Prodotto matrice-vettore sparso (SpMV) con OpenMP e CUDA

Sistemi di Calcolo Parallelo e Applicazioni

Massimo Buniy Lorenzo Grande

Università degli Studi di Roma "Tor Vergata"

Settembre 16, 2025

Roadmap

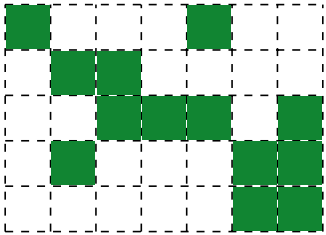
Roadmap

Obiettivi del progetto

- ✓ **Implementare e confrontare** SpMV ($y \leftarrow Ax$) su matrici sparse
- ▷ Analisi prestazioni: **OpenMP (CPU)** vs **CUDA (GPU)**
- ∗ Valutare l'impatto dei **formati dati**: CSR e HLL

Perché lo SpMV è importante

- Operazione fondamentale in calcolo scientifico e ML.
- Le matrici reali sono spesso **sparse**.
- Prestazioni dominate da accessi memoria, non dai FLOPs.

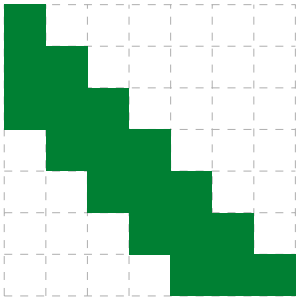


Esempio di matrice sparsa: pochi non-zeri rispetto alle celle totali

Matrici sparse: categorie

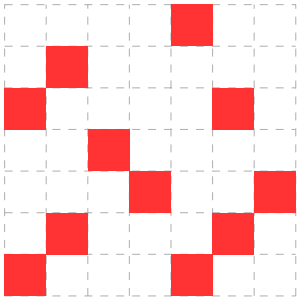
- **Strutturate**: tridiagonali, a banda, ecc. (pattern regolari).
- **Non strutturate**: distribuzione irregolare dei non-zeri.

Strutturata



Pattern regolare

Non strutturata



Pattern irregolare

Scope e scelte progettuali

- **Uniformità**: tutte le matrici trattate come non strutturate.
  - **Pro**: Un solo algoritmo per tutti i casi.
  - **Contro**: Maggior uso di memoria rispetto a formati dedicati.
- **Ambienti di esecuzione**:
  - CPU multi-core (`OpenMP`)
  - GPU (`CUDA`)
- **Metriche**: tempi medi su 10 run (esclusi I/O e check di correttezza).

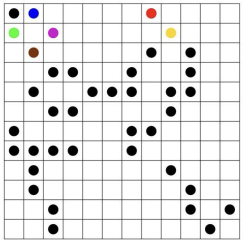
Importanza dei formati di memorizzazione

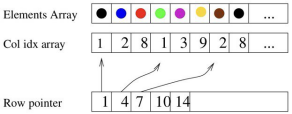
L'algoritmo è importante, ma l'efficienza dipende anche dal formato di memorizzazione.

- Una rappresentazione adatta riduce **spazio occupato** e **accessi in memoria**.
- A differenza delle matrici dense (zeri memorizzati esplicitamente), le sparse salvano solo i **non-zeri** e le loro posizioni.
- Nel nostro progetto abbiamo adottato due formati: **CSR** e **HLL**.

Formato CSR (Compressed Sparse Row)

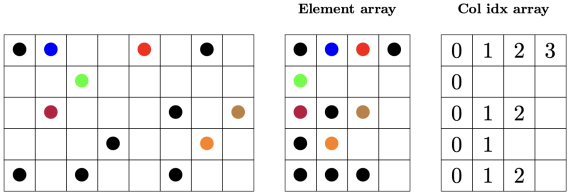
- Tre vettori: `IRP` (puntatore inizio riga), `JA` (vettore indici colonne), `AS` (vettore dei valori).
- **Pro**: accesso riga-per-riga ordinato $\rightarrow$ buona località su CPU.
- **Contro**: meno adatto a slicing per colonne.





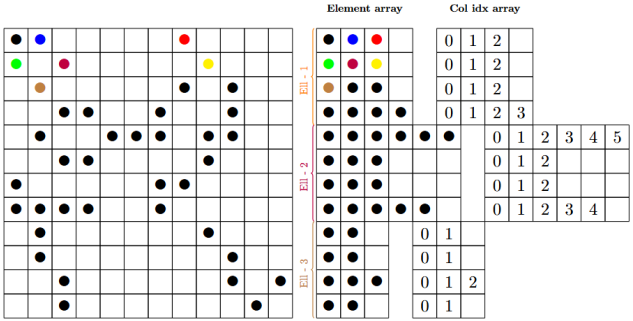
Formato ELLPACK (richiamo)

- Ogni riga ha `MAXNZ` elementi (padding se una riga contiene meno non zeri).
- Strutture 2D: `JA[M][MAXNZ]`, `AS[M][MAXNZ]`.
- **Pro**: schema regolare, ideale su GPU.
- **Contro**: forte spreco se le righe hanno un numero di non-zeri molto diverso.



Formato HLL (Hybrid Linear List)

- Divide la matrice in blocchi da `HackSize` righe.
- Ogni blocco è memorizzato in **ELLPACK** $\rightarrow$ accessi regolari.
- **Pro**: riduce lo spreco di memoria rispetto a ELLPACK puro.
- **Contro**: resta del padding nei blocchi e si introduce overhead per gestire più strutture.



Impatto della rappresentazione dati

- **CSR** è in genere più **compatto** e **cache-friendly** su CPU.
- **HLL** migliora la regolarità rispetto a sparse generico, ma soffre padding/indirezioni.
- **HLL Aligned** (variante) migliora coalescenza su GPU.

Roadmap

FLOPS

Misurano la velocità di calcolo, cioè quante operazioni in virgola mobile vengono eseguite al secondo.

Come si misura:

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

dove NZ è il numero di elementi non nulli e T il tempo medio di esecuzione.

SpeedUp

- Indica il **guadagno rispetto alla versione sequenziale**.
- SpeedUp $= \dfrac{T_s}{T_p}$
- Dove T_s = tempo sequenziale e T_p = tempo parallelo.

Note di misurazione

- Tempi = **media su 10 run** per ridurre la varianza.
- Esclusi da tutte le misure: caricamento dati e controlli di correttezza.
- Per CUDA: distinto il tempo del **kernel** dai trasferimenti H2D/D2H.

Roadmap

OpenMP: introduzione

- **OpenMP (Open Multi-Processing)** è un'interfaccia di programmazione basata su direttive per il calcolo parallelo su architetture a memoria condivisa.
- Supportata da C, C++ e Fortran, è apprezzata per la **semplicità di integrazione** in codice seriale esistente.
- Nel progetto: implementata in `C` per parallelizzare il prodotto matrice-vettore.
- Utilizzo di t thread per distribuire equamente il carico, ridurre i tempi di esecuzione e migliorare l'efficienza.

OpenMP: strategie adottate

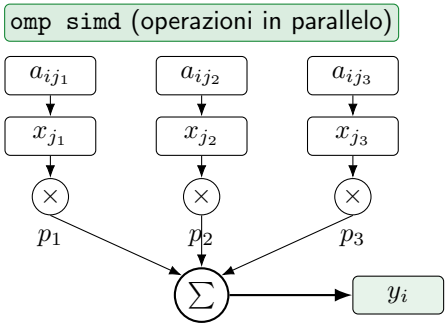
- Divisione del lavoro:
  - manuale con `ThreadDataRange`
  - automatica con `#pragma omp for`
- Ottimizzazione vettoriale con `omp simd`.
- Scheduling: **statico**, **dinamico** e **auto**.

OpenMP su CSR: Soluzioni 1–2

- **Sol1 – Divisione manuale** Le righe vengono suddivise esplicitamente tra i thread.
  - **Vantaggio**: utile per comprendere la parallelizzazione a basso livello.
  - **Limite**: nessuna ottimizzazione nell'associazione riga–thread.
- **Sol2 –** `omp parallel for` Parallelizza automaticamente il ciclo sulle righe.
  - **Vantaggio**: gestione automatica ed efficiente della distribuzione, senza calcolare esplicitamente gli intervalli.

OpenMP su CSR: Soluzione 3

- `omp parallel for` + `omp simd` con riduzione.
  - **Vantaggio**: esegue più moltiplicazioni in parallelo con un'unica istruzione.
  - **Risultato**: in media la più efficiente tra le prime tre varianti testate.



`reduction(+:)` somma i parziali in modo sicuro

Prodotti in parallelo (`simd`) → somma finale (`reduction`) → y_i.

OpenMP su CSR: Soluzione 4

- Si aggiunge la clausola `schedule(auto)` a `omp parallel for`.
- Delega al compilatore la distribuzione delle iterazioni, potenzialmente adattandola alla macchina e al carico.

OpenMP su CSR: Soluzione 5

- Utilizzo della struttura **ThreadDataRange**: Divide esplicitamente il lavoro tra i thread specificando un intervallo di righe per ciascuno.
  - **Vantaggio**: consente un bilanciamento più fine, utile con matrici sparse irregolari.
  - **Limite**: maggiore complessità implementativa; non sempre migliore di Sol3.

OpenMP su HLL: approccio a blocchi

- La matrice HLL è suddivisa in blocchi indipendenti, ciascuno in formato **ELLPACK**.
- Parallelizzazione a livello di blocco: ogni thread elabora uno o più blocchi in modo indipendente.
- Non introdotta parallelizzazione interna al blocco:
  - **Motivazione**: i blocchi sono piccoli $\rightarrow$ la doppia parallelizzazione genera overhead.
  - **Risultato**: le misure hanno mostrato un peggioramento dei tempi se si forza la doppia parallelizzazione.
- Focus progettuale: assegnazione dei blocchi ai thread e gestione errori con flag globale.

OpenMP su HLL: Soluzioni 1–2

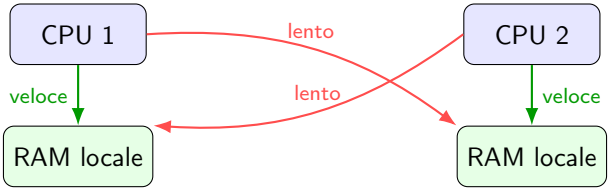
- **Sol1 – Assegnazione semplice**: Ogni thread elabora uno o più blocchi HLL.
  - **Vantaggio**: sfrutta la naturale suddivisione in blocchi; codice semplice.
- **Sol2 –** `schedule(dynamic)`: Parallelizzazione dei blocchi con schedulazione dinamica.
  - **Vantaggio**: distribuzione flessibile del carico, utile se i blocchi hanno dimensioni diverse.

OpenMP su HLL: Soluzione 3

- Utilizzo della strutuuttrua `ThreadDataRange`: Intervalli di blocchi assegnati manualmente a ciascun thread.
  - **Vantaggio**: Massimo controllo sul bilanciamento del carico, utile se i blocchi non sono distribuiti in modo uniforme.
  - **Limite**: Maggiore complessità implementativa.

OpenMP: osservazioni prestazionali

- **Scalabilità**: quasi lineare fino a ~ 20 thread (core fisici).
- **Hyperthreading** (21–40): guadagni ridotti e maggiore variabilità.
- **NUMA**: accessi remoti alla memoria possono introdurre latenze.
- **Confronto formati**: CSR in media più efficiente di HLL.



Accessi locali = veloci; remoti = più lenti → effetti del NUMA.

OpenMP — CSR: takeaway & soluzioni

- **Formato**: CSR = dati compatti e accessi contigui $\Rightarrow$ prestazioni migliori in media.
- **sol3**: `parallel for` + `omp simd` + riduzione = *best overall*.
- **sol4** (`schedule(auto)`): dipende dal bilanciamento effettivo ottenuto dal runtime.
- **sol5** (partizione manuale con `ThreadDataRange`): consente un controllo fine sulla distribuzione del lavoro e può risultare vantaggiosa in presenza di squilibri noti, ma introduce overhead e risulta meno flessibile nei casi generici.

OpenMP — HLL/HLL Aligned: takeaway & soluzioni

- **Formato HLL**: accessi meno regolari e padding $\Rightarrow$ *GFLOPS* inferiori rispetto a CSR.
- **HLL Aligned**: migliore regolarità/allineamento $\Rightarrow$ incremento sensibile dei *GFLOPS*.
- **sol1**: semplice ma rischio di sbilanciamento;
- **sol2** (`schedule(dynamic)`) bilancia meglio con costo di pianificazione;
- **sol3** (partizione manuale) ottima se carico uniforme.

Roadmap

CUDA: modello di esecuzione

- **CUDA (Compute Unified Device Architecture)**: framework NVIDIA per il calcolo parallelo su GPU.
- Organizzazione gerarchica: griglia $\rightarrow$ blocchi $\rightarrow$ thread.
  - Un kernel viene eseguito da migliaia di thread in parallelo.
  - I thread sono raggruppati in blocchi, a loro volta organizzati in griglie.
  - Un **warp** = 32 thread che eseguono la stessa istruzione su dati diversi.
- Blocchi e griglie possono essere 1D, 2D o 3D $\Rightarrow$ flessibilità nell'adattare la computazione alla struttura dei dati.

CUDA: inizializzazione e memoria

- **Configurazione esecuzione**: scelta della dimensione dei blocchi e della griglia.
  - Dimensione blocco = multiplo di 32 (warp) e nei limiti hardware (max 1024 thread nel nostro caso).
  - Numero di blocchi $\geq$ numero di SM (Streaming Multiprocessor) per massimizzare l'occupancy.
- **Memoria**:
  - **Global memory**: accessibile da tutti i thread, ma lenta.
  - **Shared memory**: veloce, condivisa solo all'interno di un blocco; richiede sincronizzazione.
  - Accessi coalescenti $\Rightarrow$ fondamentale che i thread di un warp leggano da locazioni contigue.
- Nel progetto: trasferimento strutture (CSR/HLL) e vettori su device; uso di `cudaMemAdvise` per hint di sola lettura.

CUDA su CSR: Soluzione 1 (thread-per-row)

- Ogni thread elabora **una riga** della matrice in formato CSR, calcolando la somma dei prodotti non-zeri per gli elementi del vettore.
- **Vantaggio**: approccio semplice ed efficiente per matrici sparse di grandi dimensioni con carichi di lavoro uniformi tra le righe.
- **Limite**: possibili squilibri se il numero di non-zeri varia molto da riga a riga.

CUDA su CSR: Soluzione 2 (warp-per-row)

- Ogni **warp** (32 thread) lavora insieme sulla stessa riga CSR: i thread calcolano prodotti parziali che vengono poi sommati con una riduzione interna al warp.
- **Vantaggio**: ideale per righe lunghe o dense, perché sfrutta tutti i 32 thread e riduce il numero di accessi/operazioni globali.
- **Limite**: poco efficiente per righe corte, dove parte del warp resta inattivo.

CUDA su HLL: Soluzione 1 (block-per-HLL-block)

- **Schema**:
  - 1 blocco CUDA $\rightarrow$ 1 blocco **ELLPACK** di HLL;
  - I thread del blocco CUDA $\rightarrow$ *righe* del blocco ELLPACK.
- `#pragma unroll` sul loop dei non-zeri per ridurre il controllo per elemento.
- **Vantaggi**: mappatura diretta e semplice; buona efficienza per blocchi compatti.
- **Limite**: dipende dall'uniformità del blocco (MAXNZ) e dalla regolarità delle righe.

CUDA su HLL: Soluzione 2 (coalesced writes)

- Le **scritture** del risultato di riga sono rese **coalescenti** in global memory.
- I thread del blocco scrivono su *indirizzi contigui* $\Rightarrow$ meno transazioni e latenza ridotta.
- **Vantaggi**: migliore utilizzo della banda di memoria rispetto a scritture sparse/disallineate.

Coalescenza: i thread di un warp accedono a celle di memoria contigue, così la GPU esegue un'unica transazione invece di tante separate.

CUDA: tuning della blockSize

- La scelta della **blockSize** influenza direttamente l'**occupancy** della GPU e l'uso delle risorse (registri, shared memory).
- Valori testati: $\{32, 64, 96, 128, 160, 192, 256, 320, 384, 512, 768, 1024\}$.
- Se troppo grande $\Rightarrow$ saturazione risorse per blocco $\Rightarrow$ calo di occupancy.
- **Intervallo ottimale**: ~ 192–~ 320 thread per blocco, buon compromesso tra stabilità delle prestazioni e massima occupancy.

CUDA — CSR: soluzioni a confronto

- **sol1** (un thread per riga): efficace con righe di lunghezza simile; soffre su righe sbilanciate.
- **sol2** (un warp per riga + riduzione): ottima con righe dense; sprechi quando le righe sono corte.
- Non esiste un "vincitore universale": dipende dalla distribuzione dei non-zero per riga (es. differenze tra *amazon0302* e *cant*).

CUDA — HLL e HLL Aligned

- Per HLL le due soluzioni mostrano **andamenti molto simili** al variare della `blockSize`.
- **HLL Aligned** $\Rightarrow$ accessi più coalescenti in globale, latenza minore, throughput maggiore: *GFLOPS* spesso superiori a HLL base.
- Intervallo consigliato: ~ 192–~ 320 thread/blocco per massimizzare occupancy senza penalità di risorse.

Roadmap

Dataset e classificazione

- Suite Sparse Matrix Collection (TAMU): matrici reali e variegate.

Classe	**#NZ**
Small	$\leq 10^4$
Medium	$10^4 - 10^6$
Large	$10^6 - 8 \cdot 10^6$
X-Large	$> 8 \cdot 10^6$

Esempi: *bcspwr01* (small), *amazon0302* (medium), *cant* (large), *bone010* (xlarge).

Ambiente di test (HW/SW)

- **CPU**: 2x Intel Xeon Silver 4210 @ 2.20GHz (20 core fisici, 40 thread logici).
- **Memoria**: 128 GB RAM; **NUMA**: 2 nodi.
- **GPU**: NVIDIA Quadro RTX 5000 (48 SM, max 1024 thread per blocco).
- Sviluppo: CLion locale; esecuzione remota via SSH; compilatori g++/`nvcc`.

Configurazioni di esecuzione

- **OpenMP**: test 1→40 thread (nessun pinning NUMA).
- **CUDA**: variazione `blockSize` come da lista.
- Misure: media su 10 run; tempi kernel separati da trasferimenti.

Roadmap

OpenMP vs CUDA — takeaway

- **Formato dati** è il driver n. 1 delle prestazioni (CSR $>$ HLL; HLL Aligned $>$ HLL).
- **OpenMP**: scaling quasi lineare su core fisici; nessuna singola strategia dominante (sol3 spesso migliore su CSR).
- **CUDA**: sensibilità alla struttura delle righe; scelta `blockSize` cruciale (~ 192–320); soluzioni diverse vincono a seconda della matrice.
- **Messaggio chiave**: algoritmo $\times$ formato $\times$ architettura $\Rightarrow$ sinergia necessaria per massimizzare lo *SpMV*.

Roadmap

Conclusioni e takeaway

- **Formato dati è determinante**: CSR in media più efficiente; HLL penalizzato; HLL Aligned riduce il gap su GPU.
- **OpenMP**: ottima scalabilità fino ai core fisici, ma *hyperthreading* e NUMA introducono limiti.
- **CUDA**: prestazioni fortemente dipendenti dalla distribuzione dei non-zeri per riga e dalla scelta della `blockSize`.
- **No silver bullet**: occorre scegliere *formato*, *algoritmo* e *tuning* in funzione dell'hardware e dei dati.

Future work: esplorare strategie di auto-tuning, estensioni multi-GPU, integrazione CPU–GPU ibrida.