Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Distàncies *Word Embeddings*

Projecte Targetes Gràfiques (TGA)

Antoni Casas Muñoz
Pol Martín Garcia

Maig 2020

Problema a resoldre

El problema a solucionar és la computació de similitud de paraules utilitzant el model de *word embeddings* de *GloVe* [1] amb la mètrica de similitud de cosinus [2].

Word embeddings son una representació densa d'una paraula en un espai vectorial reduït, on la representació manté les analogies semàntiques amb operacions aritmètiques simples, com per exemple king − man + woman ≃ queen. Això permet operar amb paraules, específicament amb els seus significats, igual que es podria operar amb altres tipus de dades, permetent l'ús d'aquestes representacions per a múltiples tasques.

D'entre totes les operacions que es poden realitzar en un espai vectorial, una de les més senzilles i útils de computar és la distància de dos o més punts en l'espai, així com la cerca dels punts més propers a un altre punt concret d'entre les dades. D'aquesta manera, l'objectiu d'aquest treball és usar aquesta propietat per a poder computar distàncies semàntiques entre paraules, i cercar paraules per proximitat semàntica usant la representació vectorial d'aquestes.

Mentre que per a convertir una paraula a la seva representació densa és tan simple com accedir a un diccionari i extreure el valor, sent un procés només intensiu en espai un cop el diccionari ja ha estat generat, presenten una complicació al fer la conversió de la representació densa (*word2vec*) a la representació esparsa (el vocabulari de l'idioma en què *word2vec* va ser entrenat) aquesta conversió és especialment difícil si s'han dut a terme operacions aritmètiques amb aquesta representació.
La manera de realitzar aquesta conversió és trobar la paraula, o paraules, més properes, i per això s'utilitza la similitud de cosinus. No s'empra la distància euclidiana típica, ja que aquesta mètrica ofereix poc significat en espais amb un gran nombre de dimensions, com és el cas de molts models *word2vec*. En el nostre cas l'espai és de 300 dimensions, és a dir que cada paraula té una representació densa corresponent a un vector de 300 components, també anomenat *embedding*, i per tant la distància euclidiana no seria una mètrica vàlida.

La similitud de cosinus [2] és una mètrica de similitud que és utilitzada per ser fàcil de computar, i oferir valors en el rang de $[-1, 1]$, essent 1 el valor que indica absoluta similitud Aquesta mètrica es computa com amb la següent equació 1. La mètrica no mesura la distància en l'espai, sinó que mesura com són de paral·leles les representacions.

$$\text{cosSim}(\vec{A}, \vec{B}) = \frac{\vec{A} \bullet \vec{B}}{|\vec{A}| \cdot |\vec{B}|} \tag{1}$$

Canvis al model

Per a la implementació de l'algoritme, primer hem obtingut el model de *word2vec* de *GloVe* [1], i l'hem modificat ordenant alfabèticament les paraules i afegint les normes de cada representació densa al model propi, d'aquesta manera no és necessari computar la norma en cada operació de similitud. També, s'ha fet una millora en implementacions consecutives, on el model són dos fitxers, un amb les paraules ordenades, i un altre fitxer binari amb les representacions denses emmagatzemades. El model llavors està format per unes $2.2 \cdot 10^6$ paraules (concretament 2196016 paraules), on cada paraula és una línia al fitxer, primer la paraula, després la norma, i finalment els 300 valors de la representació densa.

És adient mencionar que cadascuna de les components de les representacions denses de les paraules es troba en el rang $[-1, 1]$, d'aquesta manera s'obté una millora de precisió en emmagatzemar i operar amb nombres de coma flotant.

Ús del software

Per poder compilar el codi font s'ha creat un sistema de `CMake`. D'aquesta manera es pot compilar el programa independentment de la plataforma.

És necessari, per al correcte funcionament de la configuració del *makefile*, que `CMake` tingui la referencia del compilador de CUDA prèviament (usualment en un arxiu `CMakeCUDACompiler.cmake`); sinó s'haurà d'emmagatzemar a l'entrada `CMAKE_CUDA_COMPILER` el *path* del compilador de CUDA.

Actualment el projecte està configurat per a usar memòria *pinned* per defecte. De no voler emmagatzemar les dades en aquest tipus de memòria cal modificar l'arxiu `CMakeLists.txt` i descomentar la línia 7, que activa la definició de compilació `NOT_PINNED_MEMORY`.
Per altra banda, el projecte també està configurat per generar codi per arquitectura Turing (*sm_ 75*). De voler generar codi per a una altra arquitectura cal modificar la línia 75 de l'arxiu `CMakeLists.txt` adientment.

Un cop el projecte ja es troba compilat, hi ha dues maneres d'executar el programa resultant.

- Dades en arxiu TXT. Passar un sol paràmetre, corresponent amb el *path* d'un arxiu de text amb les paraules, normes i *embeddings*. El fitxer usat per nosaltres es pot obtenir de https://workbench.ddns.net/nextcloud/index.php/s/mcxC38NDMMzmDgQ.

- Paraules en TXT, i binari amb dades. Passar dos paràmetres. Un primer *path* a un arxiu amb el llistat de paraules (*keys*), i un segon *path* a un arxiu binari amb les normes i els *embeddings* preprocessats (*values*). Els arxius són respectivament https://workbench.ddns.net/nextcloud/index.php/s/qtkbG6Nxx2wWLnf i https://workbench.ddns.net/nextcloud/index.php/s/6FAjH6QP6sYzf9c.

La diferència entre les dues entrades és que la segona és notablement més ràpida que la primera, ja que no necessita dur a terme el *parsing* dels nombres en coma flotant.

Amb les dades carregades, s'imprimeixen per consola els temps i altra informació interessant sobre aquesta etapa, i s'inicia l'entrada de dades per a dur a terme computacions.
Aquesta entrada ha de consistir d'una paraula arbitrària i d'un valor $0, 1$. El valor identifica si s'ha d'executar també el còmput corresponent en CPU.

A partir de la Tercera versió, es pot també introduir una operació aritmètica amb sumes i restes entre paraules per buscar paraules amb relacions similars. Per a això s'introdueix l'anomenada operació finalitzada amb el símbol "¡‘, seguit del valor $0, 1$ per executar el còmput corresponent a CPU. Aquí hi ha uns exemples d'entrada:

- `bottle 0`: computa les paraules més similars a *bottle* a GPU.

- `snake 1`: computa tant a CPU com a GPU.

- `king - queen ! 0`: computa les paraules que tenen una relació similar a l'operació anterior, a GPU.

Per cada terme introduït, és dura a terme una cerca de la paraula introduïda en la base de dades. Si aquesta hi és, es procedirà a computar i escriure per consola les 10 paraules sintàcticament més semblants en la codificació *word2vec*, així com diverses mètriques per avaluar l'eficiència de la consulta. El programa acaba quan es tanca la *pipe* d'entrada.

Implementació

Hi ha quatre versions correctament implementades del projecte, amb resultats finals equivalents. Aquestes versions també són equivalents al codi seqüencial, trobat al fitxer *main.cpp* en la funció `sequentialSearch()`. Aquesta, donat un vector d'*embeddings*, l'índex d'un d'aquests, i un nombre N, troba d'entre tots els *embeddings* del vector els N més semblants a l'*embedding* identificat per l'índex. D'aquesta manera, aquest fitxer controla l'execució de tot el programa.

Pel que fa a l'estructura del projecte hi ha els següents arxius, a part de *main.cpp* però controlats per aquest, trobem els següents.

- *kernel.cu*: Fitxer CUDA amb funcions i mètodes enllaçables amb `C` que permetent l'execució dels kernels per a dur a terme el còmput de distàncies amb un *embedding*. També permet, en les seves respectives versions, precarregar memòria a la gràfica.
- *CudaHelp.cu*: Fitxer CUDA compatible amb `C`, que inclou mètodes per a reservar memòria a la GPU des de codi `C++`, tant *pinned* com genèrica.
- *GlobalHeader.h:* Header que defineix tipus comuns entre arxius CUDA i `C++`, com el tipus dels *embeddings* i el seu emmagatzematge, de manera que les definicions són comunes entre els diferents arxius. Actualment defineix el tipus bàsic dels *embeddings* com a float, ja que aquests es troben emmagatzemats en l'interval $[-1, 1]$, i els floats ofereixen suficient resolució en aquest interval, a més de ser més ràpids que els double.
- *loader.h/cpp:* Classe per a carregar els diferents arxius a estructures compatibles amb el codi. A més inclou un algoritme de cerca binària usat per la versió seqüencial del algoritme.

Algoritme

Qualsevol dels algoritmes implementats per a solucionar aquest problema es basen en 3 parts.

1. Trobar la paraula en el vector d'*embeddings*. Aquest pas sempre es du a terme amb una cerca binària en CPU, per tant no el discutirem en aquest document.
2. Dur a terme el còmput de les distàncies de cosinus (o similituds de cosinus) entre tots els *embeddings* i l'*embedding* de la paraula cercada.
3. Filtrar els resultats, i escollir les N paraules més semblants (amb major similitud) a la paraula cercada amb les dades calculades, de manera ordenada.

Els punts 2 i 3, es troben tan implementats per a CPU, a *main.cpp*, com per GPU, a *kernel.cu*.

Càlcul de similituds

El càlcul de les distàncies o similituds es du a terme a GPU pel kernel `DotProduct()`, el qual calcula el producte escalar amb cadascun dels *embeddings*, i posteriorment en divideix el resultat pel producte de normes, segons l'equació anterior 1. Això es du a terme movent el *embeddings* de la paraula cercada a memòria *shared*, i posteriorment l'usen tots els *threads* del bloc per a dur a terme el producte escalar amb un altre *embedding*, a més es pot fer ús de sincronització dins dels *warps* per obtenir una memòria de menor latència (Vegeu millora de la Tercera versió). D'aquesta manera, cada *thread* computa una sola similitud i inicialitza en el vector de posicions la relació entre la distància i la paraula en el vector original.

Filtrat i ordenació

El filtrat per GPU requereix obté les N paraules amb una major similitud. Per a això s'ha dividit el còmput d'aquest procés en dues funcions. La primera, `FirstMerge()`, divideix el vector de similituds resultants en trossos de N elements, els quals són ordenats usant ordenació per inserció donat que N sempre serà un nombre petit. L'ordenació es du a terme *on-place*, de manera que es reutilitza la mateixa memòria per emmagatzemar el resultat.
D'aquesta forma, obtenim el vector de similituds en trossos de N elements internament ordenats. Evidentment, a part d'aquest vector de similituds s'emmagatzema un vector d'índexs a les paraules originals, per no perdre la relació entre valor de similitud i la respectiva paraula. Finalment la funció `BotchedMergeSort()` aprofita els segments ordenats per a dur a terme l'ordenació en una reducció del problema. Cada *thread* compara dos dels pedaços de N elements prèviament ordenats en un de sol.
Aquesta funció redueix el nombre de similituds a comparar a la meitat per cada crida, i es va usant fins que només resta un sol vector de N elements, el qual identifica les N paraules amb major similitud.

Canvis de versió

Aquí es llisten les característiques i canvis de cada versió entregada del codi.

Primera versió

La primera versió inclou la implementació més senzilla funcional de l'algoritme. Aquesta permet trobar les 10 paraules sintàcticament més properes a una única paraula introduïda. Les implementacions dels kernels i càrrega de dades no inclouen cap optimització, per tant aquesta versió és la menys eficient però la base del projecte. Inclou comparativa d'execució entre algoritme en CPU i GPU.

Primera implementació del producte escalar

La primera implementació ha estat la més directe. Donat un cert nombre de distàncies a computar, corresponents al nombre d'*embeddings* emmagatzemats, s'assigna un thread a cadascuna de les distàncies. D'aquesta manera no és necessari compartir informació en el bloc, més enllà de l'*embedding* inicial que, com s'ha descrit anteriorment, es troba emmagatzemat a *shared* com una constant. Així doncs, cada thread du a terme el producte escalar amb un determinat *embedding* de memòria global, i a partir d'aquest en calcula la distància de cosinus final, i l'emmagatzema juntament amb la seva posició. El següent codi n'és un abstracte simplificat, a on ja es té en memòria l'*embedding* `embedCpy`, i es disposa de la seva norma `normA`; per altra banda hi ha els *embeddings* del model `c_model` i les seves normes `c_norms`.

```
embed_t acum=0; // Acumulador del producte escalar
for(unsigned i=0;i<numEmbeds;++i) {
// Acumula el producte escalar amb el embedding escollit i un embedding
↪  determinat pel thread id
acum += embedCpy[i] * c_model[idx].data[i];
}
// Computa la distància de cosinus
```

```
distances[idx] = acum / (normA * c_norms[idx]);
pos[idx] = idx; // Assigna posició
```

Segona versió

Les millores en la segona versió del programa són separables en canvis en el codi del kernel, i en carrega de les dades a memòria. Pel que fa al kernel, s'ha reduït l'espai de memòria reservat, utilitzant memòria privada a cada *thread* per emmagatzemar l'ordenació temporal en el mètode `BotchedMergeSort()`, aquesta memòria sent memòria global. A més, s'ha afegit control d'errors complet. Per altra banda, s'ha millorat substancialment la càrrega a memòria separant l'arxiu d'input en dos, un que conté els *strings* de les paraules, i un altre que conté les normes i els *embeddings* ja en binari, per estalviar la conversió a float en temps d'execució, a més que els fitxers són de menor mida en binari. Aquesta és la principal optimització d'aquesta versió. Finalment, s'ha afegit l'opció d'usar o no memòria *pinned* segons una *flag* de compilació.

Tercera versió

La millora en la tercera versió es basa un canvi en el kernel `DotProduct`, perquè com es veurà a la següent secció Resultats, és el coll d'ampolla de l'algoritme. Per altra banda, aquesta versió inclou una nova funcionalitat, que és el càlcul de similituds amb operacions entre paraules, de manera que es poden trobar altres paraules amb relacions semblants. Això s'ha dut a terme programant un petit *parser* per pila, i posteriorment computant l'operació amb els *embeddings* de les paraules, i usant l'algoritme principal amb l'*embedding* operat.

Segona versió del producte escalar

En aquest s'han canviat els accessos a memòria per a aprofitar accessos amb coalescència, fent els accessos consecutius en 32 bytes. Per aquest propòsit, s'han destinat 8 threads per similitud a calcular, ja que els *embeddings* són floats (4 bytes), això provoca accessos contigus en memòria en 32 bytes. Vegeu $\frac{32 \text{ bytes}}{4 \frac{\text{bytes}}{float}} = 8$ floats. D'aquesta manera, 8 threads d'un mateix warp s'ocuparan de computar una sola similitud, on cada thread accedeix un valor consecutiu a l'anterior. Evidentment s'ha d'ajustar el nombre de blocs adequadament. S'ha fet ús de shared memory, reservant 4 bytes a cada thread com a acumulador, reduint els 300 elements d'un embedding en 8, finalment sent reduïts utilitzant una reducció per unrolling que culmina amb un únic thread dels 8 emmagatzemant la distància final a *global memory*. Cal mencionar que ha estat necessari dur a terme sincronització a escala de warp, ja que per molt que un warp s'executi sincronitzadament (sempre que no hi haguin divergències), la seva memòria compartida pot no estar-ho. Per a solucionar això s'ha usat `__syncwarp();`.

```
unsigned row = idx / 8; // Index de l'embedding a computar
unsigned interiorId = threadIdx.x % 8;  // Id dins de l'embedding
partial[threadIdx.x] = 0;  // Inicialitza acumulador en shared
// Computa solució parcial
for (unsigned i = interiorId; i < numEmbeds; i += 8) {
        partial[threadIdx.x] += embedCpy[i] * c_model[row].data[i];
}
// Sincronitza la memòria del warp sencer
```

```
__syncwarp();
// Reducció per reconstruir a partir de la solució parcial
if (interiorId < 4) {
        partial[threadIdx.x] += partial[threadIdx.x + 4];
}
__syncwarp();
if (interiorId < 2) {
        partial[threadIdx.x] += partial[threadIdx.x + 2];
}
__syncwarp();
if (interiorId == 0) { // Finalment computar distancia i emmagatzemar
        embed_t acum;
        acum = partial[threadIdx.x] + partial[threadIdx.x + 1];
        distances[row] = acum / (normA * c_norms[row]);
        pos[row] = row;
}
```

Quarta versió

En la quarta versió s'han eliminat *overheads* innecessaris presents al codi, com sincronitzacions explícites, reservar memòria i alliberar-la cada execució si aquesta podia ser reutilitzada. També s'ha editat el format d'output del programa per a diferenciar d'on s'obtenen els diversos temps. Més important, s'ha millorat altre cop el kernel `DotProduct`, eliminant l'ús de memòria compartida per als acumuladors, i usant sincronització entre registres d'un warp.

Tercera versió del producte escalar

Per a minimitzar la latència d'accessos a memòria es pot usar `__shfl_down_sync()` en la reducció [3], que és més eficient que dur a terme sincronitzacions de memòria compartida. Aquesta funció sincronitza la memòria de certs threads, i en recupera la informació emmagatzemada en un registre d'un altre thread. D'aquesta manera podem millorar la reducció sense haver d'usar la memòria compartida a escala de bloc.
Cal dir que dur a terme aquesta reducció requereix l'ús de màscares de threads del warp a usar, i hem intentat generalitzar-les perquè estiguin disponibles en temps de compilació de manera eficient.

```
embed_t acum = 0;
unsigned row = id / 8; // Index de l'embedding a computar
unsigned interiorId = threadIdx.x % 8;  // Id dins de l'embedding
// Computa solució parcial
for (unsigned int i = interiorId; i < numEmbeds; i += 8) {
        acum += embedCpy[i] * c_model[row].data[i];
}
// Realitza les reduccions sobre el registre de la variable acum
// FULL_MASK=0xFFFFFFFF, HALF_MASK=0x0F0F0F0F, QUARTER_MASK=0x03030303
// Reducio amb tid i tid+4
acum += __shfl_down_sync(FULL_MASK, acum, 4);
// Reducio amb tid i tid+2
```

```
13  acum += __shfl_down_sync(HALF_MASK, acum, 2);
14  // Reducio amb tid i tid+1
15  acum += __shfl_down_sync(QUARTER_MASK, acum, 1);
16  if (interiorId == 0) {
17          distances[row] = acum / (normA * c_norms[row]);
18          pos[row] = row;
19  }
```

Resultats

Els resultats han sigut obtinguts en una màquina amb una GTX 2060 SUPER com a targeta gràfica, utilitzant un slot x16 PCIe 3.0, Intel I5-9600K com a CPU, i els models guardats en una NVME, Samsung 970 EVO, amb suficient RAM DDR4 per a no observar *thrashing*.

El sistema no té límits a *pinned memory*, per tant totes les execucions s'han fet utilitzant *pinned memory* i amb CUDA 10.2. Per altra banda, els resultats aquí mostrats han estat obtinguts com a mitjana de 10 execucions sobre un set de 2196016 *embeddings*, per tant s'ha creat una nova mètrica per a computar el nombre de distàncies que es poden computar i filtrar per mil·lisegon $\frac{2196016}{\text{temps}} \frac{\text{distàncies}}{\textit{millisegon}}$.

Seguidament se'n descriuen i justifiquen les diferents millores en cadascuna de les versions. Els resultats es troben resumits en les taules, a la pàgina 8. La taula 1 compara els temps entre diverses execucions en funció de la paraula, 2 en mostra el *speedup* entre versions, i 3 un resum dels temps d'execució.

Cal dir també que tot i que s'ha millorat la velocitat de càrrega de dades en la segona versió del programa, aquesta millora no ha estat avaluada, ja que no depèn de la GPU.

Seqüencial

En la implementació seqüencial, el temps d'execució d'una consulta és de 2425.9ms, amb un error estàndard de 7.91ms, això significa que ha calculat 2196016 distàncies a una velocitat de 905.2 distàncies calculades per ms.

Primera implementació

En la primera implementació en CUDA, amb una mija de 261.4ms per execució i un error estàndard de 5.19ms, indica que ha calculat una velocitat de 8400.98 distàncies per ms. En aquesta implementació, la gran majoria del temps és transferència de dades, específicament el model d'uns 2GB que és transferit a la GPU a cada execució. En total 203ms són transferències de Host a Device, 22.6ms són l'execució del kernel de `DotProduct()`, 1.6ms el kernel de `BotchedMergeSort()`, i 1ms el kernel de `FirstMerge()`, la resta sent overheads.

La transferència de dades és 10 vegades més significativa que el kernel més costós, per tant s'ha extret en següents versions, ja que aquestes dades són constants entre execucions.

Segona implementació

En la segona implementació de CUDA, el temps d'execució d'una consulta és de 36.8ms, amb un error estàndard de 1.58ms, això significa que ha calculat 2196016 distàncies a una velocitat de 59674.3 distàncies calculades per ms. Dels 3 kernels, `DotProduct()` ocupa 22ms, `FirstMerge()`

0.8ms, i `BotchedMergeSort()` 1.3ms, la resta de temps essent overhead, per tant `FirstMerge()` i `BotchedMergeSort()` es poden considerar negligibles comparat amb `DotProduct()`.

Tercera implementació

En la tercera implementació de CUDA, el temps d'execució d'una cerca és de 17ms, amb un error estàndard de 0ms, això significa que ha dut a terme el còmput a una velocitat de 129177.41 distàncies calculades per ms. Dels 3 kernels, `DotProduct()` ocupa 6.6ms, `FirstMerge()` 1ms, i `BotchedMergeSort()` 1.6ms.

El kernel `DotProduct()` ha tingut una millora significant, gràcies als accessos amb coalescència a la memòria.

Quarta implementació

En la quarta implementació de CUDA, el temps d'execució d'una cerca és de 10ms, amb un error estàndard de 0ms, això significa que ha calculat les 2196016 distàncies a una velocitat de 219601.6 distàncies calculades per ms. Dels 3 kernels, `DotProduct()` ocupa 6.5ms, `FirstMerge()` 1ms, i `BotchedMergeSort()` 1.6ms.

És una millora molt lleugera respecte a la tercera implementació pel que fa als temps d'execució dels kernels; la millora en aquest cas ha estat en els *overheads* reduïts i la reutilització absoluta de tota reserva de memòria a la GPU, no essent mai necessari alliberar memòria entre execucions consecutives.

	CPU	GPU 1.0	GPU 2.0	GPU 3.0	GPU 4.0
ring	2405ms	273ms	35ms	17ms	10ms
key	2397ms	276ms	46ms	17ms	10ms
key	2410ms	240ms	35ms	17ms	10ms
key	2403ms	237ms	46ms	17ms	10ms
ring	2436ms	267ms	31ms	17ms	10ms
king	2468ms	266ms	35ms	17ms	10ms
king	2454ms	275ms	35ms	17ms	10ms
ring	2419ms	276ms	35ms	17ms	10ms
barca	2414ms	238ms	35ms	17ms	10ms
messi	2453ms	266ms	35ms	17ms	10ms

Taula 1: Comparativa de temps d'execució amb diferents entrades. L'ordre d'execució ha estat consecutiu de dalt a baix, per a veure si la repetició d'execucions afectava.

	Speedup respecte CPU	Speedup respecte anterior	Distàncies per ms
CPU	1	1	905.2
Primera implementació	9.28	9.28	8400.98
Segona implementació	65.92	7.10	59674.3
Tercera implementació	142.7	2.16	129177.41
Quarta implementació	242.59	1.7	219601.6

Taula 2: Evolució de la millora entre les diverses versions. La mètrica "distàncies per ms“ ha estat calculada com a $\frac{2196016}{\text{temps}}$.

	`DotProduct()`	`FirstMerge()`	`BotchedMergeSort()`	Overhead	Total
V1	22.6ms	1ms	1.6ms	236.2ms	261.4ms
V2	22ms	0.8ms	1.3ms	12.7ms	36.8ms
V3	6.6ms	1ms	1.6ms	7.8ms	17ms
V4	6.5ms	1ms	1.6ms	0.9ms	10ms

Taula 3: Comparativa de la distribució de temps entre les diverses components de les versions.

Millores descartades

Per aprofitar el fet que un *fetch* de memòria global a cache és de 128 bytes segons [4], es va implementar coalescència de memòria als 128 bytes, 64 bytes i 16 bytes, en comptes dels 32 presents a l'última versió. Això va produir resultats 5ms més lents, aproximadament. Aquest fet creiem que és degut a la cache, ja que amb el model d'execució actual, tot warp agafa 4 línies de cache i itera sobre elles, provocant 4 hits per línia abans de demanar noves, que permet una millor utilització de la cache que amb el model de coalescència a 128, 64 o 16 bytes. També es va implementar una diferent sincronització intra-warp, utilitzant una màscara sencera en la primera fase de la reducció i 0 en la resta, però CUDA no afirma que un warp es mantingui convergent fins que s'arribi a una zona divergent en el codi, vegeu [5], així que aquesta menor optimització es va eliminar a favor de posar les mascares explícites, i no sincronització implícita.

Referències

[1] "Glove: Global vectors for word representation." https://nlp.stanford.edu/projects/glove/. (Accessed on 05/06/2020).

[2] "Cosine similarity - wikipedia." https://en.wikipedia.org/wiki/Cosine_similarity. (Accessed on 05/06/2020).

[3] "Using cuda warp-level primitives | nvidia developer blog." https://devblogs.nvidia.com/using-cuda-warp-level-primitives/. (Accessed on 20/06/2020).

[4] "Cuda c++ programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-3-0.

[5] Y. Lin and V. Grover, "Using cuda warp-level primitives." https://devblogs.nvidia.com/using-cuda-warp-level-primitives/, Feb 2020.