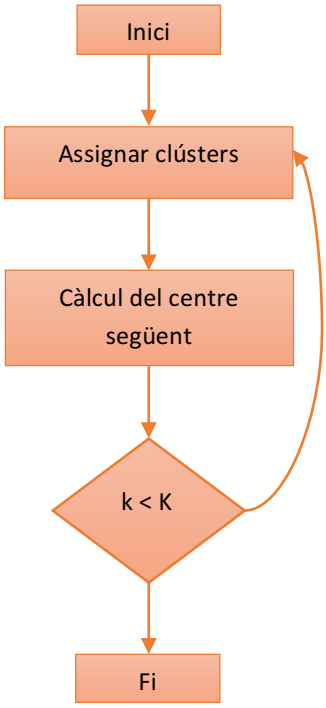


Figura 3. Algoritme k-centers

Com podem veure a la figura 2, el *k-centers* és un algoritme iteratiu que consta de 3 parts principals. En primer lloc assigna els punts al centre actual de la iteració. Després, en la mateixa iteració, fa un càlcul de quin serà el centre següent a l'actual, és a dir; calcula el punt més llunyà a l'actual. Finalment comprova si s'ha fet la iteració que pertoca per a cada clúster i, en cas de ser així acaba. Sinó, seguirà iterant fins que la k actual sigui igual a la k total.

b) K-means

L'algoritme *k-means*, funciona amb més precisió que el *k-centers*. Aquest algoritme, és també un algoritme iteratiu que pararà quan la solució sigui prou bona i ja no es pugui acotar més.

L'algoritme disposa de 3 paràmetres inicialment; N, K i D. On són, respectivament, el número de punts a tractar, el número de clústers i les dimensions que cada un dels punts tindrà. Amb aquests paràmetres l'algoritme, inicialment, començarà assignant un centre a cada clúster de forma aleatòria. La idea principal de l'algoritme és, per cada punt i de forma paral·lela, assignar-lo al centre del clúster que tingui més a prop. Cada *thread* s'encarregarà d'un punt, i calcularà la distància als centres del K clústers del nostre input. D'aquestes distàncies, n'escollirà la mínima i serà el clúster del centre escollit el que s'assignarà al punt que aquell *thread* estigui tractant. A partir d'aquí, a l'inici de cada una de les següents iteracions, el *k-means*, farà una mitjana aritmètica dels punts de cada clúster, calculant així, de nou, un centre que s'adapti més als K clústers que tinguem. Després d'això, seguirà fent reassignacions dels punts a cada un dels clústers fins que consideri que la solució sigui suficientment bona.

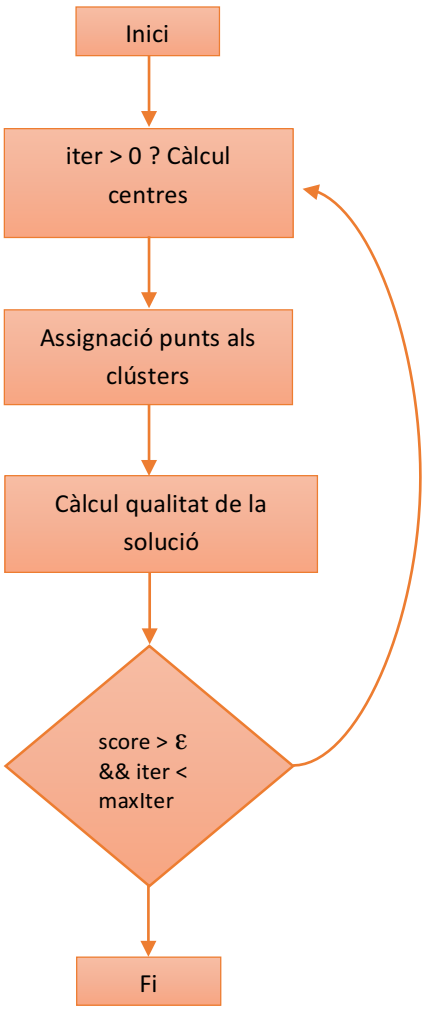


Figura 4 . Algoritme k-means

Tal i com podem veure a la figura 3, al final de cada iteració, l'algoritme calcula un *score* per tal d'analitzar si la solució és suficientment bona. Bàsicament calcula, en paral·lel, la distància a la que està cada punt del centre del clúster al que ha sigut assignat. Si l'*score* actual dista de l'anterior en un número molt proper a zero, sigui aquest ε, el *k-means* para d'iterar tot considerant que la solució ja és suficientment bona.

Per tant, com em vist, en cada iteració de l'algoritme hi ha 3 parts principals: la re-assignació de centres (sempre i quan no ens trobem en la primera de les iteracions), l'assignació dels punts en el clúster de centre més proper i, finalment, el càlcul de la qualitat de la solució. Cada una d'aquestes parts es fan, per a cada un dels punts N, en paral·lel i, per tant, les tenim a dins un *kernel*. A continuació veurem amb detall cada un d'aquests *kernels*.

- ***calcCentroids_CUDA*:**

  Aquest és el primers dels *kernels* que s'executa a cada iteració. Com s'ha explicat anteriorment, només s'executa a partir de la segona iteració. El *kernel* conté un *grid* d'una dimensió de K blocs; de manera que els *threads* de cada bloc s'encarregaran de calcular els centres d'un únic clúster. Tots els blocs analitzen tots els punts, però només els calculen la seva mitjana, si l'identificador del clúster coincideix amb l'identificador del bloc.

```
while (offset < N)
{
    // thread divergence likely
    if (ASSIGN[offset] == k)
    {
        // update centroid parts
        s_centerParts[tid] += X[d * N + offset];
        // increment number of elements in cluster
        if (d == 0) s_numElements[tid]++;
    }
    // move on to next segment
    offset += blockDim.x;
}
```

Figura 5. Assignació de només un clúster per cada bloc

  Podem veure això en el tros de codi del mateix *kernel* a la figura 4. Veiem que a dins hi ha un condicional que només permet acumular el valor del punt per després fer-ne la mirjana si: ASSIGN[offset] == k.

  Una altra de les parts claus del *kernel*, és la reducció que fa per tal de sumar tots els elements de cada una de les dimensions d'un clúster. Utilitza la suma com a operació de reducció, i aconsegueix un millor logarítmica a l'hora de sumar el conjunt de cada dimensió dels punts de cada clúster.

```
template <unsigned int BLOCKSIZE, class T>
__device__ static void reduceOne(int tid, T *s_A)
{
    if (BLOCKSIZE >= 1024) { if (tid < 512) { s_A[tid] += s_A[tid + 512]; }
__syncthreads(); }
    if (BLOCKSIZE >=  512) { if (tid < 256) { s_A[tid] += s_A[tid + 256]; }
__syncthreads(); }
    if (BLOCKSIZE >=  256) { if (tid < 128) { s_A[tid] += s_A[tid + 128]; }
__syncthreads(); }
    if (BLOCKSIZE >=  128) { if (tid <  64) { s_A[tid] += s_A[tid +  64]; }
__syncthreads(); }

    if (tid < 32)
    {
        if (BLOCKSIZE >= 64) { s_A[tid] += s_A[tid + 32]; }
        if (BLOCKSIZE >= 32) { s_A[tid] += s_A[tid + 16]; }
        if (BLOCKSIZE >= 16) { s_A[tid] += s_A[tid +  8]; }
        if (BLOCKSIZE >=  8) { s_A[tid] += s_A[tid +  4]; }
        if (BLOCKSIZE >=  4) { s_A[tid] += s_A[tid +  2]; }
        if (BLOCKSIZE >=  2) { s_A[tid] += s_A[tid +  1]; }
    }
}
```

Figura 6. La suma com a operació de reducció.

A la figura 5, podem veure aquesta reducció utilitzant la suma. Un cop tots el *threads* han executat la reducció, la suma quedarà reduïda en la primera posició del vector que es passa com a referència. Finalment, l'algoritme divideix cada una de les sumes de les dimensions de cada clúster en el número total d'elements de cada clúster (número que també troba mitjançant una reducció com la de la figura 5). Per així obtenir el nou centre de cada clúster mitjançant la mitjan aritmètica.

- ***assignToClusters_KMCUDA*:**

  Aquest *kernel*, calcula la distància d'un punt a cada un dels centres dels K clústers, i assigna el clúster que tingui el seu centre més proper al punt que està tractant. Per tant va iterant per cada clúster i calcula la distància entre el punt que tracta i el centre del clúster. A final de cada iteració, si el punt es troba més proper al clúster actual, es fa una re-assignació al mateix clúster que s'està tractant.

  És important tenir en compte que, al calcular les distàncies, els accessos són *coalesced*, de manera que així els accessos que fa un *thread*, pot ser reaprofitat per el altres *threads*, ja que es força que accedeixin a posicions properes. El que es fa és ordenar els punts a memòria per dimensions, de manera que primer tindrem tots els valors de la primera dimensió, després tots els de la segona... I així successivament fins a arribar a la *n* dimensió.

```
for (unsigned int d = offsetD; d < min(offsetD + blockDim.x, D); d++)
{
    // broadcast centroid position and compute distance to data
    // point along dimension; reading of X is coalesced
    dist += distanceComponentGPU(s_center + (d – offsetD), X + (d * N + t));
}
```

Figura 7. Accessos coalesced al vector de punts.

- ***calcScore_CUDA*:**

  Aquest és l'últim dels clústers que s'executa, i és l'encarregat de calcular la qualitat de la solució obtinguda en cada una de les iteracions. Bàsicament, tal i com fa el *calcCentroids_CUDA*, cada bloc s'encarrega d'un clúster. El que fa, finalment, és calcular la suma de les distàncies de cada punt d'un clúster, al seu centre. Amb això obté l'*score* de cada clúster que guardarà al final de cada iteració. Així l'algoritme pot anar comparant l'*score* de la iteració actual amb l'anterior, de manera que, si pràcticament no varia, pot considerar que la solució ja és prou bona i parar de fer crides als *kernels.* Aquest *kernel*, reuneix les optimitzacions de les reduccions, tal i com hem veiem a la figura 5 i l'optimització dels accessos *coalesced*, tal i com podem apreciar a la figura 6.

3.2 Possible Obstacles

En aquest apartat es mirarà de predir i definir els possibles obstacles amb els que ens podem anar trobant al llarg del projecte.

Un dels primers obstacles amb que ens podrem trobar és amb la instal·lació de la plataforma *Campaign* ja que aquesta ve de tercers i no ha sigut feta pels actors implicats en el projecte. A més, la plataforma ha sigut accelerada amb una versió de *CUDA* antiga (versió 2.1), quan actualment l'eina a sobrepassat la versió 7.0. Com que seria una pèrdua de recursos adaptar-nos a la plataforma *Campaign* i fer servir una versió de *CUDA* antiga, el que farem és adaptar els algoritmes a les últimes versions d'aquesta eina emprada per tal de paral·lelitzar codi amb les targetes gràfiques. Això, fàcilment ens portarà problemes, ja que, probablement, les últimes versions de *CUDA* no utilitzaran funcions que si utilitzaven les primeres i viceversa. Aquí s'espera una implicació per part del desenvolupador que, recolzat pel director, aconsegueixi instal·lar la plataforma *Campaign*.

Un altre dels possibles obstacles amb que es pot trobar el desenvolupador serà a l'hora d'emprar l'eina *CUDA*. En el cas del desenvolupador, l'eina no ha estat utilitzada amb anterioritat i això requerirà, com s'ha dit anteriorment, a un estudi acurat i rigorós per part seva. Per tal de solucionar-ho es recorrerà a la web de la marca *NVIDIA*, creadora d'aquesta eina i es buscaran cursos a través dels quals poder assolir els coneixements necessaris. S'han trobat cursos suportats per la pròpia marca *NVIDIA* i la plataforma *Udacity.*

Finalment ens trobarem en els obstacles típics de qualsevol intent d'obtenir temps en un codi; i és en el no obtenir els resultats esperats. Per tal d'enfocar aquest punt caldrà esser ordenats a l'hora de programar i rigorosos. En cas de no obtenir els resultats esperats, s'espera, per part del desenvolupador, una correcta i desenvolupada justificació, del motiu pel qual s'han obtingut els mateixos.

4 Metodologia

4.1 Mètodes de treball

Abans de començar el projecte caldrà tenir clars els mètodes i eines que farem servir per treballar; així com conèixer bé l'arquitectura amb la que treballarem per tal de poder analitzar en quins punts funcionen millor els algoritme.

Com explicat amb anterioritat, s'utilitzarà *CUDA* per tal d'analitzar els diferents algoritmes de *Clustering*. *CUDA* és suportat per diversos llenguatges, en aquest projecte utilitzarem C++. Bàsicament *CUDA* disposa d'un conjunt de llibreries del llenguatge que permet, utilitzant diferents sentències, accedir a la *GPU* de la nostra màquina i paral·lelitzar els nostres codis utilitzant-la.

4.1.1 Arquitectura

L'arquitectura amb la que treballarem es troba a la Facultat d'Informàtica de Barcelona i és anomenada *Hulk*. És una màquina que disposa d'una multi *GPU* amb la qual tenim la intenció de poder analitzar i executar els algoritmes de la plataforma.

Concretament la màquina disposa de 4 *GPU* de la marca *Nvidia* i corresponents al model *GForce GTX Titan.* Aquest model disposa de 6144 MB de sèrie *[16]*, però a la nostra màquina han sigut ampliats a 12206MB. A més, aquest model de *GPU* disposa de tecnologia *Kepler*

La tecnologia *Kepler*, disposa de 15 SMX (*Streaming Multiprocessors*) i, cada un d'ells disposa de: 192 processador de precisió simple, 64 de doble precisió, 32 unitats per a funions espeials i 32 unitats per a *loads i stores [17]*.

4.1.2 Metodologia de treball *Kanban*

Per tal de portar una bona organització del projecte i de les diferents tasques d'aquests, s'utilitzarà la metodologia de treball *Kanban*. Aquesta metodologia ens permetrà, tant organitzar bé les tasques del projecte, com saber l'estat en que aquestes tasques es torbin. Es va considerar utilitzar *SCRUM* per desenvolupar aquest projecte. Es va decidir no adoptar aquesta metodologia degut a que l'equip d'aquest projecte és petit i hagués comportat un cost innecessari i contraproduent.

Bàsicament s'utilitzarà una eina *on-line*, amb la qual podrem definir les tasques del projecte abans de començar-les. L'eina ens proporcionarà una pissarra (tal i com podem veure a la figura 7), en la qual podrem anar canviant l'estat de les tasques, segons en quin punt de procés es trobin.

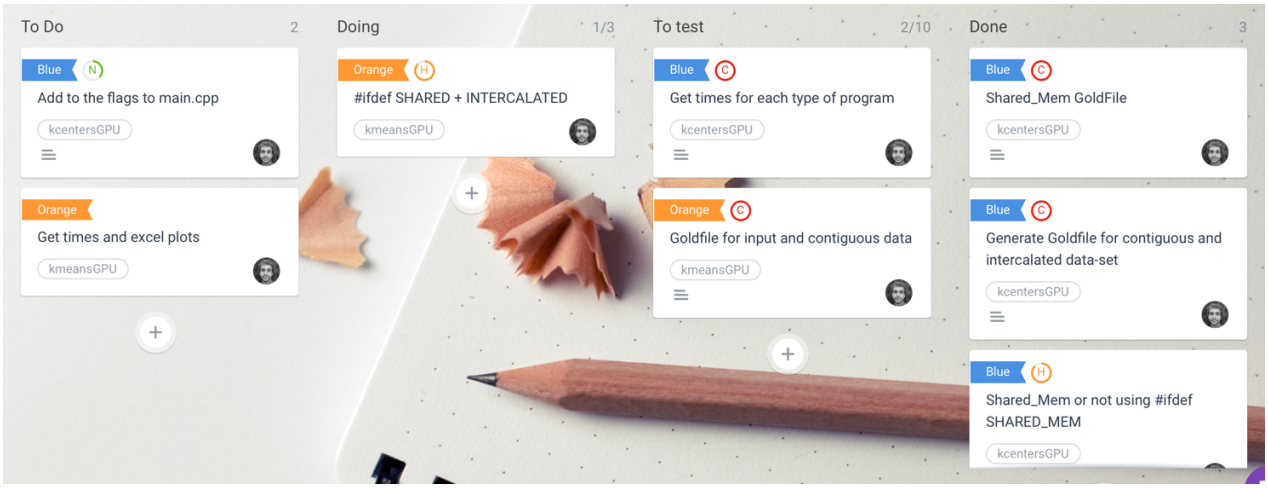


Figura 8. Taula Kanban utilitzada

4.1.3 Generador de punts

Un dels punts més importants d'aquest projecte és el generador de punts que s'utilitzarà per tal de generar diferents tipus de *data-sets*. Com que la idea principal del projecte és analitzar com *CUDA* funciona amb o sense diferents tipus d'optimitzacions, és important, per treure temps i resultats amb molt més valor, utilitzar grans magnituds de dades. Només així podrem enfocar aquest projecte al món del *Big Data*, tal i com s'explica en la introducció, i garantir que el projecte sigui útil per aquest món. Tenint en compte que utilitzarem punts de l'ordre de magnitud 10^6 i de 10^2 dimensions; podríem tenir problemes d'*allocate* en un fitxer, si hi guardem tots aquests punts. Aquí és on entrarà la classe anomenada *Generator.cpp*

Aquesta classe, donades les variables N (nombre de punts), K (nombre de clústers) i D (nombre de dimensions), és capaç de generar-nos K clústers amb N/K punts en cada clúster i, cada un dels punts contindrà les D dimensions especificades en la variable d'entrada D. Per tal de generar el punts de cada clúster s'afegeix un radi aleatori, però de l'ordre de 10^3, al centre del clúster.

```
/**
 * Generates all the points for each cluster. Stores the result at points
 */
void Generator::generateClusters()
{
    for (int numCluster = 0; numCluster < this->K; numCluster++) {
        for (int actualPointOfCluster = 0; actualPointOfCluster < this->totalPointsPerCluster;
actualPointOfCluster++) {
            for (int dimensions = 0; dimensions < this->D; dimensions++) {

                int memory = (numCluster * this->totalPointsPerCluster + actualPointOfCluster) *
this->D + dimensions;

                actualPointOfCluster % 2 == 0 ?
                points[memory] = this->centers[numCluster] + (float) ((rand() % RADIUS)/(float)100):
                points[memory] = this->centers[numCluster] - (float) ((rand() % RADIUS)/(float)100);
            }
        }
    }
}
```

Figura 9. Generador de clústers.

A la figura 8, podem veure com aquests punts són generats. Hem de tenir en compte que la variable RADIUS és de l'ordre de 10^5, la qual cosa provoca que el radi que finalment es suma al punt del clúster, és de l'ordre de 10^2. Amb aquesta funció com a generadora de punts no és, però, suficient. Si hi parem atenció veurem que no està ajudant a que els accessos siguin *coalesced* i, per tant, a l'explotació d'aquesta optimització de *CUDA*. Està escrivint a memòria seguint l'ordre de la generació de punts i no de les dimensions, que és el que ens permetrà explotar els accessos *coalesced*. Quan vulguem explotar aquesta optimització, el generador ens permet re-ordenar els punts per dimensions tal i com podem veure a la figura 9.

```
/**
 * Generates a set of points with contiguous dimensions.
 * A set of points with intercalate dimensions i required.
 */
void Generator::generateClustersContiguous()
{
    for (int actualPoint = 0; actualPoint < this->totalPoints/D; actualPoint++) {
        int iniPositionPoint = actualPoint * D;
        for (int dimension = 0; dimension < D; dimension++){
            this->pointsContiguous[actualPoint + dimension * (this->totalPoints/D)] =
             this->points[iniPositionPoint + dimension];
        }
    }
}
```

Figura 10. Generardor de punts coalesced a partir de punts non-coalesced

Aquestes dues crides són les principals en aquest generador de punts, i les que ens permetran crear aquests data-sets de grans dimensions. Les dos crides seran invocades des de la plataforma *Campaign*, abans d'executar qualsevol dels algoritmes, on podem indicar, en la mateixa execució dels dos algoritmes que analitzem, el *k-means* i el *k-centers*, els N, K i D que amb que desitgem que l'algoritme dugui a terme el *clustering.*

Finalment la classe és capaç de retornar el punter a el els punts ordenats de forma *coalesced* i/o de forma *non-coalesced*, de manera que, en cas que vulguem crear grans magnituds de dades, per tal de simular un projecte real de *Big Data*, no haguem d'escriure-les en un fitxer i poguem retornar a l'algoritme tant sols punter. A més, així podrem generar tants *data-set* com vulguem, de manera que podrem tantes proves amb diferents entrades com vulguem.

4.2 Eines de seguiment

Per tal d'avançar amb correctes en l'empresa que serà el nostre projecte, caldrà que en realitzem un correcte seguiment tant d'aquest com de les tasques que ens anem proposant.

Per aconseguir un bon seguiment serà necessari les reunions que faran, director i desenvolupador, un cop per setmana a la universitat. En aquestes reunions s'avaluarà si el projecte s'està desenvolupant en la direcció correcta. A més, en aquells punts on el projecte es trobi estancat i s'hagin trobat obstacles, es buscaran els possibles errors que hi hagi i es tractarà de solucionar-los.

Com a últim punt, per tal de ajudar al seguiment del codi es treballarà amb *repositories*. Per fer-ho s'utilitzarà l'eina *github*, que ens permetrà gestionar-los i s'utilitzarà l'eina de codi obert *git*. Podem trobar tot aquest projecte a *github* de forma pública *[18]*.

4.3 Mètode de Validació

Per poder garantir que el projecte ha arribat o estigui arribant als objectius esperats, s'hauran de seguir i dissenyar uns mètodes de validació que ens ho garanteixin.

4.3.1 Validació dels temps obtinguts

Es compliran els objectius d'aquest projecte si s'aconsegueix esbrinar amb quines eines és més potent CUDA. Això vol dir que; un cop haguem tret els recursos de memòria compartida i l'accés *coalesced* a les dades de CUDA, haurem de veure quin és el guany de l'original respecte el nostre codi. Això ho podrem fer amb la fórmula de l'*speed-up*:

$$G = \frac{T_{old}}{T_{new}}$$

Amb aquesta fórmula es podrà garantir quina ha estat la pèrdua al treure cada una de les millores. A més, per tal de calcular el guany en percentatge utilitzarem la fórmula:

$$G\ en\ \% = \frac{T_{old} - T_{new}}{T_{old}} \times 100$$

Per últim, per tal d'aconseguir els temps d'execució dels nostres codis, s'utilitzarà la classe *Timer* que el *Campaign* porta incorporada. Amb aquesta classe podrem instanciar la classe *Timer* i, iniciar i parar l'objecte *timer* que haguem obtingut. Amb això podrem obtenir els temps d'aquelles parts dels algoritmes que siguin més importants i analitzar la casuística dels temps obtinguts.

4.3.2 Generació dels *Gold Files*

Un dels passos més importants a tenir en compte serà poder validar que els algoritmes segueixen funcionant una vegada i hem aplicat els nostres canvis i implementacions. Per fer-ho es generarà el que anomenarem fitxers *Gold Files*.

Per tal de generar-los, en primer lloc, crearem una entrada de punts amb la nostra classe *Generator.cpp*, amb els paràmetres N, K i D petits, puguin ser aquests N = 10, K = 2 i D = 2. A continuació, executarem l'algoritme a validar, en la seva primera versió i sense cap de les nostres implementacions i, com a entrada, el fitxer que acabarem de generar. Finalment, guardarem la sortida resultant en el que serà el nostre fitxer *Gold File*. L'últim pas que ens quedarà, per tal de validar l'algoritme, serà executar-lo amb les nostres implementacions, fent servir com a entrada la mateixa amb la que hem generat el fitxer *Gold File* i, comprovar que la sortida n'és la mateixa.

5 Anàlisi i resultats obtinguts

En aquest capítol veurem les implementacions i canvis que s'han fet en el *k-means* i el *k-centers*. A més, veurem els resultats obtinguts amb memòria compartida o sense i explotant els accessos *coalesced* o sense fer-ho. Això farà que obtinguem 4 *outputs* de temps diferents: un sense memòria compartida ni accessos *coalesced*, un altre utilitzant memòria compartida però sense utilitzar els accessos *coalesced*, un tercer sense memòria compartida però, aquest si, explotant els accessos *coalesced* i, per últim, un que utilitzarà tant els accessos *coalesced* com la memòria compartida. Amb aquests *outputs*, que contindran els temps generats amb el *Timer* del *Campaign*, podrem generar els gràfics necessaris per tal d'estudiar i analitzar com afecten aquestes millores a l'execució dels algoritmes. En addició als dos algoritmes, hem afegit la possibilitat de variar els *Threads per Block*, a l'hora d'executar cada un dels *kernels*, per tal d'aconseguir uns resultats més complerts.

5.1 *k-centers*

El *k-centers* ha sigut el primer dels algoritmes analitzats, i amb el que s'han fet els primers canvis. Aquest és un algoritme que disposa d'un sol *kernel*, per comptes dels 3 que disposa el *k-means* i, per tant, era més senzill d'analitzar. A continuació s'explicarà, en primer lloc, els canvis i implementacions dutes a terme en els algoritmes originals i, en segon lloc, s'analitzaran i estudiaran els resultats obtinguts.

5.1.1 Implementació i canvis

a) *Shared Memory*

La idea principal per l'estudi de com afecta la memòria compartida en el l'algoritme *k-means*, era substituir les reserves de memòria compartida per reserves de memòria global. Així, podríem aconseguir executar l'algoritme usant els dos tipus de memòries i, posteriorment, analitzar-ne els resultats. Per fer-ho, s'ha permès que l'algoritme pogués detectar si estava o no definida la memòria compartida i, així, a l'hora de ser compilat, pogués detectar si agafava el tros de memòria global o el de compartida.

```
#ifdef SHARED_MEM
#define MEMBARRIER() __syncthreads()
#define WARPMEMBARRIER()
#else
#define MEMBARRIER() {__threadfence();__syncthreads();}
#define WARPMEMBARRIER() {__threadfence();__syncthreads();}
#endif
```

Figura 11. Barreres de memòria per memòria global i shared

A la figura 10, podem apreciar com definim els diferents tipus de barreres, segons si emprem memòria compartida o global. Com hi podem apreciar, incloem la crida *threadfence().* Aquesta funciona igual que el *__syncthreads()*, però els *threads* a nivell global, és a dir; tots els *threads* de tots els blocs s'esperen entre ells fins que tot el conjunt de *threads,* hagi acabat. Com veiem, en el cas de memòria global, s'ha utilitzat la crida de memòria compartida *__syncthreads()* després d'utilitzar la barrera de memòria global *__threadfence()*. Això és degut a que, per assegurar-nos que no només s'esperin els thread dels altres bloc i també s'esperin entre ells, es fa aquesta segon crida a la barrera de memòria compartida. *CHECK!!!*

```
#ifdef SHARED_MEM
extern __shared__ FLOAT_TYPE array[];                    // shared memory
FLOAT_TYPE *s_dist    = (FLOAT_TYPE*) array;             // tpb distances
int   *s_ID      = (int*)   &s_dist[blockDim.x];         // tpb IDs
FLOAT_TYPE *s_ctr     = (FLOAT_TYPE*) &s_ID[blockDim.x]; // tpb centroid components
#else                                                    // GLOBAL memory is defined
FLOAT_TYPE *s_dist;
int   *s_ID;
extern __shared__ FLOAT_TYPE array[];                    // shared memory
FLOAT_TYPE *s_ctr     = (FLOAT_TYPE*) array;             // tpb centroid components
#endif
```

Figura 12. Declaració de memòria compartida o global tenint en compte les variables definides.

A la figura 11 podem veure com decalarem memòria compartida o no, segons si volem aprofitar-la o, per contra, només fer servir la memòria global. Com veiem, en el cas de memòria global, també fem servir la memòria compartida per la variable *s_ctr. Aquesta, té un impacte molt petit en el codi ja que només s'utilitza per guardar el centre de cada clúster en el nostre codi. Per tant, hem considerat que la utilització de memòria compartida o no, en aquest cas, seria imperceptible i sempre n'utilitzem. Per contra, si que farem servir memòria global en el cas de les distàncies de cada punt al centre del clúster actual (*s_dist).

Aquest canvi si que hauria de tenir un impacte important a l'hora de dur a terme la reducció ja que, quan l'algoritme busqui el centre següent (veure figura 2), farà una reducció en paral·lel, per tal d'obtenir el punt que dista més del centre actual que està tractant i, un cop obtingut, l'assignarà com a centre del següent clúster a calcular. A aquesta reducció (veure figura *CHECK!!!*), que calcularà el valor màxim del vector s_dist en paral·lel, hi haurem d'afegir els WARPMEMBARRIER() i MEMBARRIER() que hem pogut observar anteriorment a la figura 11. La reducció inicial quedarà tal i com podem veure a la figura 12. Els *__syncthreads()* seran substituïts per MEMBARRIER() i en aquells llocs on no hi havia barrera, però que quan treballem amb memòria global, si que serà necessari, hi haurem d'afegir el WARPMEMBARRIER().

```
template <class T, class U>
__device__ static void parallelMax(int BLOCKSIZE, int tid, T *s_A, U *s_B)
{
    if (BLOCKSIZE >= 1024) { if (tid < 512 && s_A[tid + 512] > s_A[tid]) { s_A[tid] = s_A[tid + 512];
s_B[tid] = s_B[tid + 512]; } MEMBARRIER(); }
    if (BLOCKSIZE >=  512) { if (tid < 256 && s_A[tid + 256] > s_A[tid]) { s_A[tid] = s_A[tid + 256];
s_B[tid] = s_B[tid + 256]; } MEMBARRIER(); }
    if (BLOCKSIZE >=  256) { if (tid < 128 && s_A[tid + 128] > s_A[tid]) { s_A[tid] = s_A[tid + 128];
s_B[tid] = s_B[tid + 128]; } MEMBARRIER(); }
    if (BLOCKSIZE >=  128) { if (tid <  64 && s_A[tid +  64] > s_A[tid]) { s_A[tid] = s_A[tid +  64];
s_B[tid] = s_B[tid +  64]; } MEMBARRIER(); }

    if (tid < 32)
    {
        volatile T *vs_A = s_A;
        volatile U *vs_B = s_B;
        if (BLOCKSIZE >= 64) { if (vs_A[tid + 32] > vs_A[tid]) { vs_A[tid] = vs_A[tid + 32]; vs_B[tid] =
vs_B[tid + 32]; } WARPMEMBARRIER();}
        if (BLOCKSIZE >= 32) { if (vs_A[tid + 16] > vs_A[tid]) { vs_A[tid] = vs_A[tid + 16]; vs_B[tid] =
vs_B[tid + 16]; } WARPMEMBARRIER();}
        if (BLOCKSIZE >= 16) { if (vs_A[tid +  8] > vs_A[tid]) { vs_A[tid] = vs_A[tid +  8]; vs_B[tid] =
vs_B[tid +  8]; } WARPMEMBARRIER();}
        if (BLOCKSIZE >=  8) { if (vs_A[tid +  4] > vs_A[tid]) { vs_A[tid] = vs_A[tid +  4]; vs_B[tid] =
vs_B[tid +  4]; } WARPMEMBARRIER();}
        if (BLOCKSIZE >=  4) { if (vs_A[tid +  2] > vs_A[tid]) { vs_A[tid] = vs_A[tid +  2]; vs_B[tid] =
vs_B[tid +  2]; } WARPMEMBARRIER();}
        if (BLOCKSIZE >=  2) { if (vs_A[tid +  1] > vs_A[tid]) { vs_A[tid] = vs_A[tid +  1]; vs_B[tid] =
vs_B[tid +  1]; } WARPMEMBARRIER();}
    }
}
```

Figura 13. Reducció amb el màxim com a operació funcional amb memòria compartida i global

Aquests són els canvis més importants que s'han fet perquè el codi funcioni tant amb memòria global com amb compartida. El següent pas, que veurem a continuació, serà que el codi funcioni tant si garantim els accessos *coalesced* com si no ho fem.

b) Accessos *coalesced*

Com hem vist i explicat anteriorment, un altra del les utilitzacions que aprofitava la plataforma *Campaign*, i que és de les més importants en *CUDA*, són els accessos *coalesced* (veure capítol 3.3). Abans de res és important explicar com quedaran els punt ordenats en memòria en el cas que explotem els accessos *coalesced* i en el cas que no ho fem. Per això podem veure la figura 13. Recordem que, tal i com hem vist en capítols anteriors, la N equival al nombre de dimensions i la D al nombre de dimensions de cada punt. Veurem que en el cas d'accessos no *coalesced*, anomenats intercalats a la figura, tenim els punts intercalats els uns amb els altres, de manera que l'ordre que preval a memòria és el mateix ordre en que s'han generat els punts. En aquest cas, sempre tenint en compte que cada thread s'ocupa d'un punt, és molt probable que l'accés a memòria que fa un *thread*, no *pugui* ser re-aprofitat per un altre *thread* degut a la distància dels accessos d'aquests.

Per contra en l'accés anomenat *coalesced* a la figura 13, l'ordre en memòria que preval és el de les dimensions dels punts generats. Per tant primerament trobarem la dimensió 0, després la 1... I així successivament fins a arribar a la dimensió D-1. De manera que, tot i que cada *thread* s'ocupi d'un sol punt, primer accediran tots a la primera dimensió, seguidament tots a la segona, i així successivament fins arribar a la última de les dimensions, garantint així que l'accés d'un *thread* pugui ser aprofitat per tots els altres *threads.*

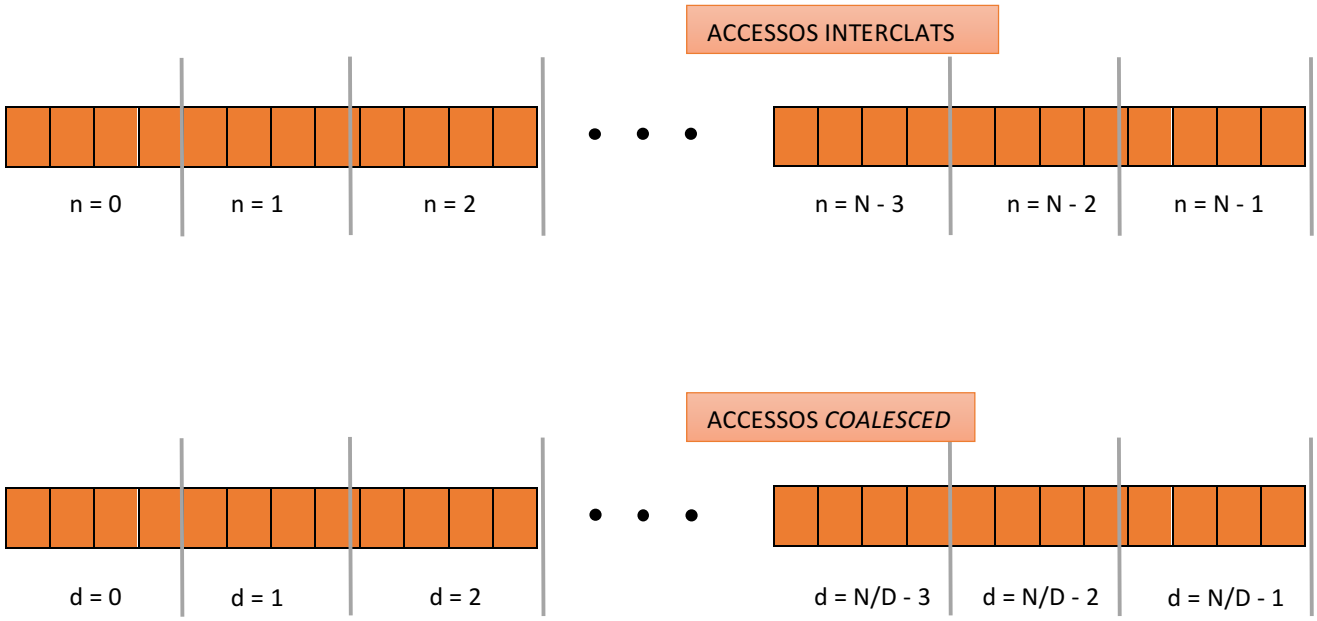


Figura 14. Accessos non-coalesced i coalesced

En el capítol 3.3, havíem vist com la plataforma *Campaign* accedeix a memòria tenint en compte que els punts estan emmagatzemats en l'ordre de les dimensions d'aquests (explotant els accessos *coalesced*). Per tal d'analitzar la millora que aquests accessos ens aporten hem hagut de garantir que el programa funcioni també amb els accessos intercalats. Per això hem implementat els accessos intercalats tal i com podem veure a la figura 14.

```
for (int d = 0; d < min(blockDim.x, D – offsetD); d++)
{
#ifdef INTERCALATED_DATA
    dist += distanceComponentGPU(s_ctr + d, X +  t*D + offsetD + d);
#else
    dist += distanceComponentGPU(s_ctr + d, X + (offsetD + d) * N + t);
#endif
}
```

Figura 15. Garantim els accessos coalesced i non-coalesced

A la figura 14 X apunta als punts que els hi volem aplicar el *clustering*, la variable s_ctr, per contra, apunta al centre actual que s'està calculant. Fent servir s_ctr sempre accedim als centres de forma intercalada; això és degut a que aquests sempre són emmagatzemats de forma intercalada. En canvi en el cas del punter X, si que canvia la forma en que hi accedim segons si estudiem el cas intercalat o el *coalesced*. Per fer-ho ens movem en blocs D en el cas intercalat i ho fem en canvi en blocs de N en el cas coalesced. Això és degut a que, si parem atenció a la figura 13, els blocs en el cas intercalat tenen un mida D igual a les dimensions en canvi, en el cas *coalesced*, els blocs tenen una mida N igual a nombre d'elements.

5.1.2 Resultats obtinguts

Per tal d'analitzar els resultats dels 4 outputs diferents, explicats a la introducció d'aquest capítol, s'han utilitzats unes entrades de: N = 10^6, K = 100 i D = 200. Amb aquests nombres elevats de dades a tractar podrem analitzar, amb temps suficientment grans les execucions del *k-centers.* A més veurem, excepte en el cas del *profiling*, que en els eixos horitzontals dels gràfics, hi ha especificat el nombre de *threads* per bloc que s'han usat per tal de fer l'execució. Això és degut a que també es van fer canvis en la plataforma *Campaign* per tal de poder analitzar com afecta aquest aspecte en l'execució dels nostres algoritmes. Pel cas geèric (*profiling*) s'ha utilitzat un TPB = 512 (*threads per bloc*).

a) Profile:

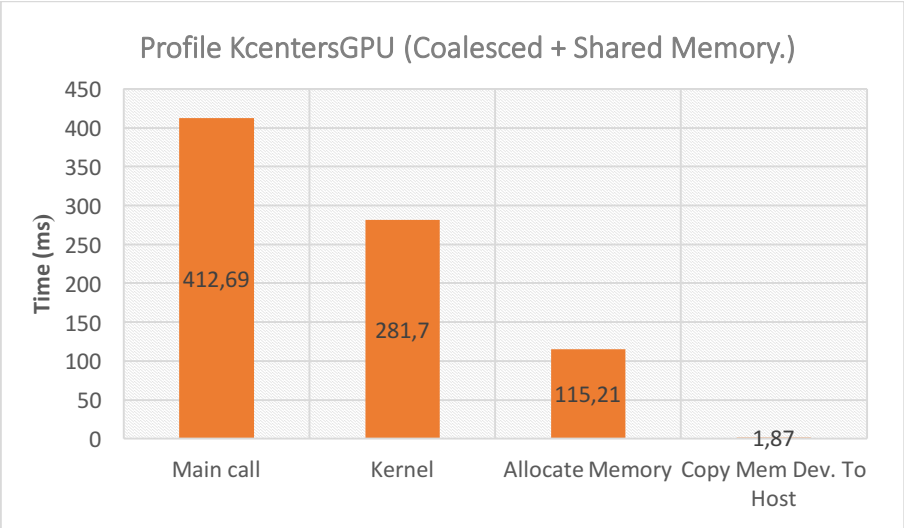


Figura 16. Profiling k-centers

En primer lloc s'ha dut a terme un *profiling* per tal de veure quines eren les parts més costoses a executar de l'algoritme *k-centers.* Podem veure el gràfic resultant de l'execució a la figura 15. Com podem apreciar-hi, s'ha escollit, per el gràfic, l'*output* que té utilitza les dues optimitzacions (*coalesced* + *shared memory*), ja que és aquest el que ens hauria de donar els millors resultats i, al que l'objectiu d'aquest projecte ens portarà, demostrar que realment és així i quin de les dues optimitzacions és més important.

En aquest gràfic apreciem que, com és lògic la crida que més tarda és la crida que fem des del *main*. Però l'interessant de les dades, és que, dins aquesta primera crida, el més costós és el *kernel* que fa servir el *k-centers.* És aquesta part la que ens proposem d'analitzar i veure com varien els temps en els quatre *outputs* possibles de l'algoritme

b) Coalesced

En segon lloc s'ha tret tant l'optimització de memòria compartida com la del accessos *coalesced*. Posteriorment hi introduirem, només, els accessos *coalesced* però seguint actuant amb memòria global. Així aconseguirem veure quina importància hi té el d'accedir a la garantint els accessos *coalesced*, o fent-ho de forma intercalada (veure figura 13). Totes les execucions d'aquest gràfic com les posteriors corresponen a l'execució del *kernel* del *k-centers*.

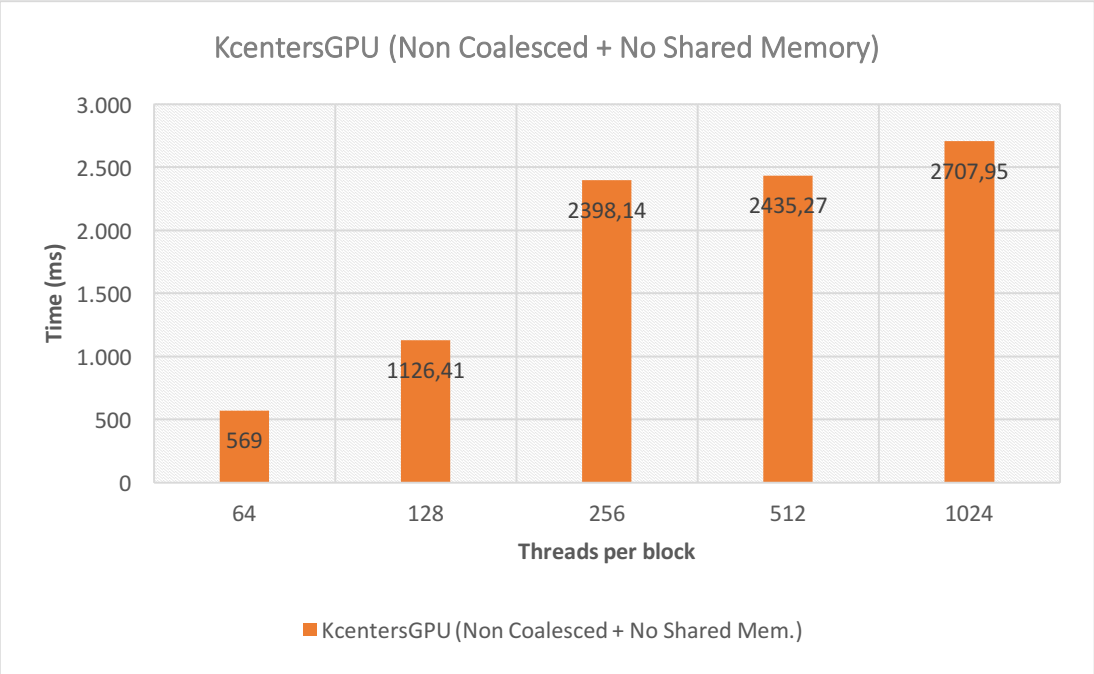


Figura 17. Gràfic amb accessos intercalats i memòria global

A la figura 16 podem apreciar com augmenta l'execució dels temps comparat a quan utilitzàvem les dues optimitzacions, cal tenir en compte que a la figura 16 no estem utilitzant cap de les dues optimitzacions. Com que en aquest apartat ens hem proposat d'analitzar quin era l'impacte dels accessos *coalesced*, hem generat un gràfic (figura 17) on es pot apreciar que l'impacte d'aquesta optimització és vital.

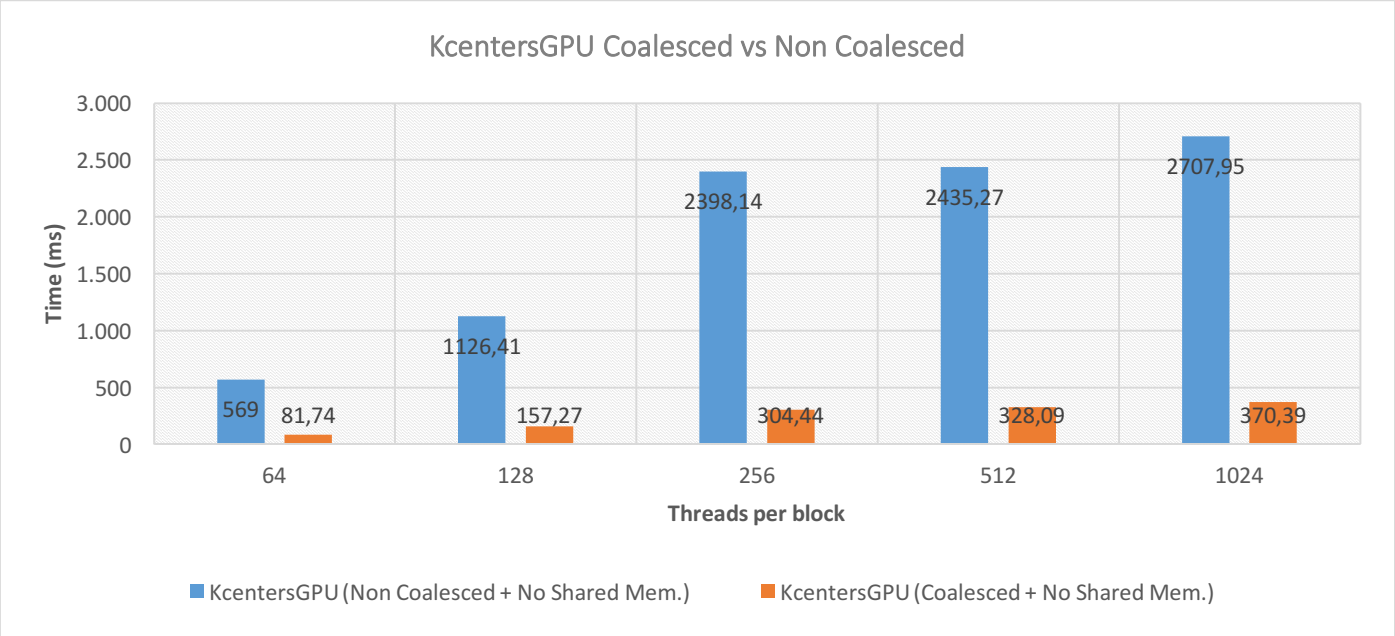


Figura 18. Importància del accessos coalesced k-centers

Podem apreciar quin guany extraordinari s'aconsegueix amb els accessos *coalesced.* Fent el càlcul (agafant TPB = 512) obtenim un guany de:

$$G = \frac{2435,27}{328,09} = 7,42$$

Amb això dir que hem obtingut un guany del 86,52%. El qual és un guany calarament molt elevat i, podem concloure que aquesta execució és essencial en l'algoritme *k-means.*

c) *Shared Memory*

En aquest apartat ens proposem d'analitzar com afecta al *k-centers* la memòria compartida. Per fer-ho, tal i com podem veure a la figura 18, hem fet, directament, un gràfic amb els quatre *outputs* que estem estudiant. Com podem apreciar, per cada TPB tenim els temps que tarda cada execució, representats en el gràfic.

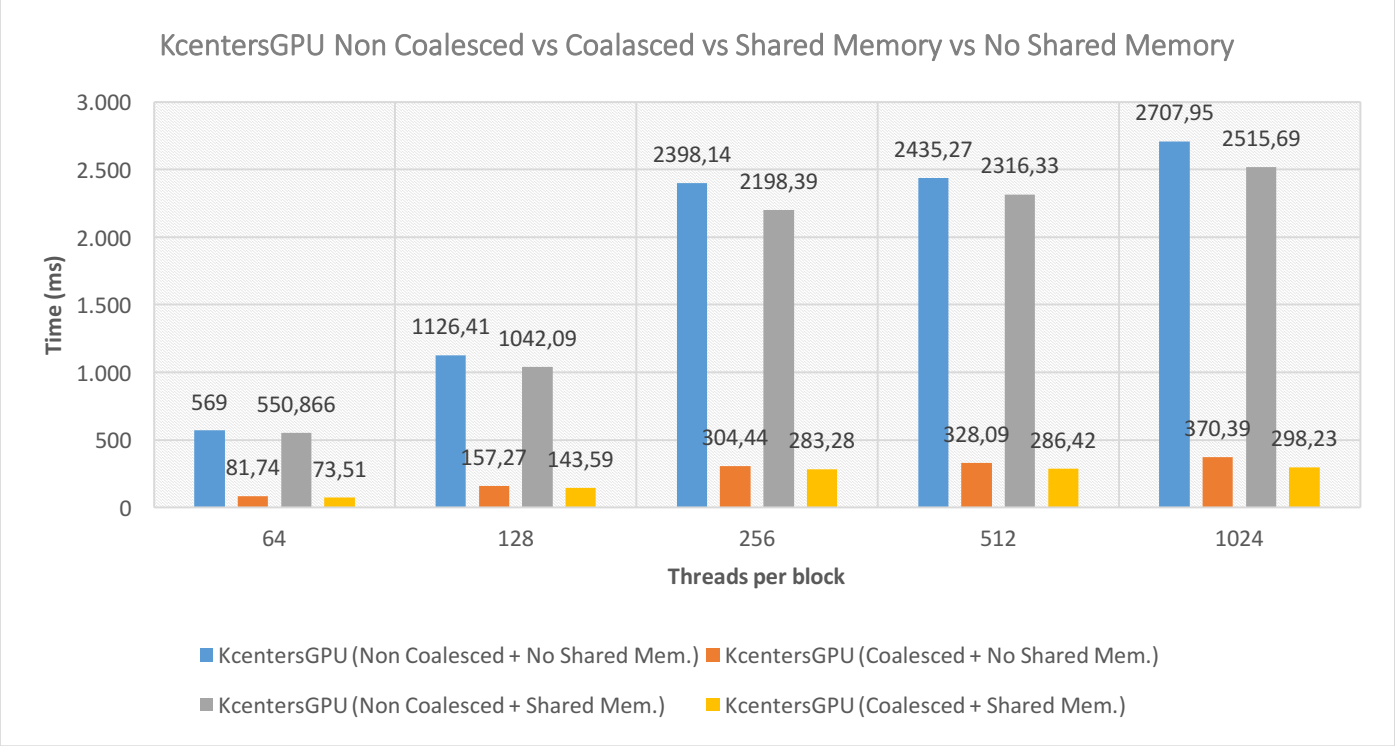


Figura 19. Les quatre sortides de l'algoritme k-centers.

Si parem atenció al gràfic de la figura 18, veurem com, a l'hora d'utilitzar la memòria compartida, sempre guanyem en temps. El qual era d'esperar, ja que en *CUDA* sempre partim de la hipòtesi que la memòria compartida és molt més ràpida que la memòria global. Abans de continuar anem a comprovar quin guany obtenim amb aquesta darrera optimització. Per fer-ho utilitzarem la sortida més ràpida de totes com a temps optimitzat, memòria compartida i accessos *coalesced*, i la sortida que només utilitza els accessos *coalesced* com a optimització utilitzada a l'apartat anterior. Ambdues amb un TPB = 512.

$$G = \frac{328{,}09}{286{,}42} = 1{,}14$$

Veiem que obtenim un augment del 12,07% en rendiment. El qual és un bon augment però no es comparable amb el guany obtingut amb els accessos *coalesced*. Després d'un anàlisi del codi, podem concloure que això és degut a que l'ús principal de la memòria compartida, només es veu reflectit a l'hora de fer la reducció. Les reduccions tenen un pes prou important en l'algoritme però, el fet d'estar emprant la pròpia reducció a l'hora de calcular els màxims, és la optimització principal. Si a aquesta optimització li'n afegim la de memòria compartida, tenim guany en l'execució del codi, tot i que, clarament, no de la forma que ho fèiem amb els accessos *coalesced.*

Variació dels temps segons el TPB

Com podem observar, els temps augmenten de forma no del tot lineal a mesura que també augmenta el TPB. Per tan sembla força clar, que l'algoritme funcionarà millor amb un TPB baix.

Per tant, com a conclusió d'aquest apartat, podem dir que l'algoritme *k-centers*, obté millor rendiment amb les dos optimitzacions utilitzades però, l'imprescindible i realment important, està en garantir els accessos *coalesced* ja que, pel que fa a memòria compartida, bona part de la optimització, ja la obtenim amb la reducció del càlcul dels màxims.

5.2 *k-means*

En aquest apartat ens disposem a explicar i analitzar els resultats obtinguts com a conseqüència de les diferents execucions de l'algoritme *k-means*. Per fer-ho em seguirem la mateixa estructura que en l'algoritme *k-centers*. En el cas que ens envolta en aquest apartat, a diferència de de l'anterior apartat on analitzàvem el *k-means*, tenim 3 *kernels* a analitzar i a modificar les seves implementacions.

5.2.1 Implementació i canvis

A continuació veurem els canvis duts a terme en cada un dels *kernels* tant per aconseguir treure i afegir l'ús de memòria compartida com per fer-ho en els accessos *coalesced*. Per facilitar la comprensió del lector, farem servir un apartat per cada un dels *kernels*, on explicarem, en cada un d'ells, els canvis duts a terme referents tant a memòria compartida com als accessos *coalesced.*

a) ***assignToClusters_KMCUDA*:**

Com s'ha explicat anteriorment, aquest *kernel* és l'encarregat d'assignar cada un dels punts al clúster de centre més proper al propi punt. Tal i com podem apreciar a la figura , s'han afegit *#defines* per tal de controlar, tant els accessos a memòria compartida o global i els accessos de forma *coalesced* o intercalda, segons escaigui en cada un dels casos.

El tros de codi de la figura 19, s'executa dins un bucle on l'element que itera són les dimensions. Seguint la mateixa explicació que en el cas del *k-centers* (veure figura 13), veiem que quan els accessos són intercalats, accedim a un bloc de mida D (nombre de dimensions) i ens movem, en aquell mateix bloc, augmentant la dimensió de l'element *t*; en els cas *coalesced*, en canvi, ens anem movent en blocs de mida N, i accedim a l'element que ens pertoca (essent aquest element l'identificador *t* del *thread*).

```
#ifdef SHARED_MEM
    #ifdef INTERCALATED_DATA
        dist += distanceComponentGPU(s_center + (d – offsetD), X + t*D + d);
    #else
        dist += distanceComponentGPU(s_center + (d – offsetD), X + (d * N + t));
    #endif
#else
    #ifdef INTERCALATED_DATA
        dist += distanceComponentGPU(s_center + (k*D + d), X + t*D + d);
    #else
        dist += distanceComponentGPU(s_center + (k*D + d), X + (d * N + t));
    #endif
#endif
```

Figura 20. #defines per controlar l'ús de cada una de les optimitzacions en l'assignToClusters_KMCUDA

Com veiem també tenim els *#defines* que controlen si estem treballant amb memòria compartida o en global. En aquest *kernel* era necessari controlar aquest cas a l'hora de calcular les distàncies entre el centre del *kernel* actual i punt que s'està tractant en aquell moment de la iteració. Això és degut a que, com podem veure en la figura 20, els centres s'emmagatzemen en memòria compartida o global segons quina de les dues usem. És per això que canvia l'accés a aquests centres si la memòria compartida està o no definida.

```
#ifdef SHARED_MEM
    extern __shared__ FLOAT_TYPE array[];          // shared memory
    FLOAT_TYPE *s_center = (FLOAT_TYPE*) array;    // tpb centroid components
#else
    extern __shared__ FLOAT_TYPE array[];          // shared memory
    FLOAT_TYPE *s_center = CTR;
#endif
```

Figura 21. Shared memory al kernel assignToClusters_KMCUDA

Si parem atenció en aquesta figura 20, veiem que la memòria compartida només s'assigna al punter **s_center* si la memòria compartida està definida. En el cas que usem memòria global, el punter **CTR* (punter que passem al *kernel* com a paràmetre i que apunta als centres de cada clúster) és el que assignem al punter **s_center*. En el cas que fem servir memòria compartida, s'assigna al punter **s_center* el centre del clúster que s'estigui analitzant a cada una de les iteracions, a través del mateix punter **CTR*. Això es fa a mesura que anem iterant el bucle on k va iterant fins k < K (essent K el nombre de clústers).

b) *calcScore_CUDA*:

Aquest *kernel,* com també hem vist anteriorment, és l'encarregat de calcular la qualitat de la solució obtinguda. És important tenir en compte que aquest *kernel*, a diferència del *assignToClusters_KMCUDA*, utilitza reduccions, cosa que haurem de tenir en compte a l'hora de dur a terme les implementacions d'aquest *kernel*, especialment les de memòria compartida en aquest cas de les reduccions.

El *calcScore_CUDA* calcula la distància de forma molt similar al *kernel* d'assignació dels clústers; per fer-ho, podem vuere la figura 21, veurem que l'estrucutra i els accessos són el mateixos que a la figura 19 i, en conseqüència, podem donar com a vàlides les explicacions donades en l'apartat anterior per la figura 19.

```
#ifdef SHARED_MEM
    #ifdef INTERCALATED_DATA
        dist += distanceComponentGPU(s_center + (d – offsetD), X + (offsetN) ∗ D + d);
    #else
        dist += distanceComponentGPU(s_center + (d – offsetD), X + (d ∗ N + offsetN));
    #endif
#else
    #ifdef INTERCALATED_DATA
        dist += distanceComponentGPU(s_center + (k∗D + d), X + offsetN∗D + d);
    #else
        dist += distanceComponentGPU(s_center + (k∗D + d), X + (d ∗ N + offsetN));
    #endif
#endif
```

Figura 22. Càlcul de distàncies calcScore_CUDA

El punt que si és diferent és el de les reduccions; aquest *kernel* utilitza la reducció amb la suma com a operació, per sumar totes les distàncies de cada un dels punts del centre del clúster que s'està analitzant. La reducció que fem servir en aquest cas la podem veure a la figura 22; com hi podem apreciar també utilitzem els *MEMBARRIER()* per tal de substituir la crida *__syncthreads()*. Aquest, ha sigut declarat tde la mateixa manera que a la figura 10.

```
__device__ static void reduceOne(int BLOCKSIZE, int tid, T *s_A)
{
    if (BLOCKSIZE >= 1024) { if (tid < 512) { s_A[tid] += s_A[tid + 512]; } MEMBARRIER(); }
    if (BLOCKSIZE >=  512) { if (tid < 256) { s_A[tid] += s_A[tid + 256]; } MEMBARRIER(); }
    if (BLOCKSIZE >=  256) { if (tid < 128) { s_A[tid] += s_A[tid + 128]; } MEMBARRIER(); }
    if (BLOCKSIZE >=  128) { if (tid <  64) { s_A[tid] += s_A[tid +  64]; } MEMBARRIER(); }

    if (tid < 32)
    {
        volatile T *vs_A=s_A;
        if (BLOCKSIZE >= 64) { vs_A[tid] += vs_A[tid + 32]; }
        if (BLOCKSIZE >= 32) { vs_A[tid] += vs_A[tid + 16]; }
        if (BLOCKSIZE >= 16) { vs_A[tid] += vs_A[tid +  8]; }
        if (BLOCKSIZE >=  8) { vs_A[tid] += vs_A[tid +  4]; }
        if (BLOCKSIZE >=  4) { vs_A[tid] += vs_A[tid +  2]; }
        if (BLOCKSIZE >=  2) { vs_A[tid] += vs_A[tid +  1]; }
    }
}
```

Figura 23. La suma com a operació de reducció. Utilització dels MEMBARRIER()

c) *calcCentroids_CUDA*:

Aquest *kernel* era l'encarregat de re-calcular els centres de cada clúster amb la mitjana de tots els punts del clúster. Per tant aquest és l'únic dels *kernels* que no necessita calcular distàncies, ja que només necessita acumular la suma de tots els elements d'un clúster així com acumular el total d'elements d'aquell clústers. Per fer-ho només ha sigut necessari modificar l'accés als elements, per tal d'adaptar el codi a si fem un accés *coalesced* o intercalat i les reduccions que també utlitiza aquest *kernel*. Pel que fa a la implementació de l'accés als punts, només cal que observem la figura 23, on podem apreciar que segons com emmagatzemem els punt a memòria accedim de maneres diferents al vector de punts.

```
#ifdef INTERCALATED_MEM
    s_centerParts[tid] += X[offset*D + d];
#else
    s_centerParts[tid] += X[d * N + offset];
#endif
```

Figura 24. Accés als punts a l'hora d'acumular-ne la suma.

Pel que fa a les reduccions la implementació duta a terme és la mateixa que la fet a la figura 22. El *calcCentroids_CUDA* utilitza una altra reducció on redueix dos vector en comptes d'un de sol ja que, per la dimensió zero, a més de reduir la suma dels punts, també redueix la suma del nombre d'elements que hi ha en el clúster que s'està analitzant. Aquesta reducció es fa servir per, al finalitzar, poder calcular la mitjana aritmètica amb la suma acumulada de tots els elements, dividida pel total d'elements de cada clúster. Degut a la similitud dels dos clústers, i per simplificar, podem prendre com a vàlida l'explicació de la figura 22.

5.2.2 Resultats obtinguts

Per tal d'analitzar els resultats dels temps obtinguts en les diferents execucions de l'algoritme *k-means*, utilitzarem una estructura molt similar a la de l'apartat anterior; separarem aquest punt en un apartat per cada un dels *kernels* i n'analitzarem els gràfics dels temps obtinguts.

a) assignToClusters_KMCUDA

Per analitzar els temps obtinguts en aquest *kernel*, ens centrarem, bàsicament en el gràfic dels temps obtinguts de la figura 24.

Si parem atenció al gràfic les columnes de color groc (excepte quan executem l'algoritme amb 64 *threads* per bloc), la millor de les execucions sempre és aquesta; quan executem l'algoritme amb *Shared Memory* i garantim els accessos *coalesced*.

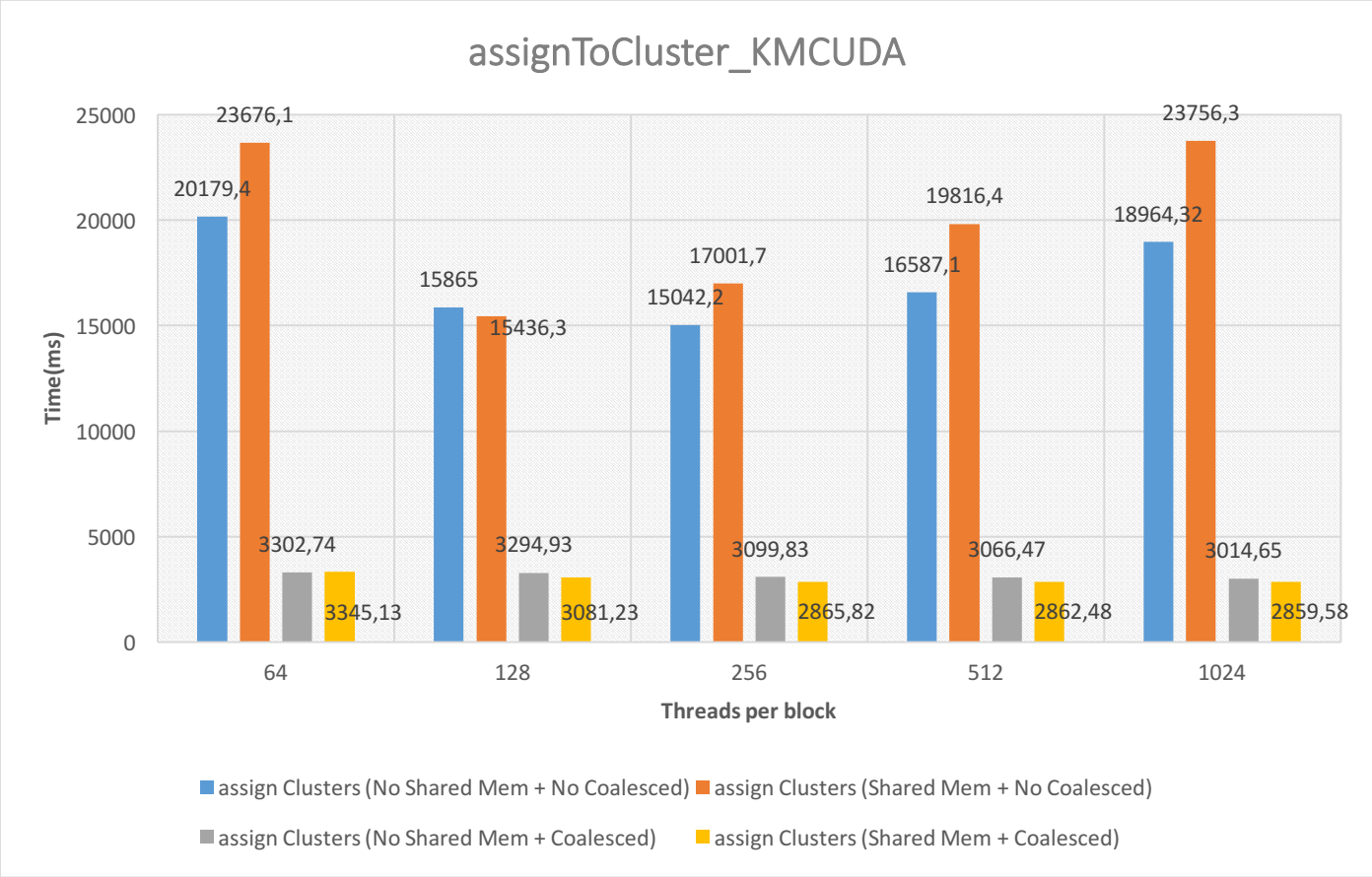


Figura 25. Gràfic dels temps obtinguts al kernel assignClusters_KMCUDA

Que el *kernel* funcioni millor amb memòria compartida és quelcom que ja ens esperàvem i normal tractant-se de ser execucions que utilitzen les dos optimitzacions que s'estudien en aquest projecte.

Veiem, a més, que les dues execucions que utilitzen accessos *colaesced*, columnes grogues i grises, tenen un guany respecte les que fan un accés intercalat (columnes blaves i taronges) molt important. Utilitzant la fórmula del guany entre les dues execucions que no utilitzen memòria compartida, però que una d'elles si que utilitza els accessos *coalesced* (sigui això columnes blaves i grises), i escollint el mateix ús que pel *profile*, de 512 *threads* per bloc, veiem que:

$$G = \frac{16587,1}{3066,47} = 5,41$$

Amb aquest fórmula veiem que, en el cas d'aquest *kernel*, el guany quan s'utilitza accessos *coalesced* respecte de quan no s'utilitzen, és de 5,41. Si fem, amb la fórmula del guany den percentatge obtenim que:

$$G \text{ en } \% = \frac{16587,1 - 3066,47}{16587,1} \times 100 = 81,51\%$$

Podem observar que obtenim un guany en percentatge del 81,51%, un guany que no deixa dubte en que, els accessos *coalesced* són, una vegada més, essencials a l'hora d'optimitzar aquest *kernel*.

D'altra banda, si analitzem com a afectat l'ús de memòria compartida veurem que els resultats obtinguts no són per res comparables amb els darrers obtinguts. De fet, quan fem accessos intercalats (columnes blaves i taronges), veiem que l'algoritme va més ràpid quan no utilitzem memòria compartida que quan si ho fem. A més en el cas dels 64 *threads* per blocs també succeeix entre les columnes grogues i grises. Aquest era un resultat impensable a l'inici del projecte i és un cas que no hem tractat en aquest projecte i que seria interessant d'estudiar en projectes futurs. A més en general el *kernel* va més ràpid amb memòria compartida sumat amb els accessos *coalesced* que de cap altra manera, la qual cosa quadra perfectament amb les suposicions inicials que fèiem en aquest projecte.

Per acabar, com a apunt final, podem dir que és normal que la memòria compartida, en cas que afecti en la millora de l'execució, afecti molt poc. I és que on més s'utilitzava aquesta optimització és en les reduccions i, si parem atenció a la implementació d'aquest algoritme, veurem que no utilitza cap tipus de reducció.

Variació dels temps segons el TPB:

Com veiem a la mateix figura 24 i exceptuant el cas de 64 *threads* per bloc, veiem que els temps tenen una tendència a augmentar de forma no del tot lineal, però tampoc podem dir exponencial, a mesura que el TPB també augmenta. Haurem de tenir en compte això a l'hora d'executar els aquest *kernel* sempre tenint també en compte com afecta aquest paràmetre als altres *kernels*.

b) *calcScore_CUDA*

Pel que fa aquest *kernel*, ens basarem en els temps obtinguts en la seva execució que podem apreciar a la figura 25.

Veiem que igual que en el cas anterior, obtenim una gran millora amb els accessos *coalesced* (columnes grogues i grises) respecte dels accessos intercalats (columnes blaves i taronges). Si, seguint els arguments i les metodologies preses amb el *kernel* d'assignació de clústers, calculem el guany amb les columnes grisa i blava quan tenim 512 *threads*, obtenim que:

$$G = \frac{1274}{253} = 5{,}03$$

Tenim un guany un altre cop essencial amb els accessos *coalesced* i, aplicant la fórmula del guany en percentatge, veurem que obtenim un guany del 80,14%, un percentatge que corrobora la millora obtinguda amb aquests accessos.

D'altra banda, en aquest *kernel*, si que veiem una importància més elevada en l'ús de memòria compartida. Si parem atenció, encara que segueix sent poc guany, si que obtenim pràcticament sempre (excepte en la columna groga amb 64 *threads*), una millor en l'ús de memòria compartida. Aquesta millora en l'ús de memòria compartida, respecte el *kernel* anterior, té sentit degut a l'ús que aquest fa de els reduccions.

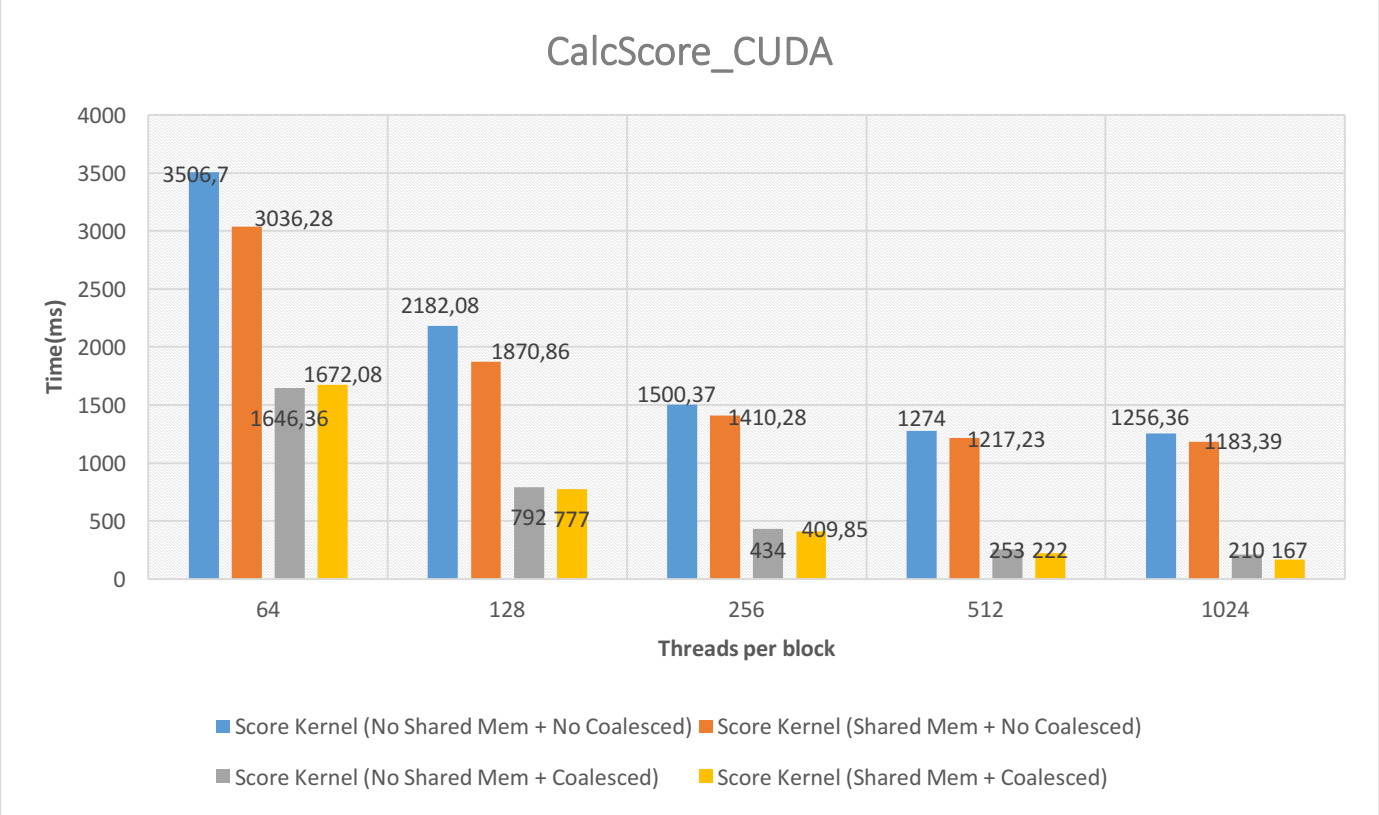


Figura 26. Gràfic dels temps obtinguts en l'agoritme k-means

Variació dels temps segons el TPB:

Veiem que, a la figura 25, el temps disminueixen, a diferència del *kernel* d'assignació de punts als clústers, de forma no lineal a l'hora que el TPB augmenta. Aquest és un altre aspecte que podem tenir en compte a l'hora d'executar aquest *kernel*, ja que sembla força clar que el funciona millor amb un TPB alt.

c) *calcCentroids_CUDA*:

Per aquest *kerenel* ens centrarem en els resultats dels temps obtinguts de la figura 26.

Pel que podem veure al gràfic d'aquest *kernel*, és el que, probablement, ens dona uns resultats més estranys i lluny de les hipòtesis de les que aquest projecte partia. Si que és cert que quan utilitzem 64, 128 i 256 *threads*, el projecte millora especialment amb l'ús dels accessos *coalesced*, i de forma molt menys accentuada en el cas de l'ús de memòria compartida per comptes de la memòria global. En canvi, amb 512 i 1024 *threads* el *kernel* sembla que els temps s'estableixen i no els hi afecta ni en sentit positiu ni negatiu l'afegit o l'extracció de les optimitzacions estudiades en aquest projecte.

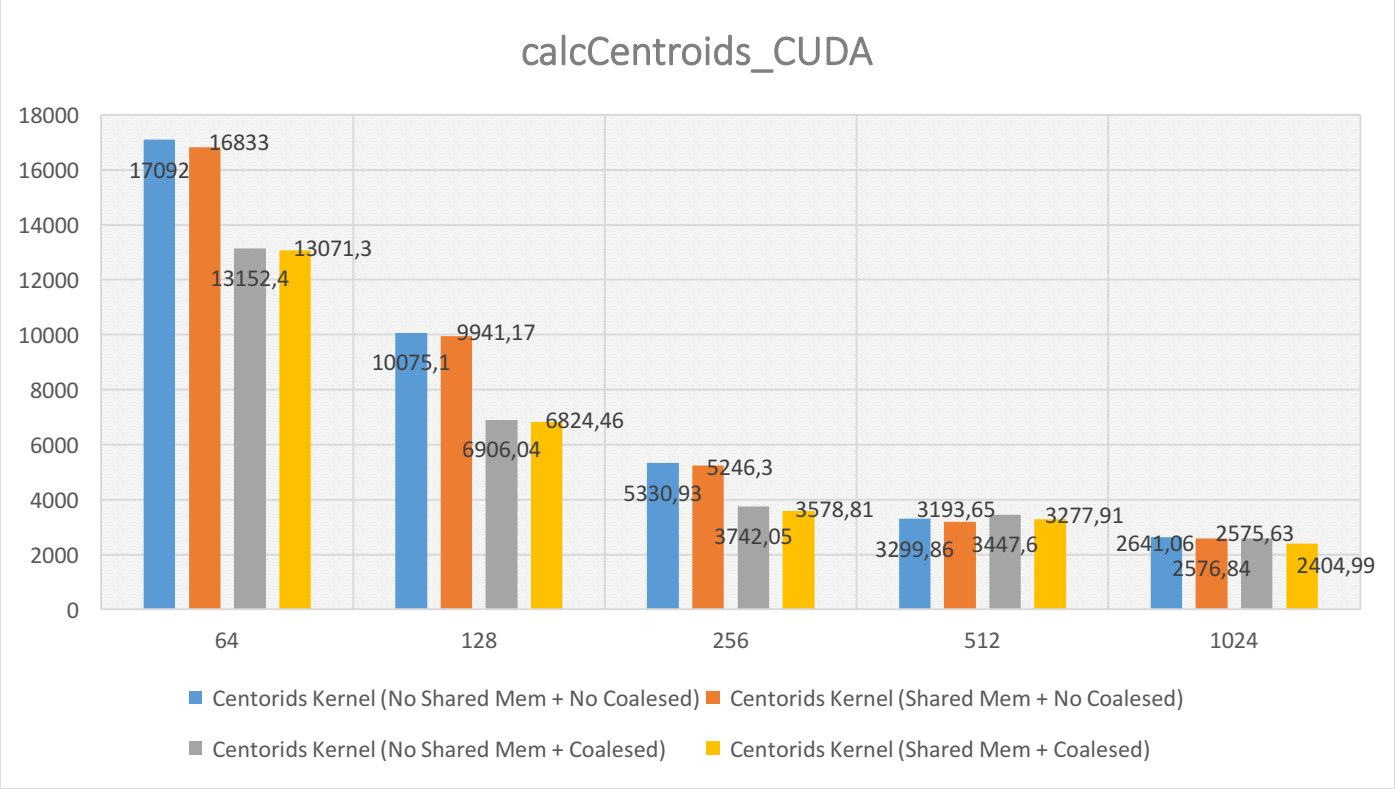


Figura 27. Gràfic dels temps obtinguts en l'algoritme calcCentroids_CUDA

Això, tenint en compte com s'ha estudiat la memòria compartida i els accessos *coalesced*, no hauria de passar mai. I menys en el cas d'aquest *kernel* que, recordant-ne la implementació, utilitza tan les reduccions; cosa que implica l'ús de la memòria compartida com a optimització i, els accessos *coalesced*; a l'hora de calcular la suma de tot els punts de cada un dels clústers.

Per tant, malauradament i en conseqüència, deixarem com a futur estudi aquesta casuística quan utilitzem 512 i 1024 *threads*. D'altra banda si que, en general, per la tendència del gràfic, podem afirmar, que l'ús de memòria compartida i d'accessos *coalesced* són positius per l'execució del *kernel*.

Variació dels temps segons el TPB:

En aquest *kernel*, de la mateix manera que passava amb el *calcScore_CUDA*, els temps disminueixen de forma no del tot lineal, a mesura que el TPB augmenta. Per tant sembla clar que funciona millor amb un TPB alt.

5.3 Variació dels resultats segons els paràmetres N, K i D en l'algoritme *k-means*

Per últim, com a part addicional, hem estudiat el comportament de l'algoritme *k-means* segons la variació dels paràmetres N, K i D. El que hem fet és, per cada *kernel*, fixar dos dels 3 paràmetres i, el tercer, deixar-lo fixe. Així en les diferents execucions on només un paràmetre variava, podíem veure com es comportava cada *kernel* segons el paràmetre en qüestió. El resultat d'això han estat 3 gràfics per cada un dels *kernels*, els quals analitzarem a continuació.

a) N com a variant

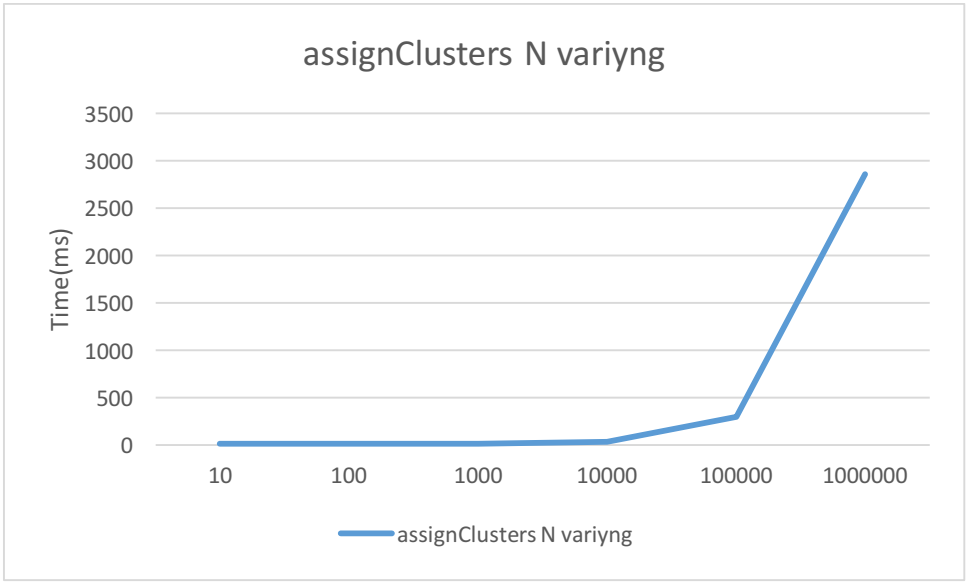


Figura 28. assignToClusters_KMCUDA amb la N variant

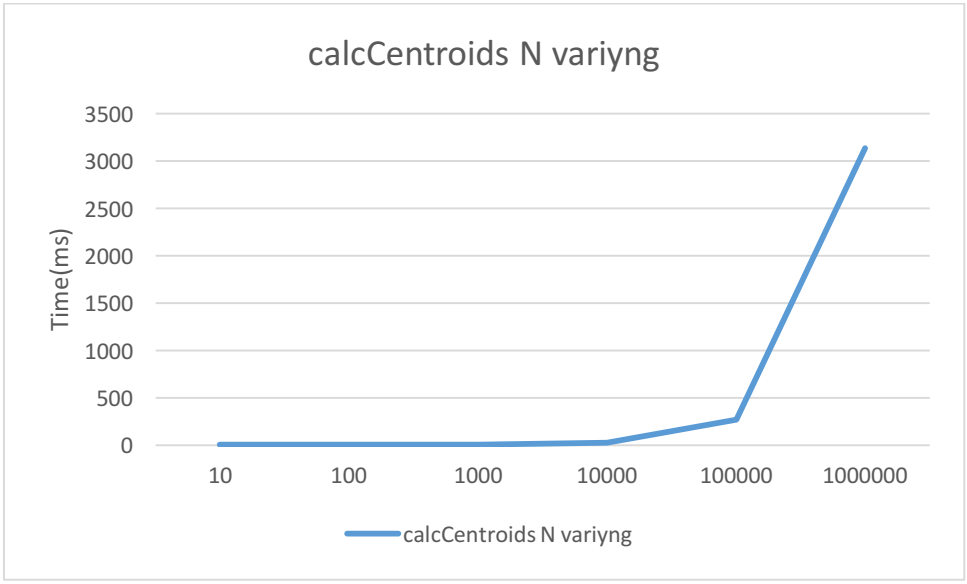


Figura 29. calcCentroids_CUDA amb la N variant

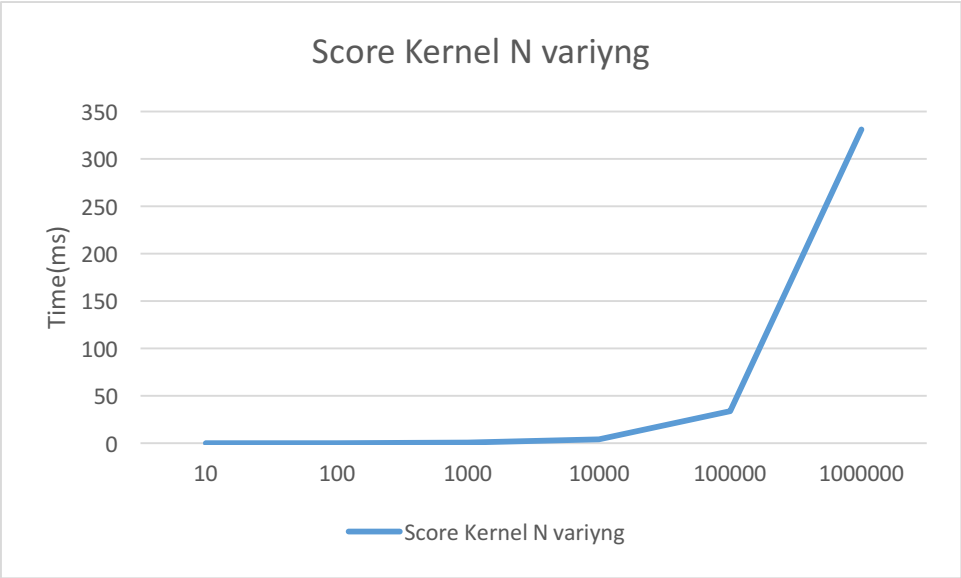


Figura 30. calcScore_CUDA amb la N variant

Com podem veure en les figures 27, 28 i 29, el paràmetre N fa que el temps augmenti de forma en principi lineal encara que la forma de la gràfica pugui semblar el contrari degut a l'escala en la que es veu el gràfic. Això ens indica i corrobora, que els 3 *kernels* analitzen tots els punts que donem com a entrada i això és el que fa que el temps augmentin així, ja que hi haurà més *threads* per cada un dels punts.

b) K com a variant

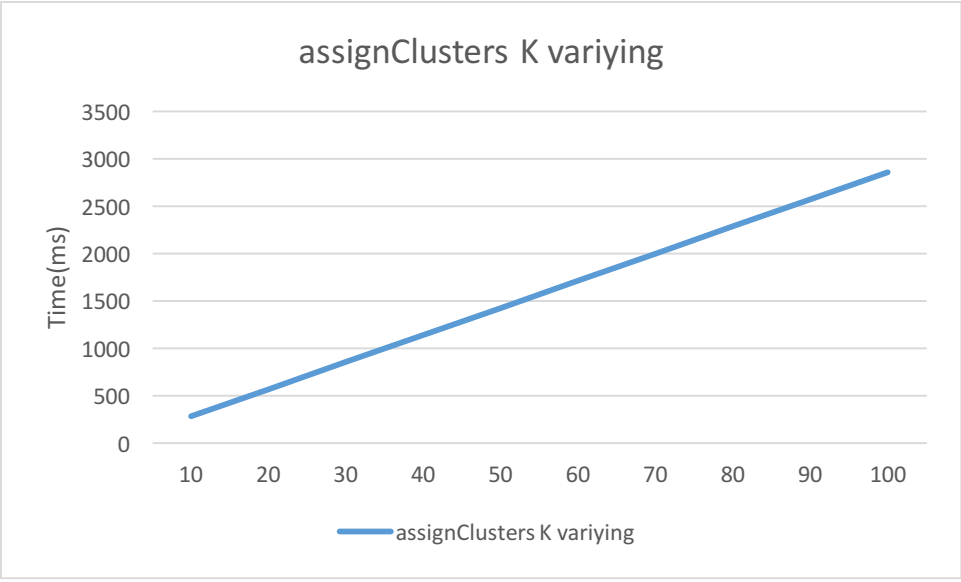


Figura 31. assignToClusters_KMCUDA amb la K variant

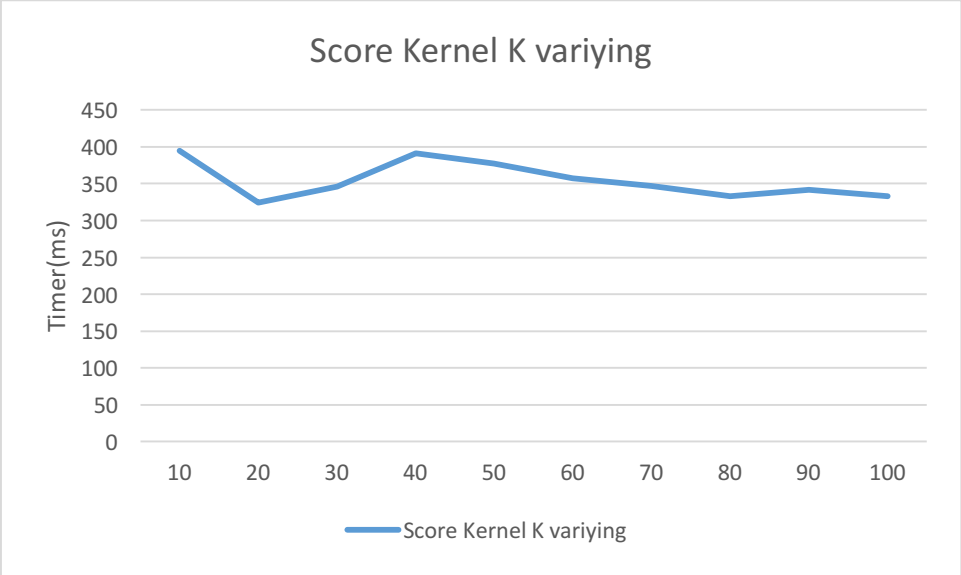


Figura 32. calcScore_CUDA amb la K variant

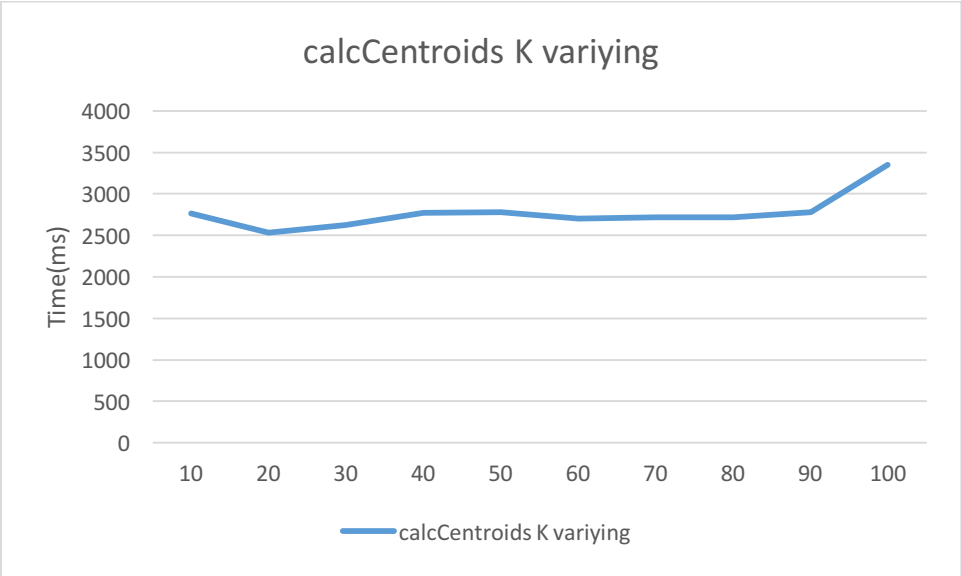


Figura 33. calcCentroids_CUDA amb la K variant

El cas de la K és, probablement, el més interessant de tots. I és que, si analitzem les figures 31 i 32, veiem que la K no influeix en el temps d'execució en els *kernels calcCentroids_CUDA* i *calcScore_CUDA*, de fet; els temps semblen mantenir-se pràcticament constatns. Si ho analitzem veurem que això és un comportament normal en aquests dos *kernels*. Aquests dos *kernels*, es dediquen a analitzar tots els punts independentment del número de clústers, tant si és per calcular-ne la mitjana, com per calcular la distància al centre del seu clúster; d'ambdues maneres passaran per tots els punts.

D'altra banda, a la figura 30, veiem que el *kernel* d'assignació de clústers, si que varia segons el paràmetre K, i ho fa de forma lineal. Tenint en compte que aquest algoritme calculava la distància de cada punt, al centre de tots els clústers, per tal d'assignar-los al clúster més proper; aquest resultat és totalment raonable.

c) D com a variant

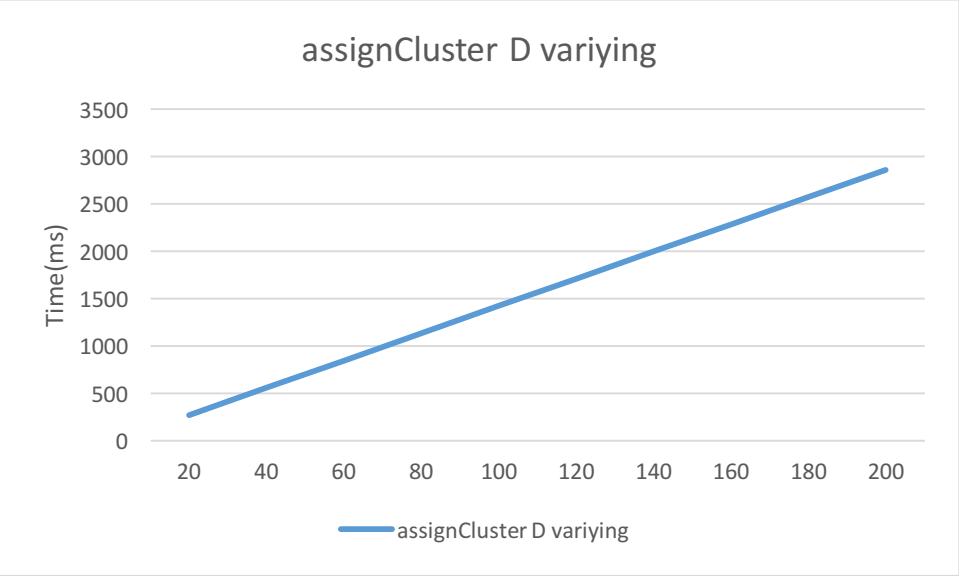


Figura 34. assignToClusters_KMCUDA amb la D variant

En el cas de D com a paràmetre variant, els 3 *kernels*, augmenten de forma lineal. Això és també del tot raonable ja que, els 3 *kernels* tenen més dades a analitzar i per tant el temps augmenta. Ho fa de forma lineal, igual que en el paràmetre n, degut a que el nombre de dimensions no afecta a número de elements a calcular, sinó, d'alguna manera, a les vegades que aquests són tractats.

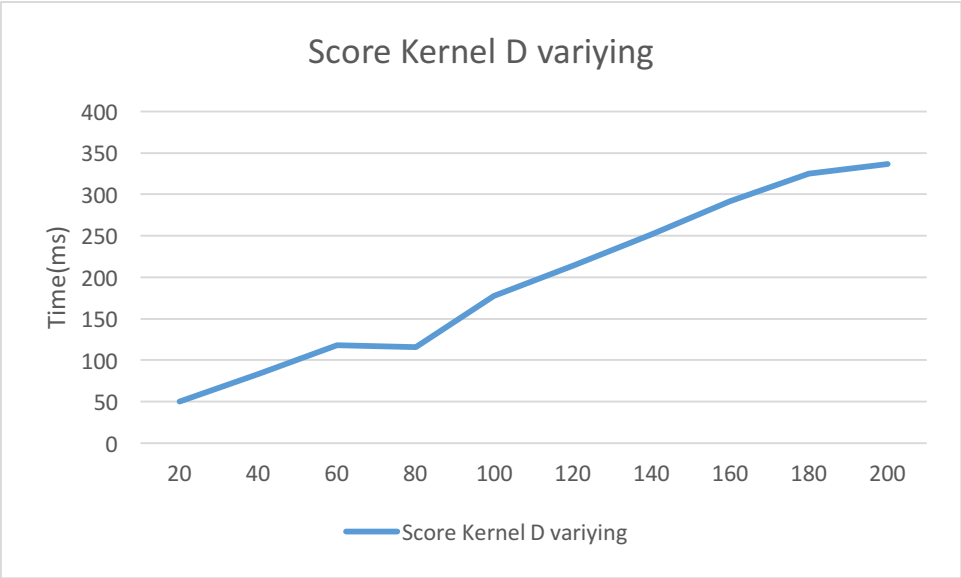


Figura 35. calcScore_CUDA amb la D variant

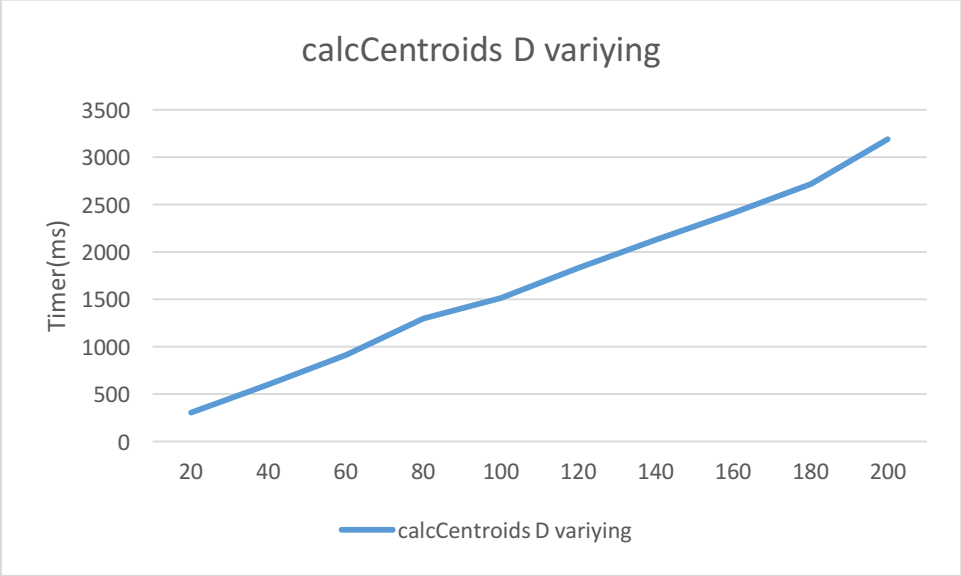


Figura 36. calcCentroids_CUDA amb la D variant

5.4 Conclusions dels resultats obtinguts

En els apartats anteriors hem estudiat com afecten la memòria compartida i els accessos *coalesced* en els algoritmes de *clustering k-means* i *k-centers*.

Hem vist i podem concloure, que ordenar els nostres punts de manera que puguem garantir els accessos *coalesced*, és essencial per l'execució dels nostres algoritmes utilitzant CUDA com a eina de paral·lelització d'aquests algoritmes. Hem arribat a obtenir fins millores superiors al 86% en el cas del *k-centers*, i similars en la majoria de casos en el *k-means,* degut a l'ús dels accessos *coalesced* en comparativa amb els intercalats.

Per tant aquesta és una millora imprescindible a tenir en compte a l'hora d'opitmitzar amb *CUDA* els algoritmes estudiats en aquest projecte.

D'altra banda, en el cas de la memòria compartida hem vist que no és ni molt menys igual d'important que l´ús dels accessos *coalesced*. Gràcies a aquesta, i en comparativa amb l'ús de memòria global, hem arribat a aconseguir una millora en percentatge del voltant del 14% en el cas del *k-centers*. En el cas del *k-means* no hem pogut provar que obtinguem un millora similar a la del *k-centers*, i fins i tot, en alguns casos, no hem tingut millora a l'hora d'utilitzar la memòria compartida. Tot i això si que és cert que en els dos casos, l'algoritme millorava quan utilitzàvem memòria compartida sumat amb els accessos *coalesced* (exceptuant el cas del *kernel calcScore_CUDA* amb 64 *threads* per bloc, que despreciarem per ser una únic). Per tant si que podem concloure que és important afegir a l´ús d'accessos *coalesced*, l'ús de memòria compartida ja que ens post donar un augment en rendiment considerable.

Per últim, esmentar que, en el cas de l'algoritme *k-centers*, hi ha resultats referents a la memòria compartida i en un cas en l'ús d'accessos *coalesced*, que no entenem degut a la pèrdua de rendiments d'aquests. No hem abordat aquests casos i els deixem com a fets dignes d'estudi en un futur. D'altra banda si que hem pogut complir els objectius d'aquest projecte, ja que hem pogut provar que l'ús de memòria compartida sumat amb els accessos *coalesced* és important a l'hora de paral·lelitzar aquests algoritmes. Hem vist, a més, que l'ús d'accessos *coalesced* és vital com a optimització dels dos algoritmes estudiats.

Variacions obtingudes segons el TPB:

En el cas del *k-centers*, sembla clar que funciona millor amb un TPB baix; ja que en l'únic *kernel* de que disposa, els temps augmenten de forma no lineal a mesura que el TPB també augmenta.

En canvi, en el cas del *k-means*, no és tant clar. Com hem pogut observar, en els *kernels* que calculen l'*Score* i els centres, els temps si que varien de la mateixa manera que ho fan en el *k-means* a mesura que el TPB augmenta. D'altra banda, no passa el mateix amb el *kernel* d'assignació de punts a cada un dels clústers. Hem pogut veure que aquest tendia a ser més costós a mesura que el TPB augmentava. No s'ha estudiat directament aquest projecte, però si que sembla que no es pot afirmar quin TPB és millor per aquest *kernel* degut a que més els 3 *kernels* no són iguals de costosos.

6 Planificació temporal

6.1 Tasques del projecte

Per tal de fer més estructurat i ordenat aquest projecte, s'ha decidit considerar, abans de començar-lo, les tasques que seran necessàries per tal de dur-lo a terme. Aquestes s'aniran completant, tal i com s'ha explicat amb anterioritat seguint el mètode *Kanban*. En aquest apartat s'expliquen les més rellevants per tal d'agafar una idea general del que comportarà aquest projecte.

6.1.1 Gestió de projecte

Aquesta tasca és necessària i molt important per tal de tenir el nostre projecte ben redactat i estructurat. Serà part de l'assignatura GEP amb la que s'avaluarà part d'aquest treball final de carrera. Consistirà en escriure una pseudo-memòria d'aquest projecte a més de fer una primera exposició oral.

6.1.2 Instal·lació i entorn de treball

Com s'ha comentat amb anterioritat, per dur a terme aquest projecte s'utilitzarà la plataforma anomenada *Campaign*, recolzada per la Universitat d'*Stanford*. Aquesta ve amb un manual d'instal·lació que s'haurà d'entendre i estudiar per tal de poder-la dur a terme.

Un cop s'hagi aconseguit instal·lar la plataforma serà hora de fer les primeres execucions dels codis que aquesta ens subministra. Això s'hauria d'aconseguir fer amb dades de gran magnitud per tal de garantir-ne el correcte funcionament i eficiència en tots els casos. Un cop això s'hagi aconseguit podrem començar pròpiament el nostre projecte.

6.1.3 Aprenentatge *CUDA*

Un cop finalitzada la instal·lació de l'entorn i garantida la seva funcionalitat i, abans de començar a mirar el codi amb profunditat, haurem d'aprendre a fer servir l'eina amb la que treballarem en aquest projecte: *CUDA*.

Per aconseguir entendre l'eina i saber-la utilitzar s'utilitzaran cursos *on-line* i llibres, els quals seran suportats per la marca *Nvidia*, marca creadora de *CUDA*. Un cop fet els cursos i realitzat un estudi acurat i rigorós de *CUDA*, podrem començar amb l'anàlisi i implementació del codi.

Aquesta és una tasca important ja que l'aprenentatge i domini d'aquesta eina és un dels punts claus i més importants d'aquest treball.

6.1.4 Anàlisi del codi

El més important per una bona implementació del codi i estudi d'aquest és haver-ne fet, prèviament, un bon anàlisi. Per tal de fer això caldrà fer un estudi acurat i precís d'aquest. El que vol dir entendre què fan els algoritmes de la plataforma *Campaign* i, especialment, com estan aquests paral·lelitzats i optimitzats.

Amb un bon treball en aquest apartat, fer la implementació, amb els canvis que es considerin convenients per tal d'estudiar els algoritmes de la plataforma, ens portarà molt menys temps i podrem actuar i resoldre els possibles conflictes i errors de forma molt més eficient.

6.1.5 Implementació del codi i canvis en la programació *CUDA*

Aquesta és la tasca més important segurament, ja que depenent dels resultats d'aquesta, podrem saber si haurem aconseguit un dels objectius clars i importants dels projecte: analitzar en quins punts CUDA és més potent i trobar de quines eines no podem prescindir.

Cal tenir clar que l'èxit d'aquesta tasca depèn, en una part molt important, la correcta finalització i la exigència que invertim en les tasques anteriors. Les tasques prèvies a la que ens estem referint ara seran vitals per un resultat satisfactori en la tasca a la que ens referim.

6.1.6 Validació dels resultats obtinguts

Un cop feta la implementació del codi, caldrà dur a terme la validació dels resultats que haguem obtingut. Això es farà seguint les fórmules i mètodes explicats a l'apartat Metodologia.

Aquesta tasca és important per una correcta correcció i garantia de que els resultats obtinguts són correctes i tenen sentit i, en cas contrari, ens portarà al motiu pel qual aquests no ho són i a un possible replantejament de la implementació.

6.1.7 Escriptura de la memòria

En aquest apartat ens proposem d'escriure la memòria del projecte i la documentació d'aquest. Una part d'aquesta estarà escrit i s'aprofitarà gràcies a la primera tasca esmentada: la gestió de projecte. Emperò aquesta necessitarà modificacions, millores i ampliacions. Això s'anirà fent al llarg del projecte i es podrà anar fent en paral·lel amb totes les tasques prèvies esmentades. Per tal de veure-ho de forma més clara tenim el diagrama de Gantt la figura 38.

6.1.8 Preparació lectura

Arribats en aquest punt s'hauran d'haver finalitzat totes les tasques anteriors. En aquesta tasca ens dedicarem a practicar i preparar la lectura del projecte que tindrà lloc a la Facultat d'Informàtica de Barcelona. Haurem d'exposar i defensar la feina feta al llarg de tot el projecte davant d'un tribunal.

6.2 Taula de planificació de les tasques

Per tal de fer una primera aproximació al temps que portaran les tasques esmentades en l'apartat anterior, s'ha construït la següent primera taula. En aquesta podrem veure les tasques analitzades en l'apartat anterior i, amb quina tasca filla estan relacionades i el cost en hores que cada una d'elles comportarà.

Com podem veure, en la taula adjunta en aquest apartat, algunes de les tasques anteriors s'han separat en diverses tasques filles i, per contra, algunes d'elles, s'han agrupat en una tasca pare. Amb això obtindrem una major separació de les tasques i, en conseqüència, una major simplificació d'aquestes.

Tasca pare o grup	Tasques filles	Temps estimat
Gestió de projecte	Abast del projecte i context	20h
Gestió de projecte	Planificació temporal	10h
Gestió de projecte	Gestió i sostenibilitat	15h
Gestió de projecte	Presentació preliminar	15h
Gestió de projecte	Presentació oral i documentació final	15h
Instal·lació i entorn de treball	Instal·lació entorn	60h
Instal·lació i entorn de treball	Execució entorn treball	30h
Implementació del codi	Aprenentatge *CUDA*	50h
Implementació del codi	Anàlisi del codi	60h
Implementació del codi	Implementació del codi	80h
Validació	Càlcul dels temps	25h
Documentació i lectura	Escriptura de la memòria	30h
Documentació i lectura	Preparació lectura	35h
Total		**445h**

Figura 37. Taula amb el càlcul dels temps de cada tasca

6.3 Diagrama de Gantt

Un cop tenim clares les tasques que seguirem gràcies a les taules i l'anàlisi d'aquestes, s'ha decidit emprar un diagrama de Gantt. Aquest ens permetrà tenir una clara visió gràfica del cost que tindran les tasques i, un dels punts més importants, les dependències entre aquestes tasques.

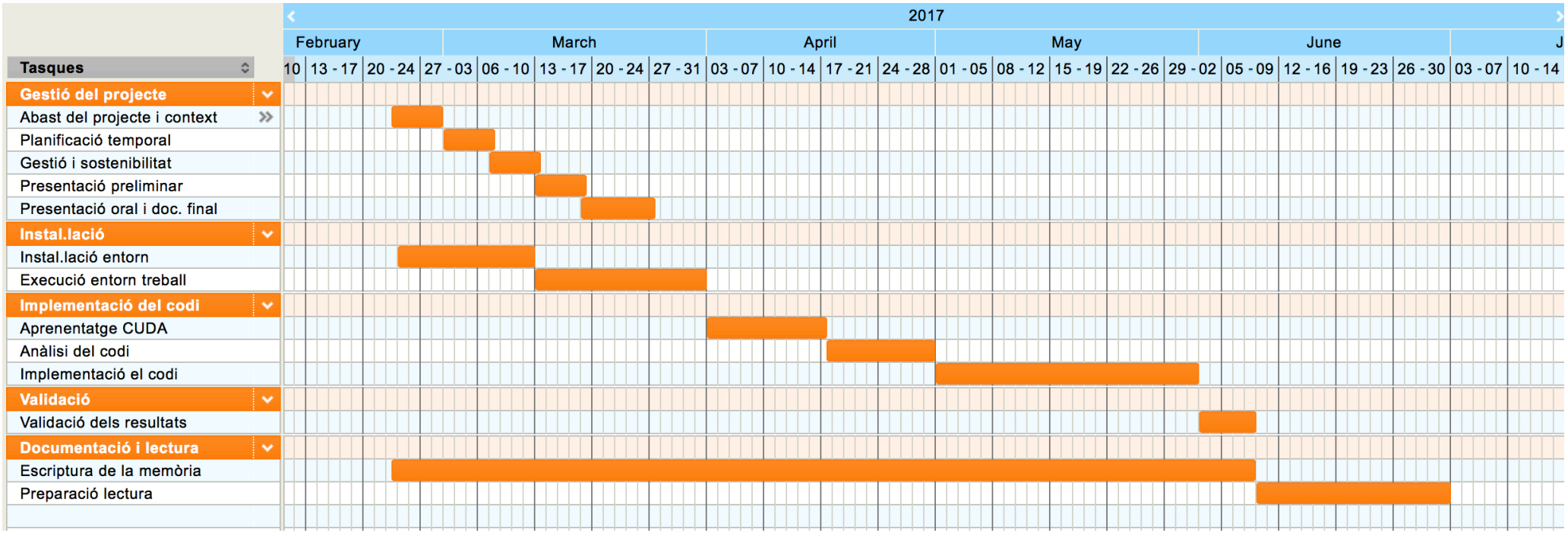


Figura 38. Diagrama de Gantt de les tasques del projecte

6.4 Valoració d'alternatives i pla d'acció

A pesar d'una acurada i rigorosa planificació del projecte, ens podem trobar amb diferents inconvenients i entrebancs que ens dificultin un avenç fluid i dinàmic entorn a aquest treball. En aquest apartat definirem quins possibles inconvenients podem trobar i quins plans d'acció seguirem per tal de superar-los amb èxit i eficiència.

- **Màquina amb la que treballem inhabilitada:** Podria passar que la màquina amb la que la Facultat d'Informàtica de Barcelona ens ha ofert per treballar no funcioni correctament, ja sigui per un problema del *hardware* o per problemes externs. En aquest cas tindrem alternatives com la de fer servir altres màquines de la pròpia universitat o, demanar ajuda al BSC, centre molt arrelat a la universitat i que probablement ens oferirà alternatives.

- **Instal·lació de la plataforma *Campaign* no assolida:** Podria ser que no s'aconseguís instal·lar la plataforma *Campaign* per falta de compatibilitat amb les nostres màquines o, perquè fa servir versions de llibreries massa antigues. En un cas així tenim moltes alternatives de codi obert a la plataforma de codi obert *github* o se n'ha trobat d'altres suportades per diferents universitat. Si que és veritat però, que la plataforma més ben documentada i fiable és la ja trobada però en cas de la impossibilitat de fer-la servir hi ha alternatives i el projecte podria seguir el seu camí amb correctesa.

- **Dificultats en la implementació del codi:** És possible, i pràcticament segur, que al llarg d'anar implementant el codi ens trobem amb diferents dificultats que ens encallin i no ens causin diversos alentiments en el desenvolupament d'aquest projecte. Per tal de solucionar-los el desenvolupador i el director del projecte es reuniran i miraran de resoldre tots els conflictes amb els que es vagin trobant.

7. Recursos humans i materials

Saber quins recursos materials i humans necessitarem per el nostre projecte, és un altre dels punts importants per tal de tenir ben planificat el nostre projecte. Els recursos humans seran aquells rols que necessitem que assumeixin diferents persones en el treball. En aquest projecte, a part del rol de director, assumirà tots el diferents rols el mateix desenvolupador. Per altra banda, els recursos materials seran totes aquelles eines físiques que necessitem per tal d'assolir els objectius en el nostre projecte.

Quan a recursos humans, necessitarem 4 rols diferents. En primer lloc un *Project Manager* encarregat de ser el màxim responsable del projecte i de dirigir el mateix per sota del director. Per continuar necessitarem un analista, que s'encarregarà de analitzar el codi i trobar quins en són els punts claus i que s'hauran d'analitzar i re-implementar. En tercer lloc tenim el rol d'enginyer de computadors, que s'encarregarà de saber com funcionarà el *hardware* de la nostra màquina i aportarà solucions a les noves implementacions. I, en quarta i última posició, necessitarem un tester, que serà l'encarregat de cobrir les tasques de validació i corroborar que hem complert l'objectiu del treball i que, els resultats són els esperats.

Un cop analitzats els recursos humans hem analitzar també els recursos de tipus materials. Aquests són recursos, bàsicament, de tipus *hardware*. En concret, necessitarem: una màquina de la Facultat d'Informàtica de Barcelona, que contindrà les targetes gràfiques que necessitem; a més, necessitarem un ordinador d'ús personal que en permetrà connectar-nos a la màquina de forma remota.

8 Gestió econòmica

Un cop tenim clars els recursos humans i la planificació temporal, és hora de començar a fer una gestió econòmica del projecte que ens envolta. En aquest apartat ens proposem de fer aquesta gestió fent una previsió dels pressupost que serà necessari per aquest projecte.

Els quatre tipus de costos, i més importants, que analitzarem seran: recursos humans, recursos *software*, recursos *hardware* i els costos indirectes. Tots ells són explicats i calculats en els següents apartats.

8.1 Recursos humans

El primer dels grans costos que comportarà un projecte informàtic com el que ens escau, serà els cost dels recursos humans. Totes aquelles posicions que exerceixi hom, hauran d'ésser remunerades. Els recursos humans necessaris seran, com hem vist amb anterioritat: un *Project Manager*, cap de projecte i màxim responsable del mateix; un analista, encarregat d'analitzar el codi i cercar en quins punts és aquest més lent; un enginyer de computadors, encarregat de saber quina part, i com fer servir, el *hardware* de la nostra màquina per tal de millorar els algoritmes i; finalment, un tester, que serà l'encarregat de comprovar que realment el codi segueix funcionant i n'especificarà els canvis de rendiment després de les implementacions. En el projecte que ens escau, el desenvolupador serà l'encarregat d'interpretar tots quatre rols.

A continuació podem veure, en la Taula 1, els costos d'aquests recursos humans. Per tal de calcular-los, s'ha utilitzat el cost per hora de cada posició amb el producte de les hores dedicades; d'aquesta manera hem obtingut el cost de cada una de les posicions.

Els costos són una aproximació que ha sigut extreta del Butlletí Oficial de l'estat (BOE) *[19]*.

Posició	Cost per hora (€/h)	Hores dedicades (h)	Cost (€)
Project manager	40	110	4400
Analista	25	90	2250
Enginyer de computadors	25	190	4750
Tester	10	50	500
Total		**445 h**	**1900 €**

Taula 1. Previsió costos dels recursos humans del projecte.

8.2 Recursos *software*

Per tal de desenvolupar i millorar els algoritmes que ens proposem en aquests projecte, necessitarem, com és evident, la utilització de recursos *software*. Recursos tals com: *CUDA*, c++, *github* entre d'altres. Tots ells són recursos codi obert i no tenen cap cost addicional per el nostre projecte. Si que és cert, que alguns d'ells, com és el cas de *github*, tenen la opció de pagar per fer servir alguna de les seves funcionalitats. En el projecte que ens escau, no farem servir cap d'aquestes funcionalitats i, per tant, no tindrà costos de recursos que siguin del tipus *software*.

8.3 Recursos *hardware*

Un altre dels punts importants que ens poden comportar costos són els recursos *hardware*. Alguns exemples en són; l'ordinador amb el que treballarem i la màquina de la Facultat d'Informàtica de Barcelona a la que ens connectarem per tal de realitzar el nostre projecte.

En la següent taula (Taula 2), podem veure aquests costos. Per calcular-los s'ha utilitzat el producte del temps que s'ha utilitzat el recurs, amb el cost per hora del mateix recurs.

Recurs hardware	Cost (€)	Vida útil (anys)	Ús (h)	Amortització (€)
Ordinador *macbook* pro	1500	7	445	10,5
Màquina *hulk* FIB	10000	10	390	44,5
Total				**55 €**

Taula 2. Previsió costos dels recursos hardware del projecte.

8.4 Costos Indirectes

El següent dels punts importants a tenir en compte són els costos indirectes que emprarem. Tals com la electricitat o l'accés a internet. Aquests són calculats en la següent taula (Taula 3), on podem veure la previsió dels costos indirectes calculats amb el producte del cost per hora per l'ús del mateix producte en hores com a unitats.

Costos indirectes	Cost (€/h)	Ús (h)	Cost total (€)
Internet	0,12	445	53,4
Electricitat	0,56	445	249,2
Local de treball	0,30	445	133,5
Total			**436,1 €**

Taula 3. Previsió costos indirectes del projecte

8.5 Costos totals

Un cop analitzats els quatre grans blocs de costos, hem cregut necessari fer un petit resum de quin serà el pressupost total previst. En la següent taula (Taula 4) podem apreciar quins són aquests costos totals.

Tipus de cost	Cost (€)
Recursos humans	1900
Recursos *software*	0
Recursos *hardware*	55
Costos indirectes	436,1
Total	**2391,1**

Taula 4. Resum costos totals del projecte

8.6 Imprevistos i desviacions sobre el pressupost inicial

Per últim, un cop calculat el pressupost en els apartats anteriors, només ens faltarà pensar i analitzar possibles imprevistos que puguem tenir i que facin augmentar el pressupost. Creiem que si s'ha fet una gestió econòmica acurada i rigorosa, l'augment en el pressupost degut a possibles imprevistos, no hauria de ser superior al 20%. Això implica que, tenint el compte el cost total calculat en el pressupost calculat anteriorment (veure taula 4), l'augment del mateix no hauria de superar:

$$\frac{20}{100} \times 2391{,}1 \; = 478{,}22 \; €$$

En l'apartat següent podrem veure, en un pla de contingència, l'import que pot augmentar el nostre projecte segons els possible imprevistos que puguin anar esdevenint.

8.6.1 Pla de contingència

En aquest apartat, calcularem el possible augment del pressupost del nostre projecte, tenint en compte els riscos que puguin anar apareixent i la contingència de que aquests succeeixin. En la següent taula (Taula 5) veiem la exposició al risc; aquest ha sigut calculat amb el producte de la ocurrència de que el propi risc succeeixi pel cost estimat d'aquest risc.

Risc acceptat	% Ocurrència	Cost estimat (€)	Exposició al risc (€)
Augment hores enginyer computadors	50%	500	250
Augment hores analista	50%	200	100
Augment hores tester	30%	100	30
Augment cost electricitat	15%	20	3
Augment cost internet	15%	20	3
Total			**386 €**

Taula 5. Pla de contingència

8.7 Control de gestió

Les possibles desviacions i causes d'aquestes, es troben altament relacionades amb el pla de contingència descrit en l'apartat 1.6.1. A continuació es descriuen les més importants.

- **Augment en hores invertides respecte les esperades:** Si fos necessari invertir més hores de les esperades, s'hauria de remunerar als diferents treballadors del projecte. Això podria incrementar en un sobre cost de 380€.
- **Augment costos indirectes:** Podria passar que ens incrementessin les factures de llum o línia d'internet, més de l'esperat. Això podria suposar un sobre cost de, com a molt, 6€.

Per tal de mantenir-nos dins el pressupost esperat i calculat a l'apartat de gestió econòmica, es reuniran setmanalment els director i desenvolupador del treball i corroboraran que s'estiguin complint els costos esperats.

9 Sostenibilitat econòmica, social i ambiental

En aquest apartat ens proposem d'analitzar la sostenibilitat econòmica, social i ambiental del projecte. Per fer-ho emprarem l'anomenada matriu de sostenibilitat. En aquesta podrem veure els costos de cada sector de sostenibilitat a analitzar.

El format dels costos serà: "ponderació cost/màxima ponderació possible". El resultat total de cada sector serà un número entre -60 i 90; essent 90 un projecte molt sostenible i un -60 un projecte gens sostenible. En la següent taula (Taula 5) podem veure la matriu resultant de l'estudi de sostenibilitat.

Per tal d'analitzar la matriu s'utilitzarà els mètode socràtic vist a l'assignatura de GEP, responent les preguntes adients per tal de justificar la pròpia matriu.

	PPP (posta en marxa projecte)	**Vida útil**	**Riscos**
Ambiental	Consum del projecte moderat. Millora degut a l'ús de la *GPU* 5/10	Vida útil del *hardware* elevada. Consum electricitat i internet. Impacte ambiental 15/20	Pocs riscos a tenir en compte. Mirar de reutilitzar el *hardware* -10/0
Econòmic	Consum de recursos i temps assumible i dins els estàndards 7/10	*Software* amb vida útil potent. Cal analitzar les possible actualitzacions 15/20	Risc possible però controlat tal i com s'ha analitzat en el punt 1. -2/0
Social	Aprenentatge *CUDA* i programació en paral·lel 10/10	Vida elevada i millora optimització dels algoritmes 15/20	Pocs riscos que puguin tenir un impacte social negatiu -2/0
Rang sostenibilitat	22/30	48/60	-14/0
Rang sostenibilitat total		**53/90**	

Taula 6. Taula de sostenibilitat

9.1 Sostenibilitat ambiental

Si analitzem la matriu de sostenibilitat (veure Taula 6), veiem que el projecte es ambientalment sostenible. L'impacte ambiental és considerable degut al ús de recursos *hardware*. Són, però assumibles, degut a la utilització dels recursos *hardware* mínims i imprescindibles.

A més, el fet de treballar en amb targetes gràfiques garanteix un estalvi energètic en comparació en treballar només amb la CPU i, en conseqüència, un major estalvi energètic i menor impacte ambiental.

Per últim s'hauran de reutilitzar el recursos *hardware* quan la vida útil dels mateixos hagi finalitzat. Així aconseguirem reduir el gran impacte que aquests recursos tenen.

9.2 Sostenibilitat econòmica

Tal i com podem apreciar a la matriu el projecte és econòmicament sostenible. Per tal d'analitzar la sostenibilitat del projecte s'ha fet un estudi econòmic del cost que tindrà el projecte (veure apartat 1: gestió econòmica), de manera que veiem que realment és un projecte solvent.

Per tal de reduir l'impacte econòmic s'han utilitzat recursos *software* de codi obert que ens han permès fer el projecte encara més solvent. A més s'utilitzen els recursos *hardware* mínims que són: una màquina multi-*GPU* i un ordinador per connectar-nos a la mateixa; això ens ha permès disminuir també el cost.

Amb aquest missatge aconseguirem que els algoritmes tractats en el projecte siguin més òptims i que, per tant, les màquines que els executin, consumeixin menys energia. Caldrà estar atent a possibles actualitzacions de *CUDA* que ens poden fer augmentar el cost del projecte.

Per últim els riscos que puguin alterar la viabilitat del projecte són mínims i, pràcticament, es poden menysprear.

9.3 Sostenibilitat social

Tot bon projecte ha de tenir un impacte social positiu, tant per la societat com per els desenvolupadors del projecte. En primer lloc suposarà un gran aprenentatge per part del desenvolupador pel que fa a *CUDA* i a la programació en paral·lel. Això permetrà que aquest desenvolupador creixi com a programador i enginyer.

A més, tindrà un impacte social per aquelles persones que utilitzin els algoritmes de *clustering*, ja sigui de forma indirecta com els enginyers del *software* en molts dels àmbits relacionats amb la intel·ligència artificial o, enginyers de computadors que tractin directament amb aquests algoritmes.

Per últim cal afegir que no s'han detectat possibles àmbits on la realització del projecte pugui ser negativa per algun sector de la societat.

10 Referències

[1] Bernard Marr. Best-Selling Author, Keynote Speaker and Leading Business. Consultat a: https://www.linkedin.com/pulse/brief-history-big-data-everyone-should-read-bernard-marr

[2] Bernard Marr. What is Big Data?. Consultat a: https://www.youtube.com/watch?v=mhe5kX10CR4).

[3] HPC4Poland. Consultat a: http://www.hpc4poland.pl/en/definitions/

[4] Nvidia. About CUDA https://developer.nvidia.com/about-cuda

[5] Google Cloud Platform. Consultat a: https://cloud.google.com/compute/?utm_source=google&utm_medium=cpc&utm_campaign=2017-q1-cloud-emea-gcp-bkws-freetrial&gclid=CjwKEAjwsqjKBRDtwOSjs6GTgmASJACRbI3f9yr3FRGUNR2-inEsXDEJnoe0fepuEDnLmLeOsOf8eBoCLWLw_wcB&dclid=CKKM3qG4z9QCFQS6Gwodb-ML3A

[6] Amazon. What is cloud Computing? Consultat a: https://aws.amazon.com/what-is-cloud-computing/

[7] Andrew Ng. Clustering – Kmeans algorithm – Machine Learning. Consultat a: https://www.youtube.com/watch?v=Ao2vnhelKhI

[8] Vipin Kumar. Cluster Analysis: Basic Concepts and Algorithms. Consultat a: https://www-users.cs.umn.edu/~kumar/dmbook/ch8.pdf

[9] Campaign. Professors: *Kai J. Kolhlhoff*, *Marc Sosnick*, *Bill Hsu* [3]; de les Universitats de Stanford. Documentació plataforma *Campaign.* Consultat a: https://simtk.org/projects/campaign

[10] Nvidia. What is CUDA? Consultat a: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/

[11] Udacity. Lesson 1 – The GPU programming mode (31). Consultat a: https://classroom.udacity.com/courses/cs344/lessons/55120467/concepts/670742980923

[12] Udacity. Lesson 3 – GPU Hardware and parallel comunications patterns (29). Consultat a: https://classroom.udacity.com/courses/cs344/lessons/77202674/concepts/774332090923

[13] Udacity. Lesson 3 – GPU Hardware and parallel comunications paterns (8) https://classroom.udacity.com/courses/cs344/lessons/77202674/concepts/799933660923#

[14] Udacity. Lesson 4 – Fundamental GPU Algorithms. Consultat a: https://classroom.udacity.com/courses/cs344/lessons/86719951/concepts/877097870923

[15] CUDA BY EXAMPLE. An Introduction to General-Purpose GPU Programming. Jason Sander. Edward Kandrot. pp. 68

[16] Nvidia. NVIDIA GeForce GTX TITAN http://www.nvidia.es/object/geforce-gtx-titan-es.html#pdpContent=2

[17] Nvidia. Nvidia's Next Generation. CUDA Compute Architecture. Kepler GK110/210. Consultat a: http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[18] Github. Projecte a: https://github.com/joanorrit/campaign-1.0

[19] BOE. Consultat a: http://boe.es/buscar/act.php?id=BOE-A-2004-14600&p=20141128&tn=2