PROGRAMMAZIONE MULTICORE E MANYCORE

ANNO 2016/2017

RELAZIONE RISOLUZIONE PROBLEMA 1 & 2

PAOLO SALOMÈ

MATRICOLA 0233502

PROBLEMA 1

È stato richiesto di effettuare una parallelizzazione tramite OpenMP e di sviluppare una versione CUDA di un codice per l'apprendimento supervisionato di una rete neurale multilivello.

Nella rete neurale la fase di apprendimento è suddivisa in due parti ben distinte:

- una fase detta *feedforward* in cui l'output di ogni neurone diventa parte dell'input per ogni neurone del livello successivo.
- Una fase detta *backpropagation* in cui l'errore tra il valore di uscita atteso e quello effettivo viene propagato partendo dall'ultimo livello per andare verso il primo.

PROBLEMA 1 OMP

Per fornire una soluzione in OpenMP sono stati riordinati tutti i cicli dell'algoritmo per accedere ad elementi contigui e sono state rese parallele diverse parti del codice iniziale. Tutto l'algoritmo è compreso in una pragma omp parallel al cui interno alcune variabili sono rimaste private in modo tale da avere una esecuzione concorrente dei thread. Per dividere il lavoro equamente tra i thread i pattern gli vengono assegnati in base al loro id ed ognuno esegue tutta la fase di feedforward e di backpropagation in modo indipendente dagli altri.

Al termine della fase di feedforward, nel ciclo per calcolare l'errore dell'uscita ottenuta rispetto a quella attesa ed il DeltaO, si calcola anche il threadDeltaHO, ossia la parte di DeltaWeightHO derivante dal contributo del singolo thread .

Nella fase di backpropagation ogni thread calcola parte delle matrici DeltaH2H (ogni thread ha solo parte dei pattern totali) ed il proprio threadDeltaH2H.

Una volta terminata l'iterazione dei pattern si trova una barriera inserita per garantire che tutti i thread abbiano calcolato l'errore parziale, così che uno solo calcoli l'errore derivante dai diversi contributi. Infine ogni thread aggiorna le matrici dei pesi Weight e dei DeltaWeight in modo indipendente suddividendosi le colonne in base al proprio id.

PROBLEMA 1 CUDA

Vista la natura dell'algoritmo di apprendimento della rete neurale composto da diversi prodotti matriciali si è pensato di parallelizzare il più possibile tali prodotti trai diversi blocchi della griglia. Inoltre, vista la grande dimensione delle matrici si è optato per fare i prodotti tra matrici a blocchi; tali blocchi si generano adagiandovi sopra una griglia di dimensione fissata la quale si sposta lungo la "matrice guida" di una determinata fase dell'algoritmo.

A differenza del codice di partenza dove per le matrici si utilizzavano più livelli di puntatori (ossia *H2H, W, DeltaH2H, DeltaW;* qui troviamo anche *Bias* e *DeltaBias* che sono state separate dalla matrice W e DeltaW), qui se ne usa uno soltanto. Tale scelta è stata fatta per poter trasferire i dati da HOST a DEVICE in una volta sola evitando latenze derivanti dall'uso di più chiamate. Ma questo porta a dover utilizzare degli array di supporto per tener conto dei diversi offset delle matrici di ogni livello (*H_matrix_H2H_index, H_matrix_DELTA_index, H_matrix_W_index, H_matrix_B_index*).

Per la matrice Target (quella contenente gli output attesi) si è usata la memoria Texture in quanto i dati non vengono mai aggiornati e c'è località spaziale nell'accedere alla matrice.

Nel caso della matrice Target l'uso della costant memory è sconsigliato in quanto i thread accedono tutti ad elementi differenti , ma è risultato adatto per salvare le variabili *alpha* ed *eta*. Infatti tale variabili utilizzate nella fase di *backpropagation* sono accedute in contemporanea da tutti i thread e questo comporta la possibilità di fare broadcast.

Per aumentare il parallelismo sono stati utilizzati i Cuda Streams. L'uso degli streams garantisce che le operazioni sul singolo stream siano eseguite nell'ordine stabilito mentre le operazioni in diversi stream possono essere alternate e quando possibile eseguite in contemporanea. I vantaggi di questa scelta sono utili per parallelizzare l'utilizzo della memoria e l'esecuzione. Per mitigare la grande latenza nella copia della matrice di Input, nella funzione di feedforward e solo nella prima epoca, si è utilizzata la cudaMemcpyAsync di una porzione della matrice, una per ogni stream.

Feedforward

Nella fase *feedforward* si calcola il prodotto matriciale tra la matrice H2H(o Input Mat al primo step) del layer corrente e la matrice dei pesi W sita tra il livello locale e quello successivo, ossia si calcola la matrice H2H del layer successivo.

Vista la grandezza di queste matrici si è pensato di adottare una *GRID* di grandezza fissata la quale slittasse sulla matrice di destinazione lungo un solo asse. Tale GRID ha dimensioni che dipendono anche dal livello analizzato, ossia è fissata in modo tale da poter coprire interamente una riga della matrice, ma anche in modo da evitare uno spreco di blocchi di thread.

È stato adottato il seguente approccio per il prodotto matrice-matrice:

1. La griglia di blocchi viene “posizionata” sulla matrice H2H del livello successivo;
2. Ogni blocco della griglia è così adibito a calcolare la sua porzione di matrice;
3. Ogni thread del blocco calcola il prodotto riga-colonna una porzione per volta: le matrici H2H del livello corrente e la matrice dei pesi W vengono fatte a blocchi, ed ogni thread calcola il prodotto riga-colonna relativo ai blocchi del round, per poi passare a quelli successivi sommando i contributi di ogni blocco.
4. La griglia viene fatta traslare e si ripete il procedimento finché la matrice di destinazione non è completata.

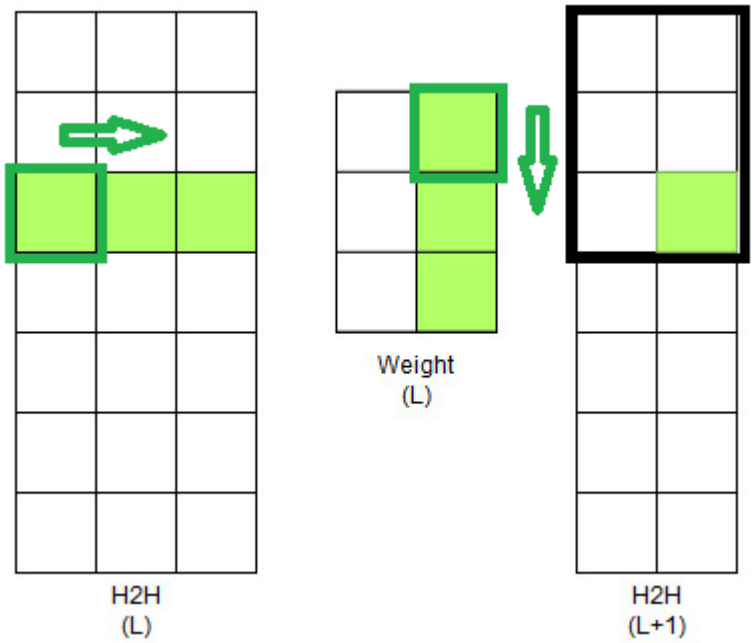

Fig.1: feedforward, calcolo H2H da parte di singolo blocco (evidenziati i round e griglia)

In questa fase si invocano:

- Feedforward: è la funzione invocata ed eseguita dall’HOST per coordinare l’intera fase. Si occupa di suddividere il lavoro tra i diversi streams fissando gli opportuni offset, dipendenti da un determinato stream e da un determinato layer. Per ogni stream e layer invoca il kernel *MMMulDevFeed* . Infine invoca anche il kernel *deviceReduceBlockAtomicKernel* ma su un solo stream in modo da calcolare l’errore totale della rete neurale.
- MMMulDevFeed: in questo kernel eseguito dal DEVICE si calcola l’offset della griglia lungo la matrice H2H, si sposta la griglia di una quantità pari alla grandezza dei blocchi*numero blocchi lungo l’asse y (gridDim.y*blockDim.y) e si invoca il kernel *MMMulDevPartialFeed* considerando l’offset legato alla posizione corrente della griglia ad ogni spostamento di quest’ultima.
- MMMulDevPartialFeed: in questo kernel eseguito ed invocato dal DEVICE ogni blocco calcola il prodotto riga colonna tra blocchi appartenenti alla matrice H2H e la matrice W. Vengono utilizzate delle matrici di appoggio site nella shared_memory grandi quanto il blocco in cui salvare il blocco corrente delle matrici su cui effettuare il prodotto.

Ogni thread carica dalla memoria globale un singolo elemento delle matrici per poi salvarlo in quelle nella shared: se il thread dovesse avere un indice tale da ritrovarsi fuori dai bordi delle matrici allora salva 0 (si utilizzano operatori ternari così da non spezzare il warp) . A questo punto ogni thread (dopo una sincronizzazione con gli altri thread del blocco) calcola il prodotto riga-colonna dei blocchi correnti,somma il contributo in una variabile locale e passa ai blocchi successivi ripetendo le operazioni appena descritte. Una volta ottenuto il prodotto completo ogni thread può calcolare la funzione di uscita e salvarla su H2H del livello successivo.
In più all'ultimo layer ogni thread calcola una parte della matrice di errore e della matrice delta dell'ultimo livello.

- deviceReduceBlockAtomicKernel: in questo kernel si calcola l'errore totale. Ogni thread del blocco somma gli elementi lungo tutta la matrice di errore di posizione pari al multiplo del suo indice all'interno della griglia (blockIdx.x*blockDim.x+ threadIdx.x) che in questo caso è lineare. Fatta questa prima riduzione ogni thread del blocco esegue il kernel *blockReduceSum*; al cui termine un solo thread (thread 0) effettua una atomicAdd verso variabile globale in cui salvare l'errore.
- blockReduceSum: effettua la riduzione di un array a livello di blocco. Ogni thread esegue una riduzione a livello di warp (*warpReduceSum*) e se è il thread designato all'interno del warp (threadIdx%warpSize==0) salva il valore ottenuto in un array in memoria shared. I thread con indice inferiore al numero di warp contenuti nel blocco assegnano ad una variabile il valore letto dall'array nella shared mentre gli altri assegneranno 0. A questo punto si effettua un'altra riduzione a livello di warp e si restituisce il risultato.
- warpReduceSum: effettua la riduzione a livello di warp.

Backpropagation

Nella fase *backpropagation* si calcolano le matrici DeltaWeight, e di conseguenza DeltaH2H , con lo scopo di riaggiornare le matrici dei pesi della rete neurale in ogni livello da cui è composta.

Vista la grandezza di queste matrici si è pensato di adottare una *GRID* di grandezza fissata la quale slittasse sulla matrice di destinazione lungo entrambi gli assi. Tale GRID ha dimensioni che dipendono dal livello analizzato, ossia è fissata in modo tale da poter massimizzare la copertura della matrice DeltaWeight ed il numero di blocchi in esecuzione concorrente, ma anche in modo da evitare uno spreco di blocchi di thread.

È stato adottato il seguente approccio per il prodotto matrice-matrice:

- La griglia di blocchi viene "posizionata" sulla matrice DeltaWeight sita tra il livello corrente e quello successivo;
- Ogni blocco della griglia è così adibito a calcolare la sua porzione di matrice DeltaWeight in modo indipendente mentre collabora con i blocchi adiacenti lungo la riga della griglia per il calcolo della matrice DeltaH2H;

- Vengono esaminate le matrici DeltaH2H del livello successivo e H2H del livello precedente blocco dopo blocco; mentre il blocco della matrice dei pesi W (trasposta) viene caricata una volta sola, questo perché tale matrice ha le stesse dimensioni della DeltaWeight sul quale si "poggia" la griglia. Per quanto riguarda DeltaWeight ogni thread del blocco calcola il prodotto di ogni elemento della riga H2H per un elemento della matrice DeltaH2H (quello con indice di blocco del thread) e lo salva in un array locale, passando quindi ai blocchi di matrice successivi su cui ripetere l'operazione per poi aggiornare l'array locale.
  Per quanto riguarda la matrice DeltaH2H invece ogni thread del blocco deve fare un prodotto riga-colonna tra DeltaH2H e la trasposta della matrice W per poi moltiplicarlo per una parte dipendente dall'elemento appartenente al blocco della matrice H2H analizzato, fatto questo si passa ai blocchi successivi.
  Thread con stesso indice threadIdx.y contribuiscono agli stessi elementi della matrice DeltaWeight mentre quelli con stesso threadIdx.x e threadIdx.y (quindi in differenti blocchi) contribuiscono allo stesso elemento della matrice DeltaH2H.
- La griglia viene fatta traslare, se necessario, e si ripete il procedimento finché la matrice di destinazione non è completata.
- Fase di riduzione della matrice DeltaWeight dei contributi generati nei diversi streams.

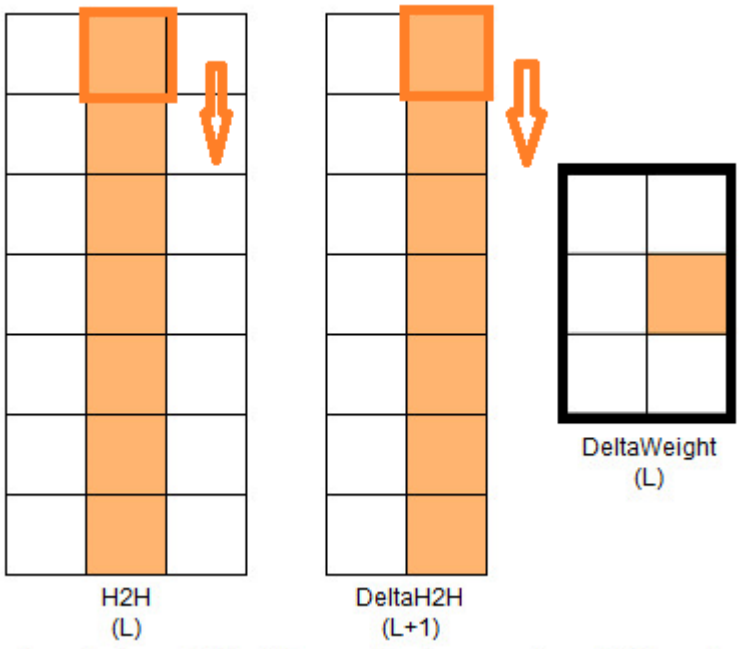

Fig.2: backpropagation calcolo DeltaWeight(L) di un singolo blocco (evidenziati i round e la griglia); la matrice DeltaH2H(L) combacia per dimensioni con H2H e per questo non riportata nella figura

Fig.3: backpropagation, dettaglio calcolo DeltaWeight su singolo blocco

In questa fase si invocano :

- Backpropagation: è la funzione invocata ed eseguita dall'HOST per coordinare l'intera fase. Si occupa di suddividere il lavoro tra i diversi streams fissando gli opportuni offset, dipendenti da un determinato stream e da un determinato layer. Per ogni stream e layer invoca il kernel *MMMulDevBack* per il calcolo delle matrici DeltaWeight (e DeltaBias) "temporanee" (una per stream) tenendo conto degli offset della griglia rispetto alla matrice su cui è adagiata. Una volta completate le operazioni per ogni livello si avvia la fase di riduzione per aggiornare la matrice DeltaWeight e quella dei pesi W di ogni livello invocando il kernel *MMMulDevBack* su un solo stream.
- MMMulDevBack: in questo kernel eseguito dal DEVICE ogni blocco calcola l'offset del blocco per determinare quale colonna della matrice H2H del livello corrente, quale colonna della matrice DeltaH2H del livello successivo, quale colonna della matrice DeltaH2H del livello corrente e quale porzione di DeltaW calcolare. Il tutto per poi invocare il kernel *MMMulDevPartialBack* .
- MMMulDevPartialBack: in questo kernel eseguito ed invocato dal DEVICE ogni blocco esegue il calcolo della parte della matrice DeltaW assegnata ad esso e della matrice DeltaBias se abilitato (solo i blocchi con blockIdx.y ==0). Si utilizzano diverse matrici di appoggio site nella shared memory con lo scopo di ospitare un blocco della matrice W, il quale viene caricato una volta sola, un blocco per la matrice DeltaH2H ed uno per H2H per ospitare la porzione di matrici che si analizza in quel turno. Si calcolano le soglie per delimitare i limiti delle matrici da cui caricare dati validi. Ogni thread salva nella matrice dedicata alla matrice W la sua trasposta (un solo caricamento a thread) oppure il valore 0 se tale thread supera la soglia consentita.
  A questo punto si analizzano blocco dopo blocco le colonne delle matrici H2H e DeltaH2H e ad ogni turno :
  - si caricano i dati nella shared memory
  - Per quanto riguarda DeltaWeight ogni thread del blocco calcola il prodotto di ogni elemento della riga H2H per un elemento della matrice DeltaH2H (quello con indice di blocco del thread) e lo salva in un array locale(somma i contributi turno dopo turno).
  - Per quanto riguarda la matrice DeltaH2H invece ogni thread del blocco deve fare un prodotto riga-colonna tra DeltaH2H e la trasposta della matrice W per poi moltiplicarlo per una parte dipendente dall'elemento appartenente al blocco della matrice H2H. Si esegue un'atomic add del valore ottenuto verso la cella della matrice DeltaH2H del livello corrente.
  - Se il blocco è abilitato somma alla matrice dei bias sita nella shared sommando il DeltaH2H del livello successivo.

  Giunti al termine delle colonne si sommano i contributi generati dai diversi thread salvati negli array locali per il calcolo di DeltaW attraverso delle atomicAdd . Thread con stesso threadIdx.x contribuiscono agli stessi elementi della matrice DeltaW(quella dello stream).

  Se il blocco è adibito al calcolo della matrice DeltaBias ed il thread appartiene alla prima riga del blocco, tali thread effettuano una riduzione lungo l'asse y dei valori raccolti nella matrice

dedicata sita nella shared memory per poi salvare nella matrice nella memoria globale(quella dello stream)

- MMMulReduction: in questo kernel eseguito dal DEVICE ogni blocco calcola l'offset del blocco all'interno della matrice DeltaW sita tra il livello corrente ed il successivo e quello di DeltaBias per poi invocare il kernel *MMMulReductionBlock* con lo scopo di ridurre le matrici DeltaW e DeltaBias"temporanee"(una per stream) in un'unica DeltaWeight.
- MMMulReductionBlock: in questo kernel ogni thread del blocco somma i contributi delle matrici "temporanee" di stream DeltaW e DeltaBias per aggiornare le matrici DeltaWeight, DeltaBias,W e Bias di origine.

Tabella tempi

Tutti i test sono stati effettuati su STELLA. Qui di seguito riporto una tabella con i tempi di esecuzione dei tre scenari.

Implementazione originale		
Epoca 1	*Epoca 2*	*Epoca 3*
174 sec	170 sec	162 sec
1087.503174	1087.407715	1087.316162
Implementazione OMP		
Epoca 1	*Epoca 2*	*Epoca 3*
14.995447	14.996415	14.957864
1087.767578	1085.885986	1084.461914
Implementazione CUDA		
Epoca 1	*Epoca 2*	*Epoca 3*
0.12234	0.061533	0.055642
1087.887939	1086.005371	1079.754883

PROBLEMA 2

Per risolvere il problema del calcolo della *prefix sum* si è adottato l'approccio che utilizza una fase di *up-sweep* ed una di *down-sweep.* Nell'algoritmo proposto si calcola la prefixSum in ogni warp per poi fare una riduzione per sommare ad ogni warp i contributi dei warp che lo precedono.

Nella soluzione si sono utilizzati tre kernel:

- Ssb_prefix_sum: in questo kernel il blocco intero si sposta lungo l'array in ingresso calcolando la prefix sum degli elementi compresi nella dimensione del blocco. Ad ogni passo:
    1. Ogni thread effettua il calcolo della prefixSum all'interno del warp attraverso il kernel *ssb_warp_prefix_sum*, passando il valore letto dal vettore di input ed un puntatore ad un array sito in memoria condivisa per tener conto della somma di tutti gli elementi del warp (*block_prefix*).
    2. A questo punto nell'array condiviso troviamo nella posizione [i] la somma di tutti i valori letti dai thread di quel warp. Quindi è possibile effettuare una riduzione a livello di warp (effettuata solo dal primo warp tramite il kernel *warp_reduction*) di tale array per aver in posizione [i] il valore precedente più la somma dei valori letti da tutti i thread appartenenti ai warp precedenti al warp [i].
    3. Ogni thread salva in output la somma tra la prefixSum calcolata al punto 1 + *block_prefix*[i-1]+ *first* (prefixSum cumulativa del passo precedente). L'ultimo thread del blocco che rientra nei limiti dell'array aggiorna *first.*
- Ssb_warp_prefix_sum: in questo kernel si esegue l'algoritmo composto dalle fasi di up_sweep e down_sweep. Al termine di up-sweep l'ultimo thread del warp salva il suo valore corrente, pari alla somma di tutti gli elementi del warp, nel puntatore che ha come parametro (ossia salva la somma degli elementi del warp di appartenenza) e poi azzera val come previsto dall'algoritmo e continua con la fase down-sweep.
- Warp_reduction: si effettua una riduzione nel warp.

Block sum 0

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
up-sweep & down-sweep								up-sweep & down-sweep							
0	0	1	3	6	10	15	21	0	8	17	27	38	50	63	77
somma contributi warp precedenti e valore blocco dell'iterazione prec							*28*	somma contributi warp precedenti e valore blocco dell'iterazione prec							*92*
0	0	1	3	6	10	15	21	28	36	45	55	66	78	91	105

Block sum 120

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
up-sweep & down-sweep								up-sweep & down-sweep							
0	16	33	51	70	90	111	133	0	24	49	75	102	130	159	189
somma contributi warp precedenti e valore blocco dell'iterazione prec							*156*	somma contributi warp precedenti e valore blocco dell'iterazione prec							*220*
120	136	153	171	190	210	231	253	276	300	325	351	378	406	435	465