Projecte TGA
Implementació KNN en CUDA

Documentació

2n Quadrimestre - curs 2019/2020

Laia Batlle i Pau Ballber

Introducció

En aquest projecte el que es vol fer és implementar una versió de l'algorisme k-nearest neighbors utilitzant CUDA. Per fer-ho aprofitarem les eines que tenim disponibles per aconseguir a base d'optimitzacions millores en el rendiment del nostre programa.

Vam escollir aquest algorisme perquè anteriorment ja n'haviem sentit parlar en algunes de les assignatures passades i creiem que és prou simple i fàcil d'entendre. El vam començar a considerar perquè el seu codi executa varies operacions que creiem que poden ser optimitzades si s'executen en paral·lel a la GPU, cosa que ens va fer decantar per aquest i no un altre. A més, actualment s'utilitza en diversos casos per fer prediccions i extreure informació en la mineria de dades, raó per la qual també ens va resultar interessant.

Com que ja hi ha moltes implementacions seqüencials d'aquest algorisme publicades i nosaltres el que realment volem és explotar l'utilització de CUDA, el que hem fet ha estat partir d'una de les implementacions més bàsiques que hem trobat en llenguatge C per adaptar-la a la nostra manera. El codi seqüencial en el que ens hem basat es pot consultar a l'apartat bibliogràfic d'aquesta documentació.

El nostre objectiu principal per adaptar aquest algorisme a CUDA és aprofitar la utilització de kernels per paral·lelitzar part del nostre codi i obtenir millores en el temps d'execució significatives. Així doncs, podrem analitzar el rendiment del programa per a cada millora afegida o, si és el cas, investigar possibles inconsistències.

Definició del problema a resoldre

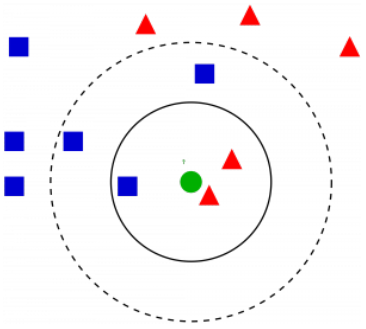
El k-nearest neighbors, o simplement kNN, és un algorisme d'aprenentatge automàtic molt utilitzat tant per a la classificació estadística com per a l'anàlisi de la regressió. En ambdós casos l'entrada consisteix en les dades d'entrenament més properes a l'espai de funcions. La sortida, en canvi, depèn de si s'utilitza per a la regressió o la classificació:

En la regressió kNN, la sortida és el valor de la propietat per a l'objecte. Aquest valor és la mitjana dels valors dels k veïns més propers.

D'altra banda, en la classificació kNN, la sortida és una pertinença a una classe. Es tracta d'una aproximació a la classificació de dades que estima la probabilitat de que un punt de dades sigui membre d'un grup o d'un altre depenent de a quin grup es troben els punts de dades que l'envolten. Més concretament, l'algorisme calcula la distància de l'element nou a cada un dels existents i ordena aquestes distàncies de menor a major per anar seleccionant el grup al qual ha de pertànyer. Aquest grup serà, per tant, el de major freqüència amb menors distàncies.

Tot i tenir diverses finalitats, aquest algorisme s'utilitza més àmpliament en problemes de classificació, per aquesta raó hem decidit centrar-nos només en aquesta part en aquest projecte.

Es pot veure un exemple de classificació kNN a la següent imatge:



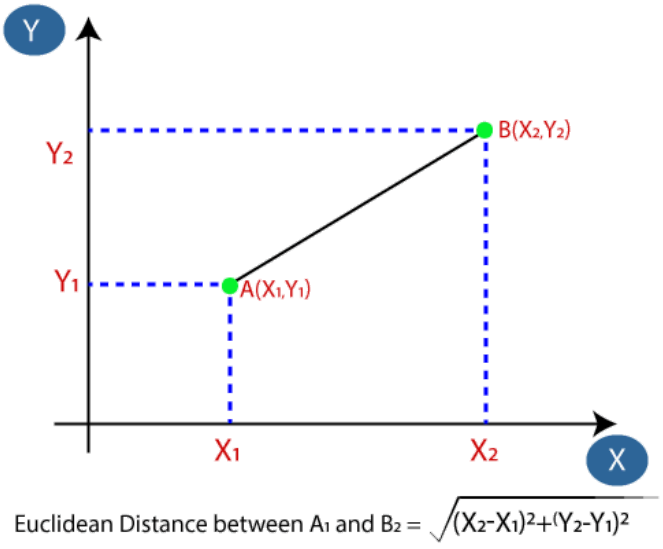
La mostra de prova (punt verd) s'ha de classificar en una de les dues categories: quadrats blaus o triangles vermells.

Si k és igual 3 (cercle interior), el nou punt s'assigna als triangles vermells perquè hi ha dos triangles i només un quadrat dins del cercle. En canvi, si k és igual 5 (cercle exterior), s'assigna als quadrats blaus, ja que hi ha tres quadrats i només dos triangles dins del cercle.

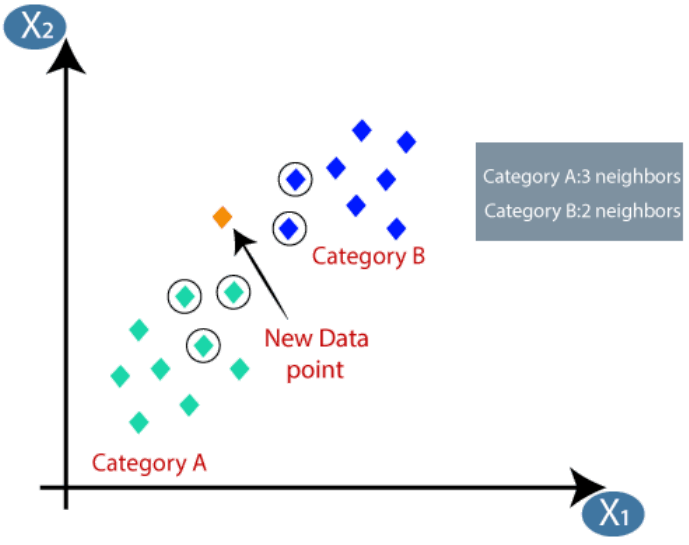
Com funciona el kNN?

Més concretament, el funcionament del kNN es pot explicar seguint els següents passos:

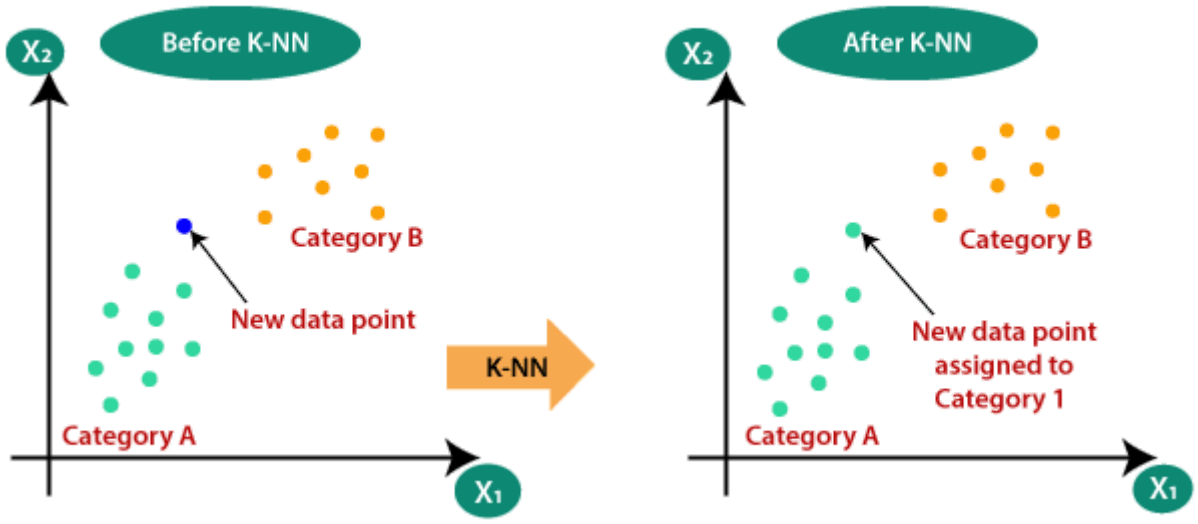
1. Seleccionar el nombre k de veïns desitjat. Per exemple, k = 5.
2. Calcular la distància euclidiana entre els punts de dades tal com es veu a la següent imatge:



3. Amb la distància euclidiana calculada s'obtenen els veïns més propers. En aquest cas, dels 5 veïns més propers, 3 són de la categoria A i només 2 de la categoria B.



4. Com hi ha més veïns de la categoria A, el nou punt de dades ha de pertànyer a aquesta categoria.



Com triar el paràmetre k?

La millor elecció de k depèn de les dades amb les que es vol treballar, però normalment, k és un nombre enter positiu, petit i imparell.

Ha de ser un nombre petit, perquè com més gran sigui k, més precisa serà la classificació, però més temps es triga a realitzar-la. A més, el fet de triar un nombre senar per a k és molt útil, sobretot en el cas d'una classificació binaria (amb només dues categories possibles), ja que això evita vots iguals entre les categories.

Particularment, si k és igual a 1, l'algorisme s'anomena nearest neighbour i l'objecte s'assigna directament a la classe d'aquest veí més proper.

Implementacions CUDA realitzades

Inicialment la nostra intenció va ser crear un programa amb dues funcions principals: una per a fer el càlcul seqüencialment tal i com estava implementat al codi que vam extreure d'internet i una altra per executar-lo utilitzant CUDA i kernels amb les optimitzacions necessàries.

knn00.cu

Així doncs d'aquí va sorgir la nostra primera versió del programa anomenada "knn00". Bàsicament, aquesta consisteix en el que s'ha mencionat anteriorment, dues funcions principals que s'executen seguidament, una funció per executar la versió seqüencial del programa i una petita implementació utilitzant un kernel per calcular la distància euclidiana entre el punt escollit sobre el que es vol fer la predicció i tots els altres punts que defineixen el conjunt de dades.

La funció seqüencial consta de tres parts: calcular les distàncies mencionades anteriorment, ordenar els punts mitjançant les distàncies de petit a gran utilitzant el sort de la llibreria algorithm i dels k elements amb distància més petita comptar la freqüencia del seu valor de predicció. Una vegada obtingudes aquestes freqüències, s'escull el grup al qual ha de pertànyer tenint en compte quina freqüència té el valor més gran.

La implementació de la funció amb CUDA es fa amb els mateixos càlculs que la seqüencial però per les distàncies s'utilitza un kernel on cada thread calcula en paral·lel la distància que hi ha entre el punt de predicció i un dels n punts del conjunt de dades. En un principi, havíem posat dins el kernel un "if" condicional per tal d'evitar que hi hagués threads que executessin el codi quan ja no era necessari, però finalment vam aconseguir fer-ho sense aquesta condició canviant els paràmetres amb els que invocàvem el kernel, optimitzant així mínimament el codi. L'intercanvi de dades entre host i device només es fa amb els paràmetres estrictament necessaris.

L'ordenació de les distàncies també canvia respecte la versió seqüencial. En aquest cas vam decidir utilitzar un altre tipus de sort, el selection sort, perquè ens calia ordenar dos vectors al mateix moment, el dels valors tenint en compte el de les distàncies.

Aquesta implementació permet introduir com a paràmetres d'execució a la línia de comandes el valor de la k (el nombre de punts a tenir en compte) i les coordenades (x,y) del punt sobre el qual es vol fer la predicció. Tal i com es pot veure en aquest exemple, s'han d'introduir els paràmetres en aquest ordre:

Comanda: `./knn.exe 20 1 2`
Definició dels paràmetres: k = 20, punt.x = 1, punt.y = 2;

Si no s'introdueix algun dels valors, aquests s'assignen per defecte. D'aquesta manera els seus valors seran: k = 15, coordenades x del punt = 2.5 i coordenades y del punt = 7.

Per inicialitzar els punts que defineixen el nostre conjunt de dades s'utilitza una variable n que indica la mida d'aquest conjunt. Aquesta està definida al codi i es creen tants punts com gran sigui aquest valor, inicialitzant-los amb valors random de coordenades (entre 0 i 99) i un valor per indicar el grup al que pertanyen (0 o 1) també aleatòriament.

Per fer una comparació de temps entre les dues funcions principals, utilitzem dos mètodes diferents. Per a la versió seqüencial s'utilitza la librería de C time.h i es calcula el temps de l'execució en milisegons. En canvi, per a la versió CUDA el que fem és crear dos events de CUDA, un a l'inici de la funció i l'altre al final perquè calculi el temps que ha passat entre els dos, també en milisegons. En aquesta també s'utilitzen els events per comprovar altres temps d'execució com per exemple el del kernel.

knn01.cu

Aquesta versió del programa és gairebé igual que l'anterior però se li afegeix un segon kernel per fer el càlcul de les freqüències dels k punts amb valors 0 i 1 que tinguin la distància més petita, és a dir, que estan més aprop del punt sobre el qual es vol fer la predicció. En un principi, teníem una variable per a cada freqüència i cada thread comprovava un dels k elements del vector de valors i, en funció de si el valor era 0 o 1, li sumava 1 a la variable corresponent. Aquestes variables després es passaven al host i allà només es consultaven per veure el resultat de la predicció. Per sumar dins del kernel i impedir que els threads accedissin a la variable al mateix temps, ho vam fer utilitzant atomicAdd. Aquesta primera versió del segon kernel va ser optimitzada a una de les següents versions.

Una altra de les diferències respecte la versió anterior és que vam afegir memoria pinned.

knn02.cu

Aquesta versió té dues grans diferències respecte l'anterior, una és que totes les variables que anteriorment eren double les vam convertir en floats, ja que no necessitàvem que la precisió fos tan gran com per utilitzar doubles i així no ens calia fer intercanvis tant grans entre el host i el device. L'altra diferència és que vam canviar el sort de la versió seqüencial perquè aquest utilitzés també el selection sort com la versió CUDA, aconseguint així una comparació més fiable dels temps d'execució. També ho vam fer per observar si la diferència de temps era només causada pel sort i no per qualsevol de les altres funcions o no.

Modificar el sort de la versió seqüencial va suposar un canvi en l'estructura de dades Point que utilitzàvem i també canvis en la inicialització de les variables, però res molt significatiu.

knn03.cu

Com que el selection sort utilitzat en la versió anterior és un algorisme molt costós en temps, vam provar de canviar-lo per altres algorismes d'ordenació més ràpids. El quicksort va ser el que ens va donar millors resultats, tot i que seguia sense suposar grans canvis.

Utilitzar kernels per executar el sort en paral·lel hauria sigut una millora molt significativa, així que se'ns va proposar utilitzar un projecte centrat en la optimització del sort fet per uns companys d'un altre grup. Ens hi vam posar en contacte i ells molt amablement ens van proporcionar les eines i el codi necessari per incorporar el seu sort al nostre projecte. Tot i que les bases no les vam implementar nosaltres, vam haver d'ajustar el codi i fer modificacions perquè funcionés correctament en el nostre cas, ja que per exemple nosaltres necessitem ordenar dos vectors alhora enlloc de només un. La versió seqüencial segueix utilitzant el quicksort.

Com s'havia dit anteriorment el codi del kernel que calculava les freqüencies també podia ser optimitzat, així doncs en aquesta versió tenim algunes diferències. En primer lloc, per evitar comprovar en tot moment que el thread no fos un assignat a un element amb posició més gran de k, vam assegurar-nos que això no pogués passar comprovant la invocació del kernel i fent-la amb els paràmetres adequats. També per acabar de treure totes les

condicions dins del kernel vam ajuntar les dues variables on es guardaven les freqüències en un sol vector de dues posicions, on el primer element conté la freqüència del valor 0 i el segon la del valor 1, canviant l'accés per no haver de cridar a cap "if".

Com que aquesta versió ja està més optimitzada i les proves les feiem amb valors més grans, també es va canviar la inicialització de les coordenades de tots els punts per un valor aleatori sense interval que ho limiti (anteriorment es feia amb un valor aleatori entre 0 i 99), així els valors que s'aconsegueixen són més diferents i els resultats són més fiables.

Anàlisi de rendiment

knn00.cu

A la primera versió va ser on ens vam adonar que el sort sería la part més costosa del nostre problema, ja que executant-ho utilitzant la versió seqüencial amb els sort de la llibreria algorithm amb una mida de n molt petita (1000), el punt sobre el qual es vol fer la predicció per defecte (coordenada x del punt = 2.5 i coordenada y del punt = 7) i la k també per defecte (15), el temps total era molt baix, 0.169000 milisegons; mentre que la nostra funció que utilitza CUDA i el selection sort tenia un temps total molt més elevat, 3.846400 milisegons. En canvi les altres operacions, com per exemple el kernel que havíem fet per calcular les distàncies tenia un temps d'execució de 0.238432 milisegons, un temps insignificant si ho comparem amb el que triga el sort.

El càlcul de freqüències, tot i que nosaltres ja teníem pensat passar-ho en un kernel per optimitzar-lo de manera seqüencial, ja tenia un temps molt baix (0.035392 milisegons) així que la diferència que ens podríem trobar seria mínima.

knn01.cu

En aquesta versió vam fer algunes de les optimitzacions que teníem pensades des del principi, com ara el kernel pel càlcul de freqüències i la memòria pinned però com que el sort ens ocupava tant temps, els canvis amb els que ens vam trobar van ser mínims tal i com ja ens imaginàvem. Per aquesta raó, vam començar a fer algunes proves augmentant el conjunt de dades i pensant en futurs canvis. Utilitzar memòria pinned amb un tamany tan petit com el que ens calia utilitzar per no arribar a temps d'execució enormes va resultar en

no poder veure si el fet d'utilitzar-la o no era una bona optimització. El mateix ens passava amb el càlcul de freqüències i, encara que vam veure que el codi podia ser optimitzat fàcilment, ho vam deixar per una futura versió.

Un altre dels problemes d'aquesta versió era que el temps de la versió seqüencial era molt més petit que el de la funció amb CUDA, per tant, per poder fer comparacions més fiables tal i com s'ha dit a la descripció de la versió knn02 ens vam proposar utilitzar el mateix selection sort en les dues funcions implementat a la següent versió.

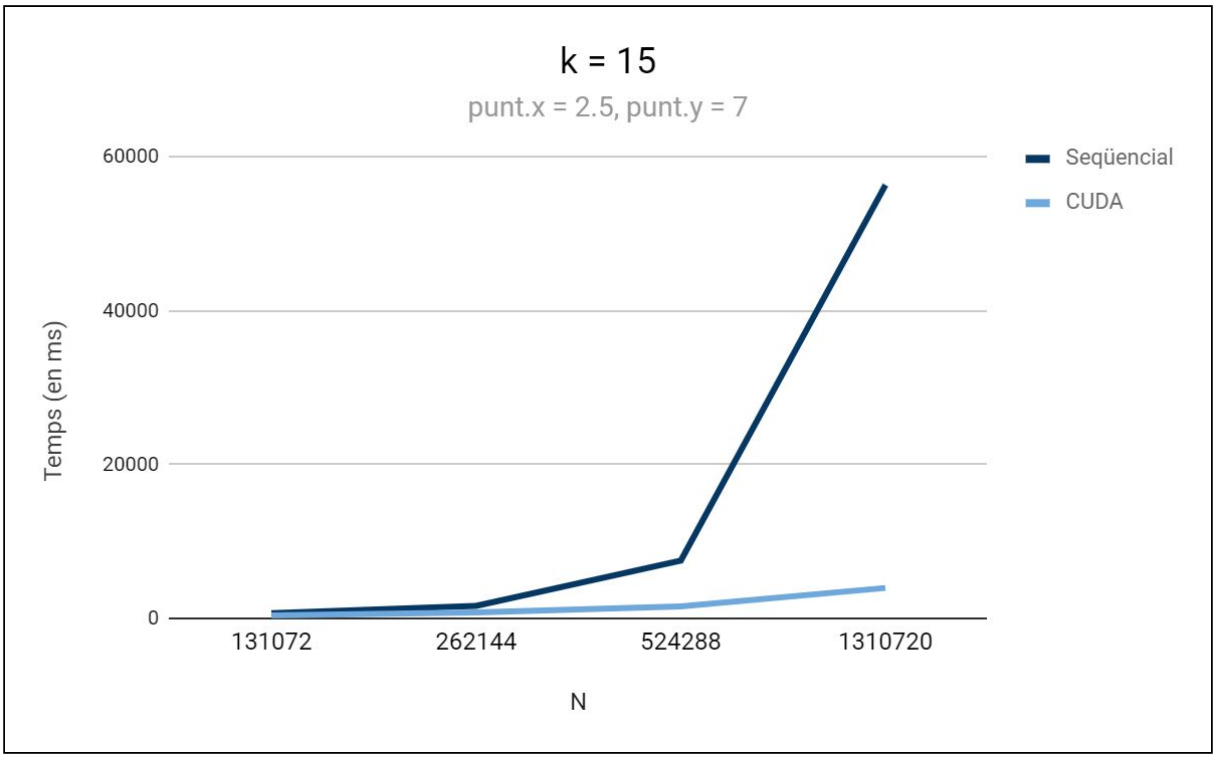
knn02.cu

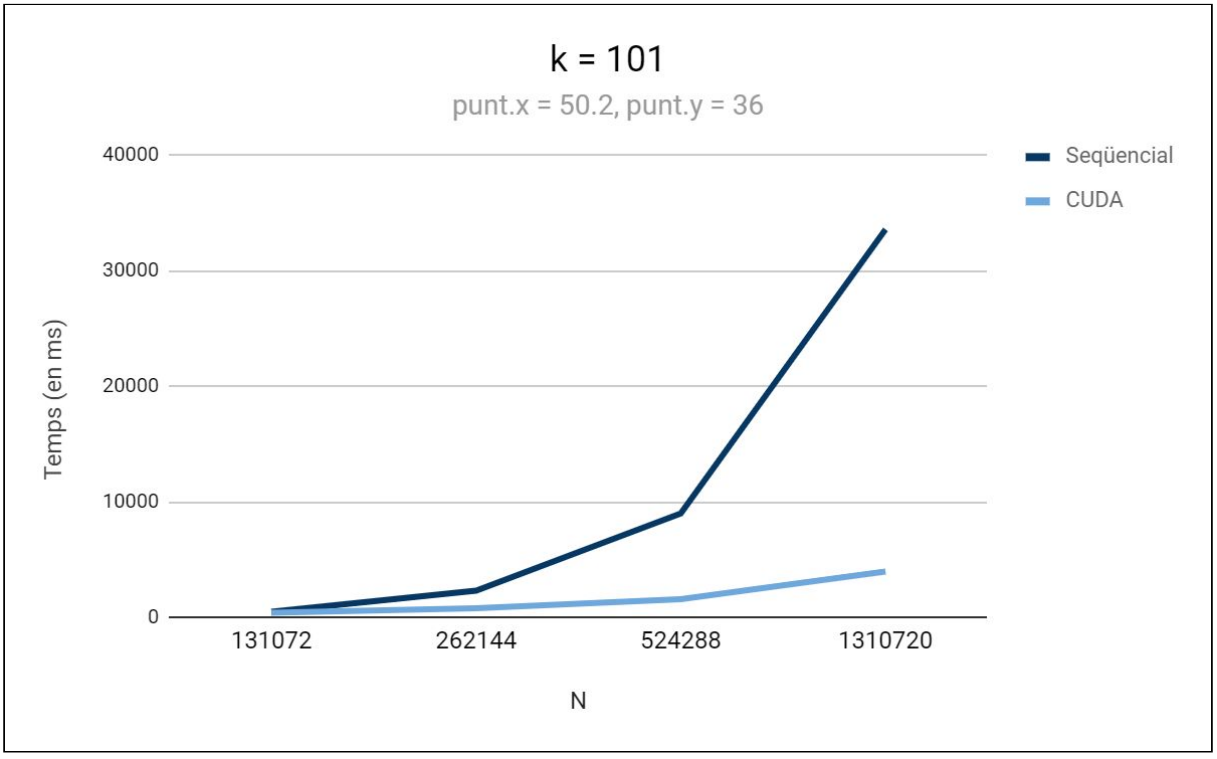
Un cop vam utilitzar el mateix sort en les dues funcions, vam veure que utilitzant un valor de n de 10000 i els altres paràmetres per defecte, la versió que utilitza CUDA era dos milisegons més ràpida que la versió seqüencial, per tant, alguna millora s'havia aconseguit tot i que fos mínima. Això ens va fer pensar en què calia canviar el sort, ja que les diferències en el temps respecte totes les altres operacions eren massa grans. A més, també érem conscients que el selection sort que s'utilitzava era una de les pitjors opcions en quant a cost en temps que es podia utilitzar ja que, tot i ser un dels més simples, requereix de $O(n^2)$ operacions per ordenar una llista de n elements.

knn03.cu

La versió final del nostre projecte ha estat la que ens ha donat un millor rendiment, arribant a ser la funció CUDA fins a 14 vegades més ràpida que la funció seqüencial. Com es pot veure en aquesta taula i en els gràfics a continuació, com més augmenta el tamany del problema més gran és la millora, cosa que és lògica ja que més codi pot ser executat en paral·lel i els kernels en treuen més profit.

	k = 15, punt.x = 2.5, punt.y = 7		**k = 101, punt.x = 50.2, punt.y = 36**	
N	**Temps Seqüencial**	**Temps CUDA**	**Temps Seqüencial**	**Temps CUDA**
131.072	658,827026	382,15292	471,966003	381,79541
262.144	1.618,862061	776,076355	2.303,318115	775,21167
524.288	7.517,395020	1.559,83252	8.978,849609	1.561,824463
1.310.720	56.360,125	3.953,549805	33.563,72656	3.952,462402



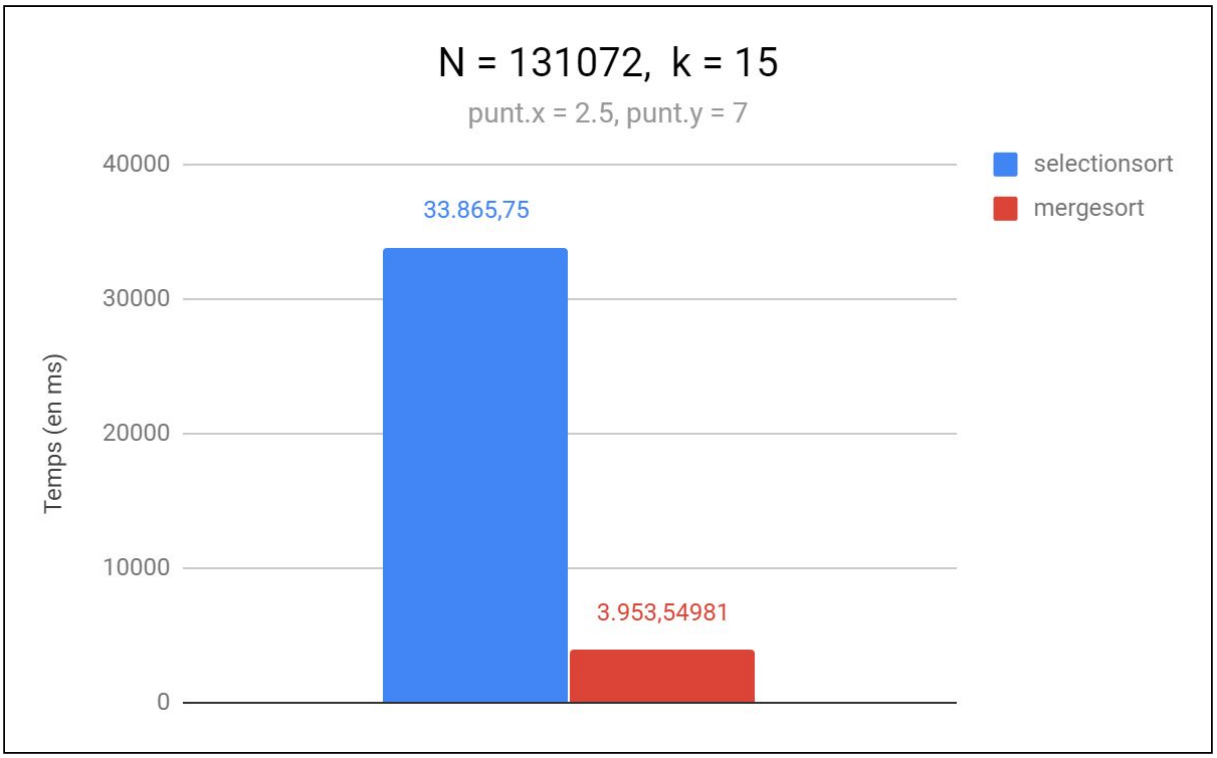


També podem observar que tot i canviar els paràmetres de la k i les coordenades del punt a tenir en compte els temps obtinguts utilitzant CUDA són gairebé els mateixos. Això a simple vista pot sorprendre ja que la k té un valor bastant més gran, però si s'analitza

detalladament podem concloure que la k només influeix en el càlcul de freqüències i aquest, tot i executar-se molt més ràpid que el sort a la versió amb CUDA, també ha estat optimitzat amb un kernel, així doncs aquesta manca de diferències es pot justificar en part per aquesta optimització. També s'ha de dir que passar de k = 15 a k = 101 és un augment poc significatiu tenint en compte la mida de la n. No obstant això, hem comprovat que provant amb k = n = 131072, com que el temps del càlcul de freqüències continua sense arribar ni a un milisegon, la diferència continua sent eclipsada pel sort.

A més, les execucions anteriors han estat repetides utilitzant memòria pinned per veure si això suposa un canvi remarcable, però els resultats no han canviat gairebé res, per tant, es pot dir que en el nostre cas no ha estat una optimització significativa. Una optimització que segurament hagués reduït considerablement el temps i que ens hagués agradat comprovar podria haver estat la utilització d'streams de CUDA, però no ha estat possible a causa de petits errors que ens hem anat trobant contínuament i que hem hagut de dedicar moltes hores a solucionar debugant codi.

Per acabar, el gràfic que hi ha a continuació simplement serveix per apreciar visualment la immensa diferència que hi ha quan s'utilitza un sort que executa paral·lelament en la GPU les operacions d'ordenació (mergesort) en comparació amb un sort seqüencial a la CPU, tot utilitzant el mateix tamany de problema (selection sort).



Bibliografia

KNN.

- Pàgina d'on hem extret el codi de l'algorisme seqüencial:
  K-Nearest Neighbours - Definition from Geeksforgeeks

- Documentació utilitzada per entendre el problema:
  - K-nearest neighbors algorithm - Definition from Wikipedia
  - What is K-Nearest Neighbor (K-NN)? - Definition from Techopedia
  - El algoritmo K-NN y su importancia en el modelado de datos - Definition from Analiticaweb
  - K-Nearest Neighbor(KNN) Algorithm for Machine Learning - Definition from Javatpoint