Andrea Bellocci
Milano, 27 Dicembre 2022

Smith Waterman Algorithm

Descrizione dell'Algoritmo

Input: 2 sequenze di caratteri

Output: Un vettore che consente di trovare il modo migliore di allineare le due sequenze, eventualmente considerando inserimenti o eliminazioni di caratteri.

L'algoritmo è nato in ambito biologico per allineare sequenze di acidi nucleici; vengono definite delle penalità per l'inserimento e l'eliminazione di un carattere, oltre che per match/mismatch, in modo che variando tali parametri sia possibile variare l'output, a seconda dell'obiettivo dell'analisi di tali sequenze.

Implementazione

Sia fissa a 512 la lunghezza di entrambe le sequenze. L'algoritmo costruisce una matrice 513*513 che usa per calcolare il vettore richiesto:

- La prima colonna e la prima riga sono inizializzate a zero;
- Ogni altra cella dipende da quella immediatamente sopra, a sinistra, e in alto a sinistra: il valore finale sarà il massimo tra 0 e i valori contenuti in queste celle, sommati alle relative penalità, rispettivamente:
  1) Per la cella a sinistra, la penalità per l'inserimento di un carattere
  2) Per la cella in alto, la penalità per l'eliminazione di un carattere
  3) Per la cella in alto a sinistra, la penalità per match o mismatch, come segue: siano i e j gli indici della cella corrente; se il carattere i-1 della prima sequenza e il carattere j-1 della seconda coincidono, allora si utilizza la penalità per i match, altrimenti quella per i mismatch.

  Nota: in caso di parità, match e mismatch hanno la precedenza sugli altri, e l'eliminazione ha la precedenza sull'inserimento, in modo che il risultato sia univoco.
- Completata la matrice, si trova la cella con il massimo valore contenuto; in caso di molteplici occorrenze di tale valore, si considera, in questo caso, solo quello trovato per primo, ovvero quello che linearizzando la matrice risulta avere indice minore.
- Infine, partendo da tale cella, si costruisce il vettore di output tracciando a ritroso in modo ricorsivo la cella di provenienza, fino a trovare uno zero.

Implementazione su scheda grafica Nvidia tramite librerie CUDA

L'algoritmo di SW si presta relativamente bene ad essere parallelizzato ed eseguito su schede grafiche, dato che la matrice da costruire può essere suddivisa in parallelogrammi, le cui celle sono indipendenti le une dalle altre, e possono essere dunque calcolate simultaneamente.

Inoltre, avendo più coppie di sequenze da esaminare (in questo caso sono 1000), è possibile operare su di esse contemporaneamente, e dunque velocizzare il processo globalmente.

L'implementazione in CUDA è suddivisa in 2 momenti: inizializzazione, e calcolo della matrice. Ognuna di queste fasi è implementata in un CUDA Kernel specifico: nella fase di inizializzazione, ogni cella di ogni matrice è indipendente dalle altre, mentre nella fase di calcolo ci sono delle dipendenze. Usando due kernel separati, è possibile variare il numero di thread che operano su ciascuna matrice: nella fase di inizializzazione si può sfruttare tutta la potenza di calcolo della GPU, mentre nella fase di calcolo, che richiede alcune sincronizzazioni, il numero di thread per matrice è tenuto limitato, in questo caso, a un warp (cioè a 32 thread), in modo da ridurre i tempi morti.

Andrea Bellocci
Milano, 27 Dicembre 2022

Per prima cosa, è inizializzato il framework di CUDA: vengono allocate le risorse necessarie sulla GPU, e poi vengono copiati i dati richiesti: tutte le sequenze da confrontare sono unite insieme in due grandi stringhe, una contenente le queries, e l'altra le references, cioè rispettivamente tutte le prime e tutte le seconde sequenze generate. Viene allocato uno spazio contiguo in cui salvare tutti i risultati di tutte le matrici: per far ciò, logicamente servirebbe una matrice tridimensionale, ma siccome in CUDA non si possono avere strutture multidimensionali, nel codice vero e proprio occorre linearizzarla. Viene infine fatto spazio per le strutture che conterranno i risultati finali, da copiare a esecuzione terminata per poterli usare sulla CPU.

Fase 1: Inizializzazione della matrice
A fase terminata, tutte le celle sono inizializzate a:

- 0, se si trovano nella prima riga o nella prima colonna;
- Match_penalty se, detti i e j gli indici di tale cella, si ha che il carattere i-1 della prima sequenza coincide col carattere j-1 della seconda sequenza;
- Mismatch_penalty, altrimenti.

In tal modo, le sequenze sono richieste soltanto nella fase di inizializzazione.
Questa fase si occupa anche di inizializzare la direzione di ogni cella a invalid.

La griglia globale di thread qui usata è unidimensionale, ma ogni blocco contenuto è in realtà tridimensionale, dato che logicamente, come già accennato, si ha a che fare con una grande matrice 3D. Si ha dunque che le 1000 matrici sono suddivise in 250 blocchi diversi, il che vuol dire che ogni blocco deve inizializzare 4 matrici diverse. Queste sono distinte attraverso la coordinata z dei thread di ogni blocco, mentre le coordinate x e y sono usate per determinare, all'interno della stessa matrice, le celle di cui il corrispondente thread dovrà occuparsi. La scheda grafica messa a disposizione di Colab consente di avere blocchi di 1024 thread al massimo, per cui ad ogni matrice vengono assegnati 256 thread, suddivisi in un quadrato nelle due dimensioni x e y, cioè 16x16; ognuno di questi thread si troverà ad eseguire solo su alcune colonne di alcune righe. Siccome in questa fase le celle da considerare sono 513x513, dato che si deve aggiungere una riga e una colonna di soli zero, si ha che alcuni thread avranno più celle di altri, dato che 513 non è divisibile per 16, e quindi è anche necessario evitare di uscire dalla matrice, o di sovrascrivere le celle sbagliate. Alla fine di questa fase, viene calcolato il tempo d'esecuzione, e poi si passa alla fase di calcolo.

Fase 2: Calcolo della matrice e restituzione dei risultati
Questa fase è ulteriormente suddivisa in due momenti distinti: prima di tutto viene eseguito l'algoritmo vero e proprio, e dopo viene eseguito il traceback per la determinazione dell'allineamento migliore.

Per l'esecuzione dell'algoritmo, ogni matrice viene suddivisa in tanti parallelogrammi, contenenti solo celle tra loro indipendenti, di modo che ogni cella di un parallelogramma possa essere eseguita in parallelo con le altre. Per evitare di verificare se il parallelogramma esce dalla matrice, il calcolo viene eseguito sempre, e ma il risultato viene salvato solo se effettivamente valido. Dato che la prima colonna e la prima riga non devono essere processate, rimangono solo 512x512 celle da calcolare, e quindi ogni thread ha esattamente lo stesso carico di lavoro degli altri.
Per prima cosa vengono calcolati i 3 valori potenziali per ogni cella:

- upleft: valore della cella in alto a sinistra, sommato con il valore della cella corrente, che dalla fase di inizializzazione vale o match_penalty, o mismatch_penalty.
- up: valore della cella in alto sommato a deletion_penalty
- left: valore della cella a sinistra sommato a insertion_penalty

Viene poi cercato il massimo tra questi 3 valori e 0: a parità, 0 ha precedenza su tutto, seguito da match/mismatch, e poi da deletion; questo ordine non è assoluto, ma dipende dall'obiettivo dell'analisi.
Tale massimo viene poi inserito nella cella corrente, insieme alla direzione da cui è originato, in modo da poter tracciare nella fase di traceback la giusta sorgente.
Quando tutte le celle sono state processate, si passa alla seconda parte; ciò ovviamente richiede di sincronizzare tutti i thread che operano sulla stessa matrice, in modo da essere sicuri di utilizzare solo valori validi.
A questo punto occorre trovare il valore massimo contenuto nella matrice: per semplicità, se più celle contengono tale valore, si considera solo il primo trovato. Partendo da tale valore, seguendo le direzioni definite nella prima parte, si esegue il traceback, fino ad approdare su una cella contenente uno zero, cioè una direzione invalida. Si restituiscono i risultati trovati alla CPU, e termina così l'esecuzione dell'algoritmo.

Per l'implementazione vera e propria, come accennato in precedenza, il numero di thread per matrice è molto limitato. La griglia è ancora unidimensionale, ma stavolta contiene soltanto 125 blocchi, il che vuol dire che ogni blocco dovrà processare 8 matrici: ad ognuna vengono assegnati 32 thread, cioè esattamente un warp, ma poiché non ci sono garanzie che tutti questi thread appartengano effettivamente allo stesso warp, è necessario sincronizzarli esplicitamente. Ciò può essere fatto solo tramite la funzione __syncthreads, che però sincronizza tutti i thread in uno stesso blocco. Per questo motivo, il numero massimo di thread per blocco è ora solo 256, e ogni blocco non è più tridimensionale, ma è soltanto in 2D.
Si ha allora che le righe della matrice sono raggruppate in 16 strisce di 32 righe ciascuna: ognuna di queste righe è interamente affidata a un singolo thread, che processa una cella per volta insieme agli altri, procedendo appunto a parallelogramma, ed evitando di uscire dalla matrice. Ogni thread dunque processa 544 celle per striscia, di cui però 32 non vengono considerate poiché esterne.
Per quanto riguarda il calcolo del valore massimo, esso può essere fatto contemporaneamente all'esecuzione dell'algoritmo: ogni thread mantiene un proprio valore massimo, aggiornato se necessario alla fine del calcolo di ogni cella, e alla fine lo inserisce in un vettore appositamente creato. Dopo la sincronizzazione, un solo thread per matrice si occupa di trovare il maggiore tra i massimi contenuti nel vettore, replicando la convenzione di considerare solo il "primo" massimo in caso di parità, cioè confrontando gli indici di ciascuno di essi. Tale vettore è allocato in memoria condivisa, dato che serve solo all'interno di uno stesso blocco, e non deve essere ritornato alla CPU.
Infine, lo stesso thread che trova il massimo si occupa di eseguire il traceback e di ritornare al chiamante i risultati in modo corretto.

Osservazioni e analisi dei risultati

Al termine dell'esecuzione dell'algoritmo, i risultati vengono confrontati con quelli prodotti dallo stesso algoritmo, eseguito stavolta sulla CPU. In particolare, vengono confrontati i valori contenuti dai vettori CPU_res e GPU_res, cioè il massimo di ciascuna matrice, e quelli contenuti in CPU_simple_rev_cigar e GPU_simple_rev_cigar, cioè le direzioni di volta in volta prese durante il traceback.
Il primo confronto dà esito positivo: i valori coincidono esattamente ad ogni esecuzione. Tuttavia, nel secondo confronto si riscontrano delle discrepanze.
Modificando leggermente il codice, è possibile cercare cosa è andato storto: in particolare, ponendo in CPU_res e in GPU_res, al posto dei valori massimi, le coordinate di tali valori (ad esempio prima la i e poi la j), si nota che ad ogni esecuzione esse coincidono esattamente ad ogni esecuzione, segno che la prima parte dell'implementazione in CUDA è corretta. A questo punto, l'unica possibilità è che sia errata la parte in cui vengono assegnate le direzioni da seguire ad ogni cella. Tuttavia, dopo molteplici tentativi, non è stato possibile individuare un errore, e neppure copiando il codice della CPU all'interno del kernel è stato

possibile eliminare le discrepanze. Analizzandole ulteriormente, è emerso che ad ogni esecuzione, le differenze tra i due vettori sono circa 18mila (su un totale di 1024000 elementi, cioè circa l' 1,7%), e indicano sempre che nell'implementazione in CUDA il traceback termina mediamente prima della versione per la CPU: tutte gli errori infatti si verificano esclusivamente per direzioni invalide nel vettore GPU_simple_rev_cigar, e direzioni invece valide nell'altro, il che suggerisce appunto che per un motivo ignoto, la strada per raggiungere una cella a zero è più breve nella versione per GPU che in quella per la CPU. Di conseguenza, la versione in CUDA offre un allineamento leggermente più corto (circa 18 caratteri in meno, il 3%) rispetto a quello offerto dalla versione software, ma a parità di punteggio ciò dovrebbe comportare una migliore corrispondenza tra i singoli caratteri.

Conclusioni

Il file SmithWaterman.cu contiene tutto il codice del progetto; il main, in particolare, esegue prima l'implementazione sulla CPU, e poi passa alla versione in CUDA. Infine controlla i risultati e li stampa, insieme ai tempi d'esecuzione delle due versioni.

Per compilare il codice è stato utilizzato il comando "nvcc SmithWaterman.cu -O3 -o a.out"; l'eseguibile così ottenuto in media mostra tempi d'esecuzione pari a circa 2 secondi sulla CPU, e a circa 0.3 secondi sulla GPU; mediamente dunque la versione in CUDA è circa 9 volte più veloce della versione in C. Rimuovendo l'opzione "-O3", il compilatore produce un codice per la CPU circa 3 volte più lento, mentre la velocità della versione hardware rimane invariata, segno che il codice CUDA non subisce ottimizzazioni diverse nei due casi. Non avendo una scheda grafica Nvidia, sia trovare gli errori sia tentare di ottimizzare manualmente il codice (ad esempio variando il numero di thread per matrice, la dimensione della griglia e la multidimensionalità dei blocchi) è risultato particolarmente lungo e complesso, e ora della fine non sufficiente per risolvere tutti i problemi. Neppure eseguendo il traceback in un kernel apposito, con differenti disposizioni di thread, il tempo d'esecuzione su scheda grafica varia in modo apprezzabile, né è stato possibile ottenere lo stesso allineamento prodotto dalla versione software (è possibile compilare il codice in modo che il traceback sia separato dal resto, usando l'opzione "-D SEPARATE_TRACEBACK").

In generale, sono state utilizzate tutte le funzionalità di CUDA presentate a lezione, esclusi gli strumenti di debug e profiling, che apparentemente richiedono di avere una scheda grafica Nvidia per funzionare; questo ha ovviamente influito negativamente sulla scrittura del codice, e verrà tenuto in considerazione in eventuali futuri acquisti di nuovi computer/schede grafiche.

Andrea Bellocci