Matrici Sparse: SparseArrays vs CuArrays vs GraphBLAS

Valerio Caravani, Marta Paolucci

23/02/2020

Indice

1 Introduzione

Il corso di Calcolo Parallelo e Distribuito ha avuto come caso di studio principale la libreria Linear Algebraic Representation (LAR) sviluppata dal professor Alberto Paoluzzi. LAR, in poche parole, punta ad ottenere una caratterizzazione minima della geometria e della topologia di un complesso cellulare ricostruendo la catena principale (anche detta pipeline 3D) di generazione dei vertici, spigoli, facce e celle di un qualsiasi solido. La rappresentazione scelta per descrivere la matematica in LAR è un insieme di matrici sparse di tipo intero; infatti le matrici sparse sono la principale struttura dati utilizzata nella libreria LAR. Le matrici sparse vengono molto spesso sfruttate nel calcolo scientifico soprattutto per la loro proprietà di memorizzare solo gli elementi non nulli, presenti nella matrice, in modo da risparmiare sia in termini di tempo (nella computazione) e sia in termini di spazio (nella memorizzazione).

Durante il corso si è posta l'attenzione sulla valutazione e sul miglioramento delle performance della libreria LAR, sviluppata nel linguaggio Julia.

1.1 Packages

Il cuore delle computazioni presenti in LAR sono proprio le operazioni tra matrici sparse, quindi si é ritenuto utile concentrare gli sforzi di ricerca sulla scelta di un package che garantisse il miglior bilanciamento tra performance e usabilità. Durante lo studio e l'analisi si è svolta una comparazione tra tre diversi package Julia che presentano supporto alle operazioni tra matrici sparse, i packages a cui ci riferiamo sono rispettivamente SparseArrays, CuArrays e SuiteSparseGraphBLAS. I tre package fanno leva su principi differenti e sono di seguito brevemente descritti.

1.1.1 SparseArrays

Julia supporta l'utilizza di vettori e matrici sparse nella libreria standard SparseArrays.jl, essa rappresenta la modalità "nativa" per memorizzare vettori e matrici sparse. Nello specifico Julia memorizza i vettori e le matrici nel formato CSC (Compressed Sparse Column) quindi il sistema principale di memorizzazione è per indice di colonna e non di riga, portando così ad una ottimizzazione del sistema stesso. Le matrici sparse in julia sono del tipo SparseMatrixCSC(Tv,Ti), dove Tv è il tipo dei valori memorizzati e Ti è il tipo (intero) usato per memorizzare indici di riga e puntatori alle colonne.Poniamo l'attenzione però sul fatto che le computazione operate da SparseArrays avvengono esclusivamente su CPU.

SparseArrays è attualmente il package principale usato da LAR.

1.1.2 CuArrays

Il package CuArrays offre un'astrazione di programmazione per GPU-array elegante ed estramemente generica. L'utilizzo di CuArray può fornire significativi aumenti di velocità rispetto agli array normali senza modifiche al codice. L'unicità di questo package risiede nell' alto livello di astrazione proposto, in sintesi l'utlizzo della libreria non richiede un esperienza pregressa nel campo della programmazione per GPU.

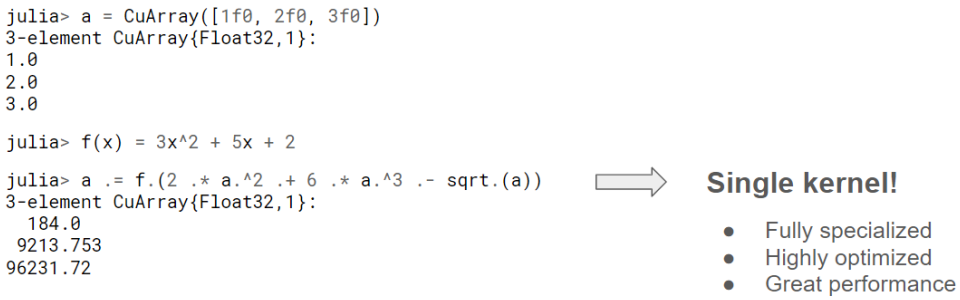

Figura 1: Esempio di operazione con CuArrays

1.1.3 CuArrays.CUSPARSE

Parlando invece del package CuArrays, questa presenta diverse sottolibrerie, come ad esempio: CUBLAS, CUDNN, CUSPARSE, ecc. La libreria CuArrays.CUSPARSE ci viene in aiuto, infatti quest'ultima permette la creazione di matrici CUDA sparse (presenti sia in formato CSC e CSR) e fornisce un set di operazioni BLAS. Tuttavia la libreria è ancora in fase di sviluppo e ad oggi non supporta tutte le operazioni necessarie.

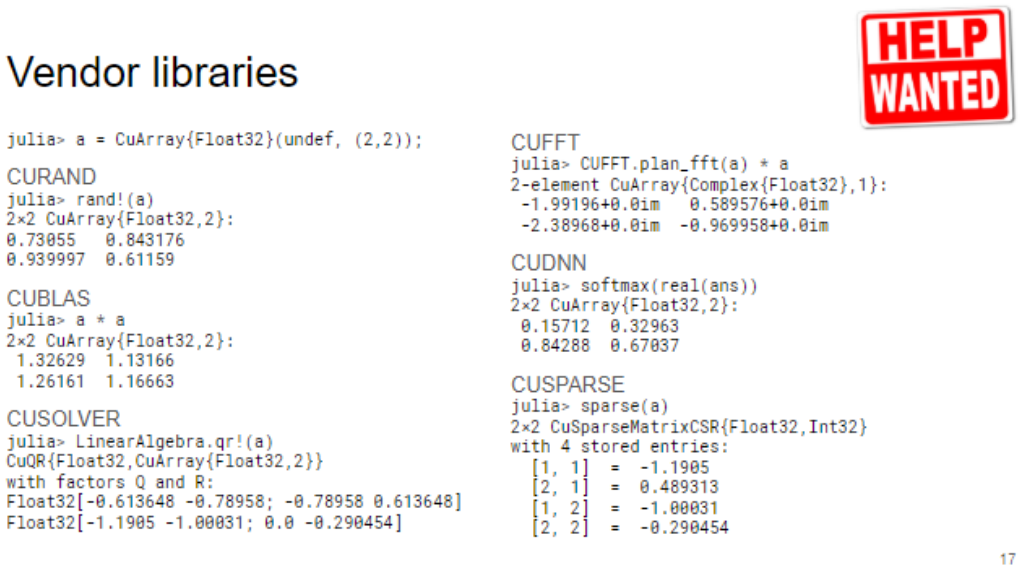

Figura 2: Immagine tratta da una presentazione di Tim Besard, da notare la richiesta di aiuto alla community

1.1.4 GraphBLAS

GraphBLAS è un progetto che si propone di definire uno standard per le operazioni su grafi nel linguaggio dell'algebra lineare. Nello specifico lo standard definisce un insieme di operazioni su matrici sparse basate sull' algebra estesa dei semianelli, utilizzando una varietà quasi illimitata di operatori e tipi. Quando queste operazioni algebriche vengono applicate a matrici di adiacenza sparse si vanno ad equiparare a calcoli su grafi. Graph-

BLAS quindi fornisce un framework potente ed espressivo per la creazione di algoritmi su grafi basati sull'elegante matematica dei semianelli.

SuiteSparseGraphBLAS.jl rappresenta il wrapper Julia della libreria originaria sviluppata in C.

1.2 Obiettivi del progetto

Il progetto ha previsto uno studio di comparazione dei tre package presentati precedentemente. Nello specifico i package sono stati comparati dal punto di vista della moltiplicazione tra matrici sparse e divisioni di matrici sparse per uno scalare.

I tre package sono stati testati partendo da un codice di esempio isolato dall'intera libreria: il file casaletto.jl. In casaletto.jl viene presentata la pipeline di generazione degli operatori di cobordo. Da qui è stato possibile isolare le operazioni di moltiplicazione tra matrici sparse ed effettuare i confronti.

Di seguito vengono riporati i tre obiettivi principali perseguiti durante la ricerca:

1. Implementazione della pipeline presente in casaletto.jl con i package:
    - CuArrays
    - CuArrays.CUSPARSE
    - SuiteSparseGraphBLAS
2. Benchmark delle diverse implementazioni in termini di tempo;
3. Confronto in termini di memoria tra SparseArrays e SuiteSparseGraphBLAS;

2 Casaletto.jl

Come già detto, casaletto.jl rappresenta il codice di test del progetto, infatti nel file viene eseguita la pipeline di generazione degli operatori di cobordo necessari alla rappresentazione del casaletto stesso. A partire da questo file è stato possibile estrapolare le operazioni di moltiplicazione tra matrici sparse ed effetuare le comparazioni tra i vari package, utilizzando macro specifiche per generare gli output desiderati.

Viene dapprima mostrata e descritta l'implementazione originaria del file casaletto.jl tramite le matrici sparse di SparseArrays.jl; in seguito vengono presentate le diverse implementazioni e i risultati relativi ottenuti tramite l'utilizzo dei package CuArrays, CuArrays.CUSPARSE e GraphBLAS.

Si noti la funzione K, quest'ultima ritorna le matrici sparse utilizzate per la generazione degli operatori di cobordo:

SparseArrays.jl

SparseArrays.sparse

```
function K( CV )
        I = vcat( [ [k for h in CV[k]] \
            for k=1:length(CV) ]...)
        J = vcat(CV...)
        Vals = [1 for k=1:length(I)]
        return SparseArrays.sparse(I,J,Vals)
end
```

Di seguito si riporta il codice (semplificato) presente nel file Casaletto.jl..

```
------------------------------------------------------------------

using LinearAlgebraicRepresentation, SparseArrays
Lar = LinearAlgebraicRepresentation

V,CV = ... #  geometria e topologia del casaletto


M_0 = K(VV)
M_1 = K(EV)
M_2 = K(FV)
M_3 = K(CV)

s_1 = M_0 * M_1'
s_2 = (M_1 * M_2') .\div 2
s_3 = (M_2 * M_3')
s_3 = s_3 ./ 4
s_3 = s_3 .\div 1

S2 = sum(s_3,dims=2)
inner = [k for k=1:length(S2) if S2[k]==2]
outer = setdiff(collect(1:length(FV)), inner)

# Visualizzazione con ViewerGL

------------------------------------------------------------------
```

2.1 Casaletto-cusparse.jl

L'implementazione di casaletto tramite le funzioni offerte in CuArrays.CUSPARSE ha rappresentato il primo passo del progetto. Purtroppo si è arrivati soltanto ad una im-

plementazione parziale della pipeline (la moltiplicazione è su gpu, le altre operazioni di divisione intera non sono su cpu). Non è stato possibile operare le intere operazioni su GPU tramite CuArrays.CUSPARSE in quanto la libreria risulta incompleta anche nella operazioni piu semplici. Ad esempio non è possibile stampare a schermo le matrici o accedere ai suoi elementi in quanto la funzione get-index non risulta implementata per il tipo CuSparseMatrix(CSC o CSR).

CuSPARSE

CuArrays.CUSPARSE.CuSparseMatrixCSR

```
function K_cusparse( CV )

        I = vcat( [ [k for h in CV[k]] \
                for k=1:length(CV) ]...)
        J = vcat(CV...)
        Vals = Float32[1 for k=1:length(I)]
        M = sparse(I,J,Vals)

        d_M = CuSparseMatrixCSR(M)

        return d_M
end
```

Il codice qui riportato mostra l'utilizzo dell'operazione CUSPARSE.gemm per eseguire la moltiplicazioe tra matrici sparse in Cuda. La funzione permette, tramite i suoi parametri di trasporre le matrici in input per permettere la moltiplicazione. Qui le matrici interessate nella moltiplicazione devono essere esclusivamente delle matrici di tipo CSR (utilizzando il tipo CSC si va in errore, non sappiamo perchè), il collect che permette di "riportare" le matrici su cpu ottiene comunque delle matrici CSC, essendo l' unico tipo di matrici sparse implementato in Julia.

```
-----------------------------------------------------------------

using CuArrays.CUSPARSE


d_M0 = K_cusparse(VV)
d_M1 = K_cusparse(EV)
d_M2 = K_cusparse(FV)
d_M3 = K_cusparse(CV)

d_s1 = CUSPARSE.gemm('N','T',M_0_cusparse, \
                          M_1_cusparse,'O','O','O');
d_s2 = CUSPARSE.gemm('N','T',M_1_cusparse,\
```

```
                          M_2_cusparse,'O','O','O');
d_s3 = CUSPARSE.gemm('N','T',M_2_cusparse, \
                          M_3_cusparse,'O','O','O');


s_2 = collect((d_s2)) .   2;
s_3 = (collect(d_s3) ./ 4) .   1;
```

2.2 Casaletto-cuarrays.jl

L'impossibilità di tradurre l'intera pipeline su GPU tramite CuArrays.CUSPARSE ha portato il gruppo a tentare una via diversa: abbandonare le matrici sparse Cuda in favore di matrici Cuda dense (CuArray bidimensionali) in maniera tale da poter computare l'intera pipeline su device.

L'estrema flessibilità della package CuArrays ha portato a non dover cambiare alcuna riga di codice rispetto a Casaletto.jl ad esclusione di una conversione in CuArrays, tramite la funzione CuArrays.cu applicata nella funzione K-cu. Ovviamente il passaggio a matrici dense comporta a vari limiti nell'occupazione di memoria in fase di computazione (vedi sezione Benchmark).

CuArrays

CuArrays.cu

```
function K_cu( CV )
        I = vcat( [ [k for h in CV[k]] \
                    for k=1:length(CV) ]...)
        J = vcat(CV...)
        Vals = [1 for k=1:length(I)]
        return cu(sparse(I,J,Vals))
end
```

```
using CuArrays


d_M0 = K_cu(VV)
d_M1 = K_cu(EV)
d_M2 = K_cu(FV)
d_M3 = K_cu(CV)
```

```
d _ _1  = d_M0 * d_M1
d _ _2  = (d_M1 * d_M2') .   2
    # fully specialized kernel!!!
d _ _3  = (d_M2 * d_M3') ./ 4 ) .   1
    # fully specialized kernel!!!
```

2.3 Casaletto-graphblas.jl

L'implementazione GraphBLAS ha permesso al gruppo di toccare con mano una liberia versatile ed estremamente potente come SuiteSparseGraphBLAS. Superate le difficoltà iniziali dovuta alla macchinosità della sintassi e delle operazioni, l' implementazione è risultata tutto sommato di semplice stesura. L'implementazione GraphBLAS ha permesso al gruppo di comparare il lavoro svolto per i package dedicati al calcolo su GPU con un ulteriore package che gira su CPU oltre SparseArrays.

Risulta però necessario introdurre alcune funzioni per comprendere meglio il codice finale. In primis viene mostrata la funzione di generazione delle matrici sparse, come fatto precedentemente per gli altri package.

GraphBLAS Sparse Matrix

SuiteSparseGraphBLAS.GrB-Matrix

```
function K_GBLAS( CV )
        I = (vcat( [[k for h in CV[k]] \
                for k=1:length(CV) ]...));
        J = vcat(CV...);
        X = [1 for k=1:length(I)];

        M = GrB_Matrix(OneBasedIndex.(I),\
                        OneBasedIndex.(J), X);
        return M
end
```

Di seguito vengono mostrate e descritte le altre funzioni utilizzate nella generazione degli operatori di cobordo.

La seguente funzione mostra la generazione di un operatore unario, utilizzato da GraphBLAS per eseguire la divisione intera su tutti gli elementi di una matrice sparsa.

GraphBLAS Unary Operator

```
float_division_by_1 = a -> a / 1

function floatdivbyn_sigma(sigma,f_div)

        FLOATDIV_BYN = GrB_UnaryOp()
        GrB_UnaryOp_new(FLOATDIV_BYN, f_div, \
                        GrB_FP64, GrB_FP64)

        sigma_divbyn = GrB_Matrix{Float64}()
        GrB_Matrix_new(sigma_divbyn, GrB_FP64, \
                       GrB_Matrix_nrows(sigma),\
                       GrB_Matrix_ncols(sigma))

        GrB_Matrix_apply(sigma_divbyn, GrB_NULL, \
                         GrB_NULL, FLOATDIV_BYN, \
                         sigma, GrB_NULL)

        return sigma_divbyn

end
```

Passando come parametri float-division-by-1 e la matrice sparsa d'interesse S alla funzione floatdivbyn-sigma si ottiene un opeatore unario per divisione non intera di tutti gli elementi di S di nome FLOATDIV-BYN. La funzione GrB-Matrix-apply applica l'operatore unario FLOATDIV-BYN ad S.

Nel codice casaletto-graphblas.jl si fa uso di un'altro operatore unario, definito per la divisione intera. Non viene riportato in sede di relazione in quanto del tutto uguale alla generazione dell'operatore FLOATDIV-BY, a meno del tipo. Ciò suggerisce un eventuale futura modifica, l'aggiunta di un parametro tipo, in modo da rendere la generazione degli operatori parametrica con il tipo per far sì che l'operazione sia più flessibile.

```
-----------------------------------------------------------------

using SuiteSparseGraphBLAS, GraphBLASInterface
GrB_init(GrB_NONBLOCKING)

# init GrB_Descriptor for mxm
desc = init_descriptor_mxm(); # get descriptor: (transpose m2, replace

# compute s_1
s_1 = init_s(M_0,M_1);
```

```
GrB_mxm(s_1, GrB_NULL, GrB_NULL, \
GxB_PLUS_TIMES_INT64, M_0, M_1, desc)

# compute   _2
s_2 = init_s(M_1,M_2);
GrB_mxm(s_2, GrB_NULL, GrB_NULL, \
GxB_PLUS_TIMES_INT64, M_1, M_2, desc)
s_2 = intdivbyn_sigma(s_2,integer_division_by_2);

# compute s_3
s_3 = init_s(M_2,M_3);
GrB_mxm(s_3, GrB_NULL, GrB_NULL, \
GxB_PLUS_TIMES_INT64, M_2, M_3, desc)
s_3 = floatdivbyn_sigma(s_3,float_division_by_4)
s_3 = intdivbyn_sigma(s_3, integer_division_by_1)

------------------------------------------------------------
```

3 Benchmark: mxm

Dopo aver implementato casaletto.jl con le diverse "formule" offerte dai package, le quattro implementazioni sono state oggetto di benchmark. Inizialmente sono state esaminate soltanto le operazioni di moltiplicazioni tra matrici sparse, le successive operazioni sono state incluse invece nel benchmark riguardante casaletto.jl presente nella sezione successiva.

Per valutare i tempi di esecuzione è stato fatto uso della macro @btime del package BenchmarkTools.jl. Per poter memorizzare in una variabile i tempi di esecuzione è bastato convertire la macro @btime in @belapsed. Per valutare la quantità in byte di memoria allocata è stata utilizzata la macro @allocated.

Per i package CuArrays e CuSParse la comparazione con SparseArrays è stata fatta esclusivamente in termini di tempo, mentre per GraphBLAS è stato fatto un confronto con SparseArrays in termini di tempo e spazio.

I grafici di seguito riportati descrivono lo speedup della computazione all'aumentare del numero di vertici della geometria descritta tramite LAR. Si è fatto uso di una scala logaritmica per favorire la leggibilità del grafico, orientativamente i valori sull'asse delle ascisse rappresentano, crescendo, un numero successivo, di un ordine di grandezza, di vertici in input.

Per generare l'input è stata usata la funzione Lar.cuboidGrid che genera una griglia di cubi proporzionata all'input immesso. La funzione di benchmark semplicemente calcola e ritorna i tempi nell'esecuzione delle moltiplicazioni tra matrici (sparse e non) nelle varie implementazioni dei vari package usati. Per maggiori dettagli di implementazione si rimanda al link della repo github, presente nella sezione Conclusioni.

Per ogni diverso ordine di grandezza in input è stato misurata la somma complessiva dei tempi dovuti alla generazione dei tre operatori, questo tempo totale ottenuto con un package e con un certo input è stato confrontato con il tempo ottenuto usando Sparse Arrays come riferimento.

Per l'esecuzione dei benchmark è stata utilizzato il superserver NVIDIA DGX-1 del Dipartimento di Matematica e Fisica.

3.1 SparseArrays vs CuArrays

Partendo dal benchmark eseguito per confrontare SparseArray e CuArrays si può notare un effettivo aumento delle prestazioni in corrispondenza di valori in input fino a 10^4 vertici, con input maggiori si nota una drastica riduzione dello speedup dovuta alla saturazione delle risorse ed infine si giunge ad un errore dovuto alla superamento della capacità di una singola tesla p100.

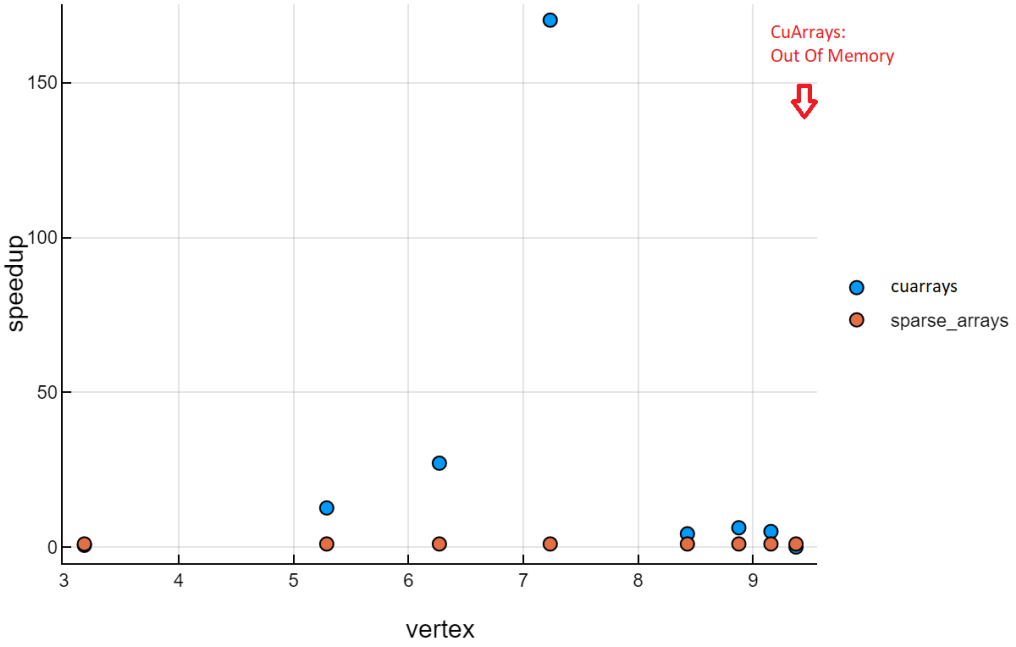

Figura 3: sparse-arrays vs cuarrays

3.2 SparseArrays vs CuArrays.CUSPARSE

Questo grafico mostra l'effettivo e costante miglioramento delle prestazioni nell'eseguire le tre operazioni tramite le matrici sparse cuda. Il miglioramento consistente si ottiene in corrispondenza degli input maggiori. Questo risultato, purtroppo non troverà riscontro nei benchmark su casaletto.jl, dove non verrano analizzate soltanto le moltiplicazioni, ma anche le successive divisioni, che il gruppo non è riuscito ad implementare in pure cuda.

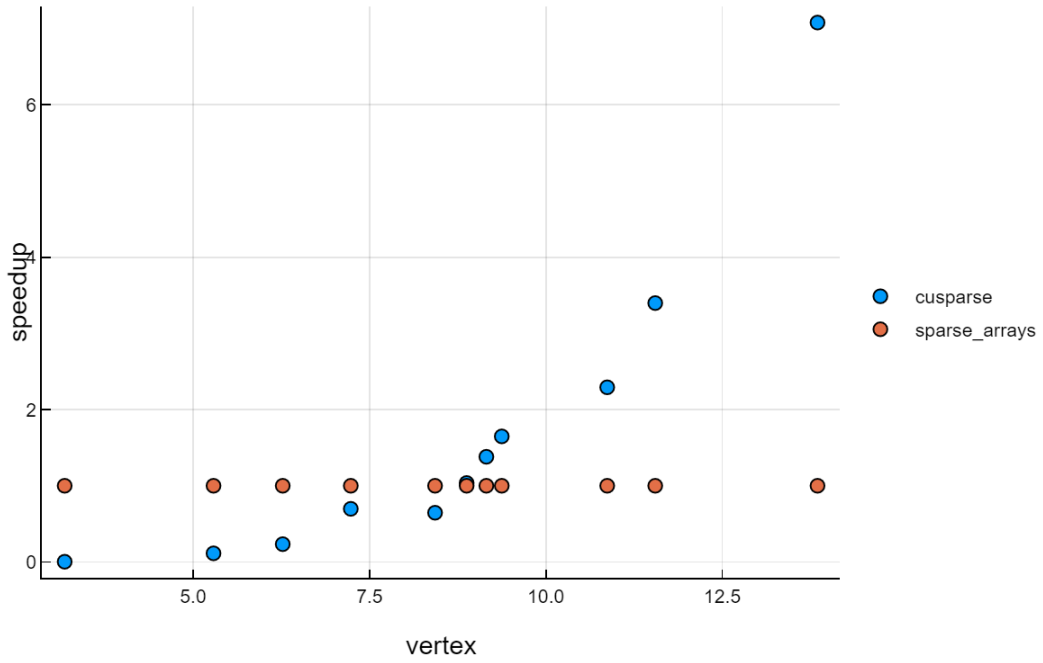

Figura 4: sparse-arrays vs cuarrays

3.3 SparseArrays vs GraphBLAS

Per quanto riguarda GraphBLAS il miglioramento significativo delle prestazioni si ha in termini di memoria. Lo speedup in temrini di tempo è in generale 1.5X. Ci si aspettava uno speedup più significativo, resta il sospetto che un utilizzo più consapevole della libreria potrebbe portare a risultati migliori anche in termini di tempo.

3.3.1 Spazio

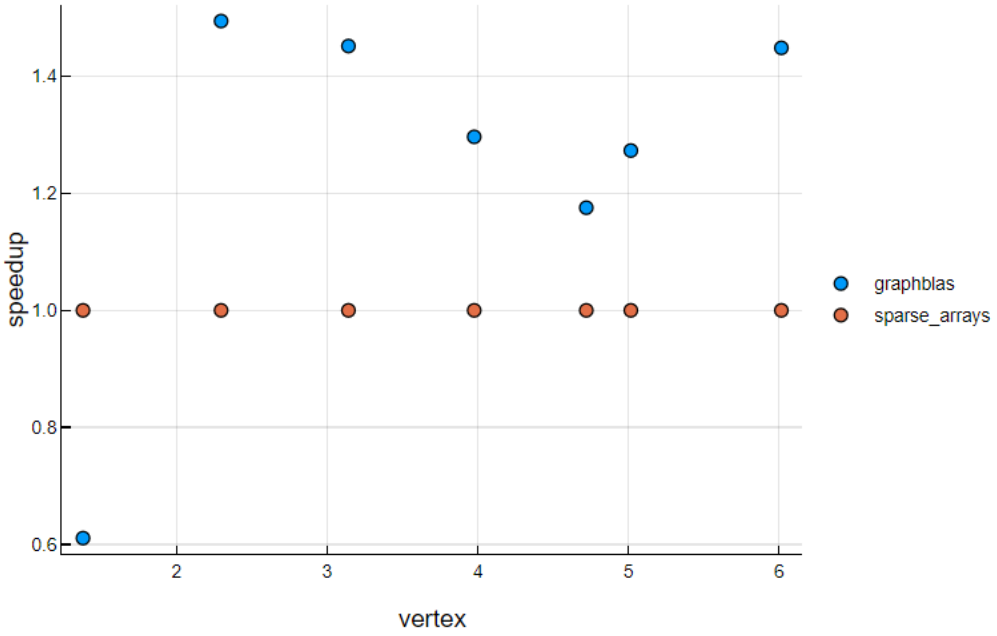

Figura 5: sparse-arrays vs graphblas

3.3.2 Tempo

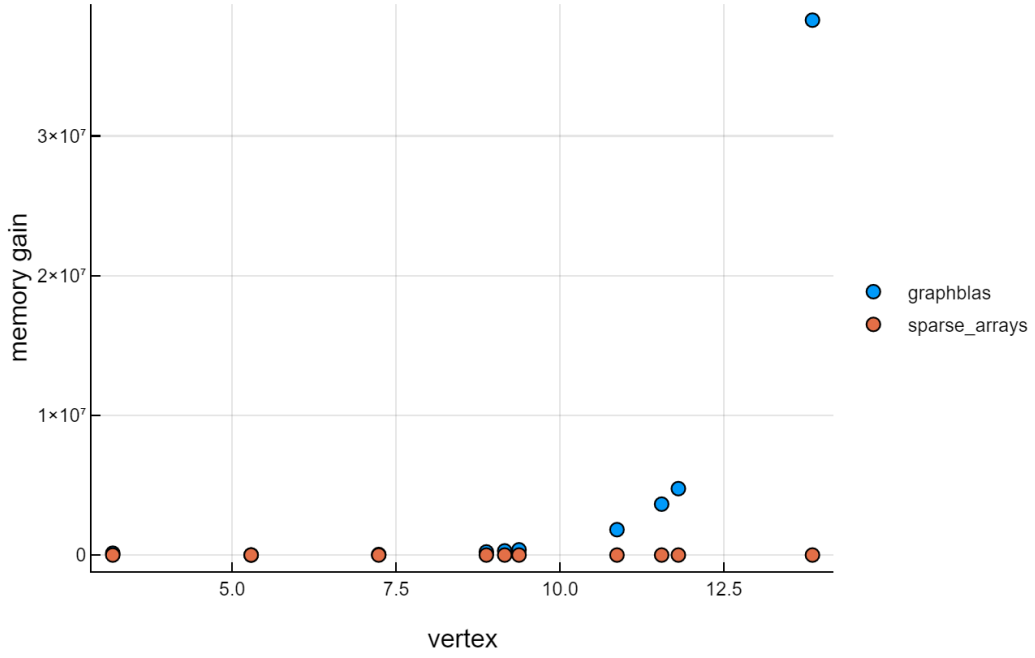

Figura 6: sparse-arrays vs graphblas

4 Benchmark: Casaletto.jl

Vengono di seguito riportate le tempistiche ottenute nell'esecuzione dell'intera pipeline con i dati presenti in casaletto.jl Per effettuare il calcolo delle tempistiche sono state estese le funzioni di benchmark precedenti, è stato però necessario usare la macro @time l'effettiva correttezza degli output generati in quanto la macro @btime comportava una computazione troppo onerosa portando la DGX-1 in stallo.

- SparseArrays

```
julia> @info("M0*M1'")
[ Info: M0*M1'

julia> t1 = Base.@elapsed @time M_0 * M_1';
  0.296353 seconds (242.89 k allocations: 16.629 MiB)

julia> @info("M1*M2'÷2 ")
[ Info: M1*M2'÷2

julia> t2 = Base.@elapsed @time (M_1 * M_2') .÷ 2;
  0.445707 seconds (378.63 k allocations: 41.495 MiB)

julia> @info("(M1*M3'/4)÷1")
[ Info: (M1*M3'/4)÷1

julia> t3 = Base.@elapsed @time ((M_2 * M_3')  ./ 4) .÷ 1;
  0.362946 seconds (388.66 k allocations: 29.470 MiB)
```

Figura 7:

- CuArrays

```
julia> @info("M0*M1'")
[ Info: M0*M1'

julia> t1 = Base.@elapsed @time M_0 * M_1';
  0.883744 seconds (519.24 k allocations: 26.413 MiB)

julia> @info("M1*M2'÷2 ")
[ Info: M1*M2'÷2

julia> t2 = Base.@elapsed @time (M_1 * M_2') .÷ 2;

 22.609182 seconds (25.34 M allocations: 1.247 GiB, 3.55% gc time)

julia> @info("(M1*M3'/4)÷1")
[ Info: (M1*M3'/4)÷1

julia> t3 = Base.@elapsed @time ((M_2 * M_3')  ./ 4) .÷ 1;
  0.810787 seconds (820.93 k allocations: 43.072 MiB)
```

Figura 8:

- CUSPARSE

```
julia> @info("M0*M1'")
[ Info: M0*M1'

julia> t1 = Base.@elapsed @time CUSPARSE.gemm('N','T',M_0,M_1,'O','O','O');
  0.018646 seconds (81 allocations: 1.828 KiB)

julia> @info("M1*M2'÷2 ")
[ Info: M1*M2'÷2

julia> t2 = Base.@elapsed @time collect((CUSPARSE.gemm('N','T',M_1,M_2,'O','O
','O'))) .÷ 2; #divisione su cpu
  0.704944 seconds (1.16 M allocations: 79.188 MiB, 2.22% gc time)

julia> @info("(M1*M3'/4)÷1")
[ Info: (M1*M3'/4)÷1

julia> t3 = Base.@elapsed @time (collect(CUSPARSE.gemm('N','T',M_2,M_3,'O','O
','O')) ./ 4) .÷ 1;
  0.216545 seconds (186.71 k allocations: 16.894 MiB)
```

Figura 9:

- GraphBLAS

```
julia> @info("M0*M1'")
[ Info: M0*M1'

julia> t1 = Base.@elapsed @time GrB_mxm(sigma_1, GrB_NULL, GrB_NULL, GxB_PLUS
_TIMES_INT64, M_0, M_1, desc);
  0.004173 seconds (6 allocations: 192 bytes)

julia> @info("M1*M2'÷2 ")
[ Info: M1*M2'÷2

julia> t2 = Base.@elapsed @time compute_s2()
  0.064422 seconds (14 allocations: 320 bytes)
0.064657794

julia> @info("(M2*M3'/4)÷1")
[ Info: (M2*M3'/4)÷1

julia> t3 = Base.@elapsed @time compute_s3();
  0.019052 seconds (22 allocations: 448 bytes)
```

Figura 10:

5 Conclusioni

I benchmark effettuati confermano gli effettivi benefit ipotizzati nell'utilizzo di librerie per calcolo su GPU, di fatti i maggiori speedup, rispetto a SparseArrays, si ottengono con l'utilizzo di CuArrays e CuArrays.CUSPARSE. CuArrays si è dimostrata la libreria per GPU più versatile ma mostra dei limiti in quanto le matrici dense in computazioni Lar-like (le matrici sono grandi e molto sparse) non sono supportate dall'hardware per gli

input maggiori di circa 10.0000 vertici. GraphBLAS non mostra uno speedup in termini di tempo che ne giustifica l'utilizzo, bensì in termini di memoria si dimostra di gran lunga il package migliore.

I benchmark su casalletto.jl evidenziano come il migliori compromesso in termini di prestazioni (tempo e memoria) e usabilità si ottiene in corrispondenza dell'utilizzo di GraphBLAS. L'intero codice del progetto è presente in una repo pubblica di github, al link: https://github.com/vcaravani/pdc_project.

6 Referenze

1. https://github.com/JuliaAttic/CUSPARSE.jl/blob/master/test/gemm.jl
2. https://docs.nvidia.com/cuda/cusparse/index.html
3. https://juliaobserver.com/packages/CUSPARSE
4. https://docs.julialang.org/en/v1/stdlib/SparseArrays/
5. https://nextjournal.com/sdanisch/julia-gpu-programming
6. https://github.com/JuliaGPU/CuArrays.jl/blob/master/src/sparse/wrappers.jl
7. https://abhinavmehndiratta.github.io/2019-06-07/an-introduction-to-graphblas
8. https://github.com/vcaravani/pdc_project/tree/master/references