Prodotto matrice sparsa-vettore parallelo

SCPA project a.y. 2024/2025

Stefano Belli, matricola 0350116

Università degli Studi di Roma "Tor Vergata"

Agenda

Il problema da affrontare

Si vuole effettuare una moltiplicazione matrice sparsa-vettore (SpMV):

$$y = Ax$$

- Rappresentare una matrice sparsa A interamente è un'inutile spreco di memoria.
- Usare la rappresentazione intera per l'SpMV implica l'esecuzione di operazioni il cui risultato è noto ($0 \cdot a = 0 \ \ \forall a$), quindi spreco di risorse di calcolo
- Occorre fare affidamento a dei formati di rappresentazione alternativi

Formati di rappresentazioni delle matrici sparse

Per ovviare al problema, si possono rappresentare in memoria le matrici con vari formati di rappresentazione, ne esistono moltissimi ma quelli che ci interessano sono:

- **COO** - COOrdinate format
- **CSR** - Compressed Storage by Rows
- **ELL** - ELLPACK
- **HLL** - Insieme di blocchi ELL

Implementazione di SpMV problematica

Possiamo dire che per tutti i formati, implementare la SpMV è problematico: la rappresentazione compressa e l'indirizzamento indiretto implicano un **maggior numero di operazioni verso la memoria** (risp. alle fl.ops.) e l'**impossibilità di sfruttare appieno la cache locality**.

Conversione dei formati delle matrici sparse

- Le matrici scaricate dal sito `https://sparse.tamu.edu/` e salvate sul filesystem
- Le matrici vengono lette dal file in formato MatrixMarket con l'ausilio di `https://math.nist.gov/MatrixMarket/mmio/c/mmio.h` e `https://math.nist.gov/MatrixMarket/mmio/c/mmio.c` e convertite in COO
  - Avviene la conversione in COO di matrici che siano reali o pattern e generali o simmetriche
  - Se la matrice è simmetrica bisogna ricostruire il triangolo mancante
  - Bisogna tenere opportunamente traccia degli explicit zeroes
  - Bisogna riportare gli indici in base 0
- Da COO, le matrici vengono convertite, quando necessario, in CSR o HLL

Misurazione delle prestazioni

Le misure di prestazioni sono avvenute sul server di dipartimento, che ha le seguenti **caratteristiche tecniche**:

- CPU: Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz (40 CPUs)
- NUMA:
  - NUMA node(s): 2
  - NUMA node0 CPU(s): 0-9,20-29
  - NUMA node1 CPU(s): 10-19,30-39
- GPU: NVIDIA Quadro RTX 5000

La **metrica** utilizzata per misurare le prestazioni sono i *FLOPS*:

$$FLOPS = \frac{2 \cdot NZ}{T}$$

dove NZ è il numero di nonzeri della matrice e T è la media in secondi del tempo d'esecuzione del kernel
Per ogni kernel **vengono effettuate** 50 esecuzioni.

Implementazione SpMV seriale

- Il prodotto SpMV seriale è stato implementato sia per CSR che HLL.
- L'algoritmo è quello classico per entrambi, non presenta particolarità, per HLL, l'hack size è di 1024
- Parallelizzare gli algoritmi aiuta a migliorarne le prestazioni

Implementazione SpMV CSR con OpenMP

- Per il nucleo di calcolo CSR, ogni iterazione corrisponde a una riga della matrice sparsa.
  - Assunzione di uniformità di carico, anche se in realtà il numero di nonzeri per riga può essere diverso
  - L'effetto è meno evidente rispetto ad HLL: il loop che effettua prodotto scalare è uno solo
  - Usare `schedule(dynamic)` in questo caso non ne giustifica l'overhead
- Utilizzando la direttiva del preprocessore `#pragma omp parallel for schedule(static)` permettiamo al compilatore di gestire automaticamente e in modo trasparente la gestione dei thread.
- Con `schedule(static)`, la runtime di OpenMP preassegna un certo numero di iterazioni del ciclo ai vari thread, il cui numero è impostato con la chiamata `omp_set_num_threads()` in modo equo
  - Overhead dovuto alla runtime omp nullo o quasi: l'assegnazione è fissa e non varia
  - I thread ricevono un certo set di iterazioni da processare, in modo equo, le iterazioni assegnate a ciascun thread sono contigue

Implementazione SpMV HLL con OpenMP

- Per il nucleo HLL, ogni iterazione corrisponde a un blocco ELL
- `#pragma omp parallel for schedule(dynamic)`
  - Il parametro $hs = 1024$
  - I blocchi ELL possono avere un numero di colonne (*maxnz*) molto diverso
  - Alcuni thread potrebbero gestire blocchi ELL più piccoli (o più grandi)
- Utilizzando `schedule(dynamic)`, i thread hanno la possibilità di "pescare" tra il pool di iterazioni ancora da processare e non seguire una preassegnazione statica.
  - Overhead maggiore (risp. sched. static) dovuto alla scelta delle iterazioni ogni volta che il thread termina il lavoro corrrente
  - Bilanciamento del carico (efficiente nel caso di iterazioni che richiedono tempo d'esecuzione differente)

Qualche misura di prestazioni per OpenMP

	1	4	8	16	32	40
Cube_Coup_dt0	1.39	3.79	4.27	5.10	5.62	4.86
ML_Laplace	1.46	3.72	4.25	6.47	7.65	7.62
af_1_k101	1.39	3.69	4.18	4.40	5.46	5.38
nlpkkt80	1.44	3.24	3.81	3.93	4.13	4.03
PR02R	1.43	4.09	4.37	4.61	4.36	4.32

Tabella: GFLOPS per CSR al variare del numero di thread

	1	4	8	16	32	40
Cube_Coup_dt0	1.28	2.45	2.67	2.77	3.18	3.25
ML_Laplace	1.44	3.18	3.69	4.63	3.53	3.12
af_1_k101	1.46	2.98	3.13	3.43	3.73	3.33
nlpkkt80	1.34	2.75	2.62	2.67	2.61	2.61
PR02R	0.85	1.70	1.72	1.86	1.89	3.05

Tabella: GFLOPS per HLL al variare del numero di thread

Implementazione SpMV CSR con CUDA

Sono state realizzate 3 implementazioni

- Una prima versione "CSRv1" basilare: ogni thread effettua moltiplicazione di una riga della matrice per x
  - Accessi non coalescenti verso AS e JA
  - Divergenza dei warp impatta
- Una seconda versione "CSRv2" dove ogni riga corrisponde ad un warp e solo il primo thread di ciascun warp effettua la moltiplicazione di una riga per x
  - Spreco enorme di risorse
  - Essendo un solo thread nel warp a processare, non possono esistere problemi legati alla memoria

Implementazione SpMV CSR con CUDA (CSRv3)

- Una terza variante "CSRv3": a ogni riga corrispnde un warp e tutti i thread del warp partecipano alla moltiplicazione della riga per x
  - Ciascun thread moltiplica 0,1 o più nonzeri che contribuiscono al prodotto scalare
  - Utilizzo della memoria condivisa: ogni thread del warp scrive risultato parziale del prodotto scalare della riga di competenza
  - `__syncthreads()` non necessario
  - Il primo thread di ciascun warp effettua riduzione, scrive quindi il risultato in memoria globale
  - Accessi coalescenti verso AS e JA
  - Two-way bank conflict per la memoria condivisa presente

In tutte le varianti, essendo i nonzero sparsi sulla riga (in colonne diverse), l'accesso al vettore x, non è coalescente (global memory).

Implementazione SpMV HLL con CUDA

Sono state realizzate 2 implementazioni e in entrambi i casi $hs = 32$ - viene effettuata la trasposta delle matrici AS e JA per garantire accesso coalescente, `cudaMallocPitch` e `cudaMemcpy2D` vengono utilizzati per accedere a memoria globale allineata correttamente

- La versione "HLLv1" basilare: ciascun thread moltiplica un blocco ELL per il vettore x
  - Divergenza dei warp impatta
  - Nonostante le matrici siano trasposte, l'accesso non è coalescente (AS e JA) perchè ciascun thread opera su un blocco ELL diverso

Implementazione SpMV HLL con CUDA (HLLv2)

- La versione "HLLv2": il blocco ELL corrisponde al warp e ciascun thread del warp effettua la moltiplicazione della riga del blocco ELL per x
  - Accessi coalescenti (AS e JA) dato che tutti i thread del warp operano sullo stesso blocco $\Rightarrow$ le trasposte hanno l'effetto desiderato
  - Utilizzo della memoria condivisa: i thread del warp vi scrivono il risultato della moltiplicazione della riga per il vettore x
  - `__syncthreads()` non necessario
  - Il primo thread del warp ha la responsabilità di scrivere i risultati nella posizione corretta del vettore y
  - Two-way bank conflict presente

Sia per CSRv3 che HLLv2, il two-way bank conflict può essere facilmente risolto passando a operazioni FP32 (e quindi, la shmem deve ospitare dei `floats`).
Sussiste inoltre il problema dell'accesso non coalescente al vettore x.

Qualche misura di prestazioni per CUDA

	CSRv1	CSRv2	CSRv3
Cube_Coup_dt0	4.16	9.81	14.25
ML_Laplace	3.85	8.50	14.37
af_1_k101	5.24	7.99	7.41
nlpkkt80	5.96	7.70	5.86
PR02R	4.07	7.90	10.40

Tabella: GFLOPS per CSR al variare della versione del kernel

	HLLv1	HLLv2
Cube_Coup_dt0	1.92	45.10
ML_Laplace	2.10	44.88
af_1_k101	2.08	42.44
nlpkkt80	1.99	42.36
PR02R	1.42	31.17

Tabella: GFLOPS per HLL al variare della versione del kernel

Conclusioni

- In termini sommari, HLLv2 GPU è in assoluto la variante che è più performante, mentre la peggiore risulta HLLv1 GPU: questo mostra quanto ottimizzare è importante, dato il distacco notevole
- In generale, CSR è più prestante su CPU che GPU (e viceversa per HLL)
- Le misure di prestazioni esposte sono parziali rispetto a quelle effettuate sulla totalità delle matrici - ad esempio nel confronto tra CSR e HLL nel caso CPU non ci si rende conto dell'importante differenza (se si guardano solo le esecuzioni del kernel su queste 5 matrici) che esiste
- Potrebbe essere una buona idea, specialmente per le GPU, implementare i kernel per operazioni floating point a 32 bit

Grazie per l'attenzione!