Parallelizzazione CUDA/MPI di sciddicaT

Giorgio Andronico (mat. 227815)

5 ottobre 2021

Indice

Sommario

In questo documento verrà brevemente discusso il modello di automa cellulare *sciddicaT*, utilizzato per simulare in vari contesti lo scorrimento non-inerziale di un fluido su una superficie. Il contesto ivi utilizzato si basa su dati descriventi la frana del Tessina, avvenuta nel 1982 a Chies d'Alpago (BE). In seguito, verranno descritte le implementazioni parallele dell'algoritmo fondamentale dell'automa cellulare, sviluppate in CUDA e MPI. Verranno confrontati dettagli implementativi e prestazioni.

1 Introduzione e background tecnico

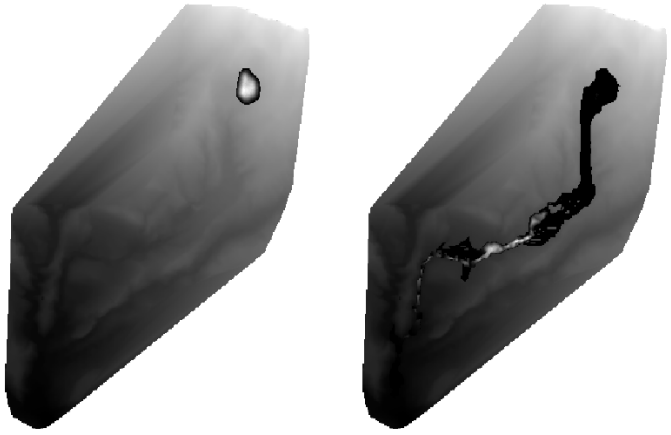
1.1 Il modello matematico

SciddicaT è un semplice simulatore del flusso di un fluido su una superficie, un automa cellulare che usa l'algoritmo di minimizzazione delle differenze. Quest'ultimo è usato per calcolare i flussi dei vicini di una determinata cella. Nonostante la sua intrinseca semplicità, il modello è in grado di simulare frane non-inerziali su superfici topografiche realmente esistenti, come la frana del Tessina, avvenuta nel 1982 in provincia di Belluno. Di seguito un breve riassunto del funzionamento del modello.

sciddicaT è una quintupla: $< R, X, S, P, \sigma >$.
Ove:

- R è la griglia bidimensionale di partenza
- X è un pattern geografico di celle contenente il vicinato di Von Neumann per ogni cella (che ricordiamo essere nord, sud, est, ovest)
- S è l'insieme degli stati di ogni cella. E' suddviso nei seguenti sottostati:
  - S_z è l'insieme dei valori che rappresentano l'altitudine topografica (metri sul mare)
  - S_h è l'insieme di valori che rappresenta la densità del fluido
  - S_o^4 è l'insieme dei valori che rappresenta i flussi uscenti da una data cella ai 4 vicini
- $P = p_\epsilon, p_r$ è l'insieme dei parametri che gestisce la dinamica della simulazione. In particolare, p_ϵ specifica la minima densità sotto la quale il fluido non può uscire dalla cella a causa dell'effetto di aderenza, mentre p_r è il fattore di smorzamento della fuoriuscita.
- $\sigma : S^5 \longrightarrow S$ è la funzione di transizione (deterministica) composta da tre processi elementari:
  - $\sigma_0 : S_o^4 \longrightarrow S_o^4$ imposta a zero i flussi uscenti da una data cella ai suoi quattro vicini
  - $\sigma_1 : (S_z \times S_h)^5 \times p_\epsilon \times p_r \longrightarrow S_o^4$ calcola il valore di fuoriuscita dalla cella centrale alle celle vicine grazie all'algoritmo di miminizzazione delle differenze, così da poter permettere una distribuzione equa sulle quattro celle adiacenti.
  - σ_2: $S_h \times (S_o^4)^5 \longrightarrow S_h$ aggiorna la densità del fluido della cella centrale considerando il cambiamento dei vicini avvenuto precedentemente.

Si noti che il dominio R è implicitamente considerato nei sottostati S.



(A) Initial configuration (B) Final configuration

Figura 1: Configurazione iniziale e finale dopo 4000 step.

1.2 Computing general-purpose su GPU

Negli ultimi anni si è diffusa rapidamente la pratica di utilizzare le GPU non solo per computer graphics, ma anche per altri scopi più generici, da qui l'acronimo GPGPU: calcolo General Purpose su Graphics Processing Unit. Le GPU, infatti, spesso sono utilizzate come acceleratori di calcoli nelle applicazioni di calcolo scientifico o nel caso delle reti neurali. Gli algoritmi data parallel sono tra i più adatti ad essere eseguiti sulle GPU, date le loro caratteristiche principali:

- Possibile gestire grandi stream di dati
- Parallelismo a grana fine di tipo SIMD
- Latenza bassa per operazioni floating point

Si noti infatti che le GPU hanno un'architettura drasticamente diversa da quella delle CPU, come illustrato in figura. Le GPU, infatti, presentano molte unità processanti elementari (ALU), quindi sono adatte a calcoli ripetitivi su grandi quantità di dati. La presenza di memoria sulla scheda evita il collo di bottiglia che si ottiene dovendo andare a reperire le informazioni in RAM: le GPU sono dotate di una propria VRAM. Le CPU, come si può vedere, hanno decisamente più risorse orientate al controllo e alla cache. In altre parole, le GPU sono utili per eseguire istruzioni semplici (dato il minore spazio dedicato all'unità di controllo sul chip) in contemporanea su grandi quantità di dati. Esistono diversi paradigmi di programmazione per programmare utilizzando le GPU: nel presente documento è fortemente sfruttato ed esplorato il modello CUDA.

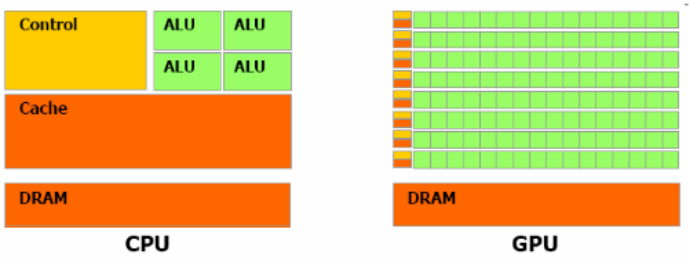


Figura 2: Visione architetturale.

1.3 Il modello CUDA

CUDA sta per "Compute Unified Device Architecture" ed è stato sviluppato da nVidia come tecnologia basata su un modello di programmazione general purpose, nel quale gli utenti lanciano da un processo host (eseguito sulla CPU) dei gruppi di thread sulle GPU, visti come coprocessori dedicati a operazioni data parallel. L'ambiente software è dotato di API per caricare programmi nelle GPU attraverso i "kernel", migrare dati tra memoria centrale e locale, il tutto avulso da riferimenti ad oggetti grafici. CPU e GPU accedono a memorie separate, ed è necessario allocare e trasferire esplicitamente dati tra le due memorie. Ad essere gestito dall'host sono:

- Allocazione della memoria del device
- Trasferimento da e per la memoria del device
- Avvio algoritmi paralleli sul dispositivo

1.3.1 Threading e Memoria

Le porzioni parallele di un codice, dunque, sono eseguite su un device (GPU). E' un coprocessore per la CPU o host con una propria memoria (device memory), ed esegue molti threads in parallelo attraverso i kernel. I thread sono molto leggeri, dotati di un overhead di creazione bassissimo, ed infatti le GPU richiedono la creazione di migliaia di threads per raggiungere la massima efficienza.

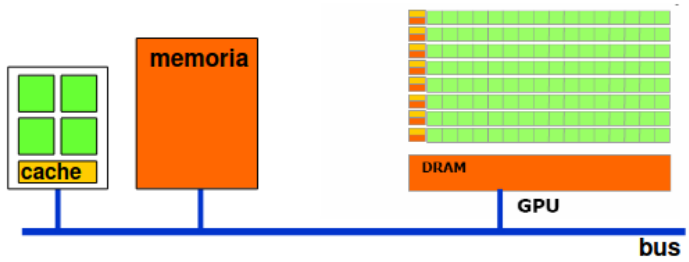


Figura 3: Come la GPU interagisce con il sistema.

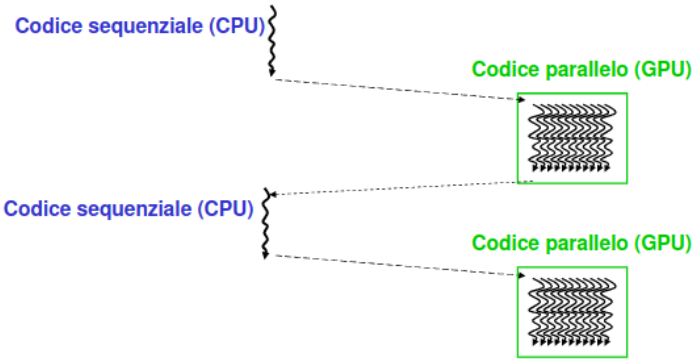


Figura 4: Sezioni seriali nella CPU si alternano a sezioni parallele massicce nella GPU.

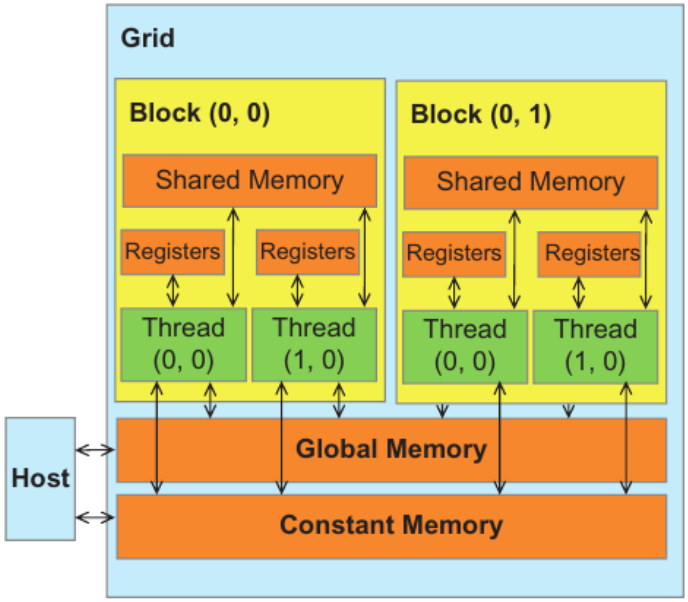


Figura 5: Layout dei thread e della memoria in CUDA.

Più precisamente, i kernel sono eseguiti su una griglia (1D o 2D), composta da blocchi di thread. Ogni blocco della griglia è un insieme di thread, e può avere una struttura 1D, 2D o 3D. Tutti i blocchi di una stessa griglia hanno la stessa struttura. Tale organizzazione semplifica i riferimenti in memoria quando si opera su dati multidimensionali, come nel caso dell'image processing o anche di sciddicaT. La memoria globale è il principale mezzo di comunicazione tra host e device, il suo contenuto è visibile a tutti i thread, che fornisce elevati tempi di accesso. Esiste anche la memoria shared, un blocco di memoria notevolmente più veloce ma visibile solo ai thread di un determinato blocco: esiste un'area

di shared memory per ogni blocco lanciato. Si noti che una GPU è composta da diversi Streaming Multiprocessors, composti da un quantitativo di memoria shared e un numero variabile di ALU. Un blocco di thread è assegnato ad un unico streaming multiprocessor: questo significa gestire un gruppo di threads come una unità logica, fornendo ai threads dello stesso blocco la possibilità di utilizzare direttive di sincronizzazione e memoria condivisa. La scelta di non fornire direttive di sincronizzazione globali (tra threads di diversi blocchi) permette di gestire i blocchi di threads in modo indipendente: l'ordine di esecuzione dei blocchi di threads non è fissato a priori. I threads vengono ulteriormente suddivisi in gruppi, detti warp. La dimensione del warp dipende dall'architettura sotto esame, e i thread appartenenti allo stesso warp vengono gestiti dall'unità di controllo (warp scheduler) in modo congiunto. Per sfruttare al massimo il parallelismo intrinseco del SM, i thread dello stesso warp devono eseguire la stessa istruzione: se questa condizione non si verifica si parla di divergenza dei threads. Se i thread dello stesso warp stanno eseguendo istruzioni diverse l'unità di controllo non può gestire tutto il warp: deve seguire le successioni di istruzioni per ogni singolo thread (o per sottoinsiemi omogenei di threads) in modo seriale.

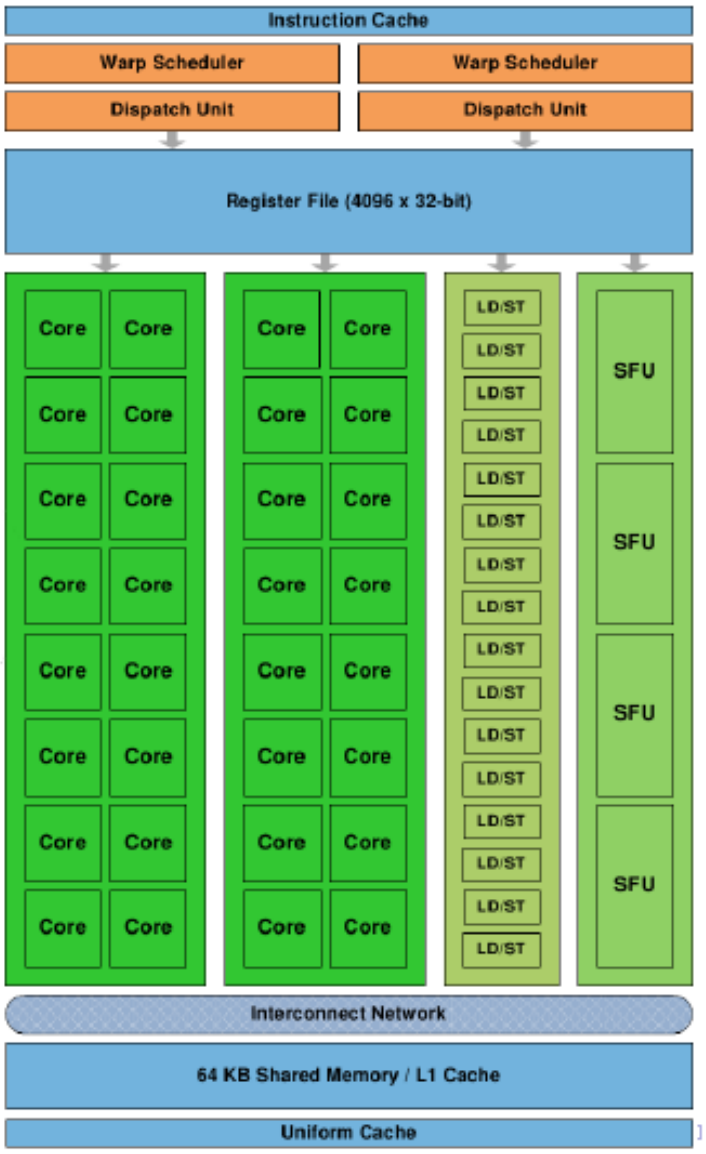


Figura 6: Forma di uno Streaming Multiprocessor.

2 Implementazioni parallele

L'algoritmo è stato implementato in maniera parallela utilizzando diverse tecniche. Nello specifico, è stata utilizzata una workstation dotata di CPU Intel Xeon E5440 @ 2.83GHz e due nVidia GTX 980. Di conseguenza, sono state implementate quattro versioni:

- Parallelizzazione *straightforward*:
  - Monolithic kernel
  - Grid-stride method
- Parallelizzazione con *tile* e *Shared Memory*:
  - Con halo cells
  - Senza halo cells
- Parallelizzazione Multi-GPU (usando MPI) della versione più performante

Si noti che i tre buffer Sz (dim. 610x496), Sh (dim. 610x496), e Sf (dim. 2440x496), pur essendo logicamente delle matrici bidimensionali, sono salvate in memoria come buffer lineari con tecnica *row-major*. Al momento CUDA permette di operare solo con buffer linearmente memorizzati in RAM.

Si noti anche che il numero di thread totale lanciato, in base alla dimensione scelta del singolo blocco di thread, può variare leggermente: in generale, la formula per calcolare questi parametri è: $\lceil numT_x = 610/bSize_x \rceil, \lceil numT_y = 496/bSize_y \rceil$.

E' possibile che vengano lanciati più thread di quelli effettivamente richiesti, ma l'effetto di ciò è trascurabile, dato che l'algoritmo usa solo una piccola frazione di ciò che la GTX 980 ha a disposizione (ovvero una griglia di dimensione (2147483647, 65535, 65535)[Kin15]).

Il programma è diviso in tre kernel, direttamente corrispondenti a σ_0, σ_1, σ_2.

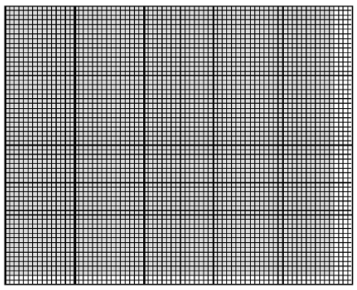


Figura 7: Grid. i quadratini più chiari sono i thread inutilizzati. Dimensioni relative.

2.1 Implementazione con metodo *monolithic kernel*

Questo tipo di implementazione parallela è la più semplice ed immediata, come suggerisce il nome stesso: dall'inglese *straightforward* che significa "immediato" in italiano. In questo algoritmo, viene lanciata una grid di thread di dimensione 610x496, e ciascun thread si occupa di una singola cella dei buffer S_h e S_z. Per quanto riguarda il buffer S_f, il thread (i, j) si occuperà di $S_f(n \times 610 \times 496 + i \times 496 + j)$, con $i = 0...3$. Quindi, si occuperà di 4 celle.

2.2 Implementazione con metodo *grid-stride loop*

Con questa tecnica, un thread singolo si occupa di più di una cella di ciascun buffer. Il vantaggio di questa tecnica è la riutilizzabilità: piuttosto che assumere che la griglia sia abbastanza grande per coprire tutta la matrice, semplicemente il kernel lavora su celle a distanza di una grid-size. Assumendo una grid size di 30, il thread 0 lavorerà sulla cella 0, 30, 60 etc., mentre il thread 1 lavorerà sulla cella 1, 31, 61 e così via, coprendo tutto il dominio. E' stato anche notevolmente più facile debuggare il programma usando questa tecnica, in quanto semplicemente si può lanciare un kernel dove si ha una grid con dentro un solo thread, simulando quindi l'esecuzione seriale.

2.3 Parallelizzazione utilizzando la Shared Memory (senza halo cells)

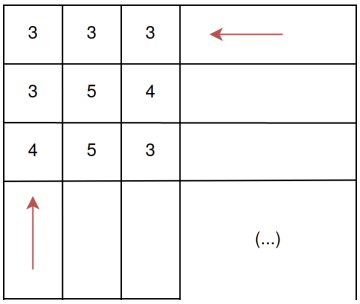


Figura 8: Heatmap dell'accesso alle celle. Le frecce indicano i bordi del blocco.

Nelle versioni descritte in 2.1 e 2.2 vengono effettuati spesso degli accessi non necessari alla memoria globale. Prendiamo come esempio un blocco di dimensione 3x3. Ci sono alcune caselle che vengono accedute diverse volte, perchè sono vicine di più thread diversi. Tuttavia, ricordando che ogni blocco ha a disposizione una sua rapida shared memory, è possibile sfruttare il principio di località dei dati (presente in sciddicaT) per minimizzare il numero di accessi alla memoria globale. Nello specifico, ogni thread carica la sua casella (i, j) in shared memory. Dopo di che, se deve accedere ad un vicino, andrà a fare il fetch dalla memoria shared e non più da quella globale. Infatti, al thread in posizione (i, j) è garantita la presenza in memoria condivisa di valori in $(i-1, j)$, $(i+1, j)$, $(i, j+1)$, $(i, j-1)$. (ammesso che il thread non si trovi "ai bordi" del proprio blocco. In tal caso il fetch dev'essere effettuato comunque da memoria globale. Si veda la sezione 2.4 per maggiori informazioni.) La dimensione del tile è sempre pari, in questo caso, alla dimensione del blocco di thread.

Si noti, inoltre, che la shared memory è usata:

- Per S_h, in σ_2
- Per S_z, in σ_2
- Per S_f, in σ_3

Infatti, in σ_2 il thread (i, j) accede a caselle diverse da (i, j) solo sui buffer S_h ed S_z; lo stesso dicasi per S_f e σ_3. In σ_3, dato il particolare pattern di accesso al buffer S_f (illustrato in fig. 9), è stato necessario salvare più informazioni all'interno del tile, che è stato reso "tridimensionale". Ogni thread carica la propria casella (i, j) di S_f ed anche altre 3: $(n \times 610 \times 496 + i \times 496 + j)$, con $i = 1...3$.

2.4 Parallelizzazione utilizzando la Shared Memory (con halo cells)

Per ridurre ulteriormente gli accessi alla memoria globale, è stato effettuato un miglioramento alla versione illustrata in sezione 2.3. Per ogni tile, sono state utilizzati quattro mini-buffer monodimensionali che rappresentano le righe attorno a quel tile. Questi buffer, ovviamente, sono riempiti solo dai thread che si trovano ai bordi del tile, come in fig. 10. Questo metodo riduce a zero gli accessi alla memoria globale durante la computazione.

La scelta di utilizzare quattro array rappresentanti le righe halo, al posto dell'approccio più "tradizionale" di "allargamento del tile" e maschera per convolution, è stato adottato per un semplice motivo. Il primo, è che sciddicaT non utilizza una vera e propria maschera. L'allargamento avrebbe come effetto la memorizzazione di 4 celle inutili (dato che non si considerano i vicini diagonali) per tile. Tuttavia, è possibile che questo dia luogo a control flow divergence, avendo molti branch if-else all'interno del codice. L'effetto di ciò però, probabilmente, è totalmente trascurabile in quanto le prestazioni dell'algoritmo sono ottime rispetto alla versione straightforward (si veda il punto 3.4).

2.5 Parallelizzazione utilizzando la Shared Memory (senza halo cells) e MPI

La versione più performante del codice è stata quella illustrata in sezione 2.3 (maggiori informazioni nella sezione numero 3, "Valutazioni delle prestazioni". Questa è stata la base dell'implementazione

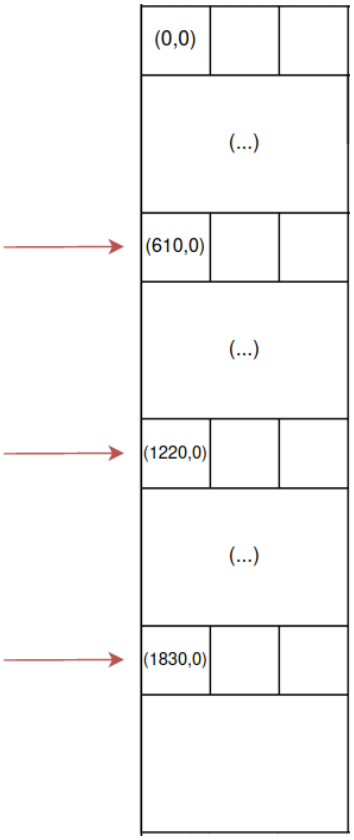


Figura 9: Pattern di accesso di un thread di esempio a S_f.

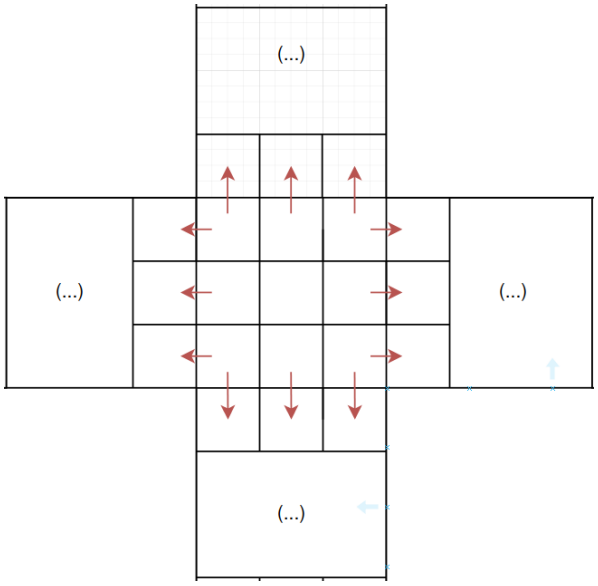


Figura 10: Pattern di caricamento in righe halo.

multiGPU con MPI. La versione finale dell'algoritmo richiede di essere eseguito con due processi: ciascuno verrà assegnato ad una singola GPU mediante una apposita istruzione CUDA. Il processo 0 fa le veci sia del master che del worker, mentre il processo 1 fa esclusivamente le veci del worker. La suddivisione delle strutture dati è stata fatta "per riga": i.e., le prime 305 righe di S_z e S_f vanno al processo (e quindi alla GPU) 0, mentre le seconde 305 righe vanno al processo 1. Tuttavia, trattandosi di un automa cellulare, è stato necessario tenere in considerazione gli accessi di una data cella ai propri

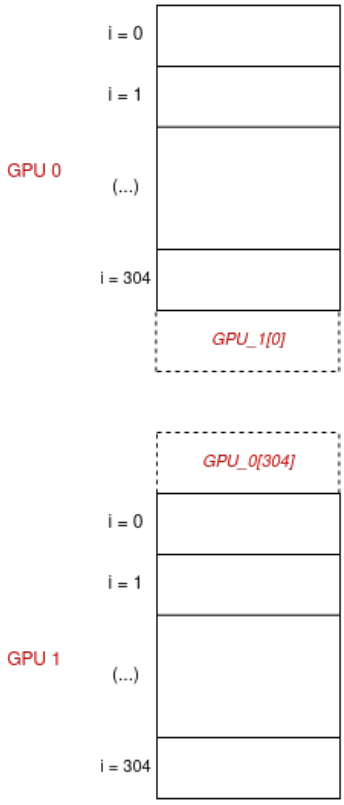


Figura 11: Pattern di accesso di un thread di esempio a S_f.

vicini a un livello macroscopico. Nella fattispecie, ogni GPU è dotata di una "riga halo" sia per il buffer S_z che S_h, per un totale di due righe halo per processo. Come si evince dalla figura 11, in aggiunta alla propria porzione di competenza, il processo 0 contiene una copia della prima riga della porzione del processo 1 (e viceversa). Un thread, posizionato sull'ultima riga della porzione di competenza, potrebbe necessitare di accedere alla prima riga della porzione che non gli appartiene (nel tentativo di reperire informazioni del "vicino di sotto"). In tal caso, il thread (i, j) accede alla j-esima casella della riga halo. Alla fine di ogni ciclo della simulazione, i due processi aggiornano vicendevolmente il contenuto delle righe halo, per mantenere consistente lo stato globale delle strutture dati. Questa implementazione parallela riesce a computare il flusso originale con circa il 75% di accuratezza dopo 4000 passi, come mostrato nella figura 12.

3 Valutazione delle prestazioni

Tipo implementazione	Tempo parallelo (8x8)	Speedup
Monolithic kernel	1.65s	40.0
Grid-stride loops	3.48s	19.4
Tiled, no halo	1.44s	46.9
Tiled, halo	1.45s	46.5
Multi-GPU, tiled, no halo	2.83s	23.8

Tabella 1: Tempi di esecuzione complessivi e relativo speedup.

Si noti che l'implementazione seriale dell'algoritmo richiede un tempo di 67.5 secondi per eseguire 4000 passi della simulazione. Vengono utilizzati come tempi di riferimento quelli rilevati dalle versioni con dimensione blocco pari a 8. Questa dimensione si è rivelata essere quella in grado di dare prestazioni globalmente migliori sull'arco di tutte le implementazioni; ci si riferisca alle sezioni successive per

l'esame di altre dimensioni blocco. Si può notare come la versione meno performante sia la grid-stride loops. La conclusione da trarre è ovvia: l'estendibilità del codice grid-stride arriva con un costo di efficienza non indifferente, dato che ogni thread si occupa di più di una cella, oltre a dover calcolare gli indici del ciclo for. La versione strutturalmente più vicina alla grid-stride, ovvero la monolithic, riesce a performare molto meglio in virtù del fatto che ogni thread svolge un solo compito. Il calcolo degli indici del ciclo nella versione grid-stride non compensa computazionalmente l'istruzione di controllo presente nella versione monolithic (che assicura che solo i thread associabili ad una cella della matrice lavorino) all'inizio di ogni kernel.

L'implementazione monolithic kernel presenta delle prestazioni accettabili. Uno speedup notevole rispetto alla versione seriale, ma con un fattore limitante: gli accessi ripetuti, da parte di diversi SM, alle stesse posizioni di memoria globale (si veda la figura 8).

Le implementazioni più performanti sono le due che fanno uso della shared memory: minimizzando il numero di accessi duplicati alla memoria globale, il vantaggio prestazionale è ovvio. Non si nota una particolare differenza tra halo e no-halo, imputabile alle dimensioni relativamente ristrette della matrice. L'utilità delle righe halo, con tutta probabilità, impiega un pò a "scalare". Tuttavia, non è semplicemente possibile allargare le dimensioni della matrice per verificare empiricamente questa informazione, in quanto il dataset è di dimensioni fissate.

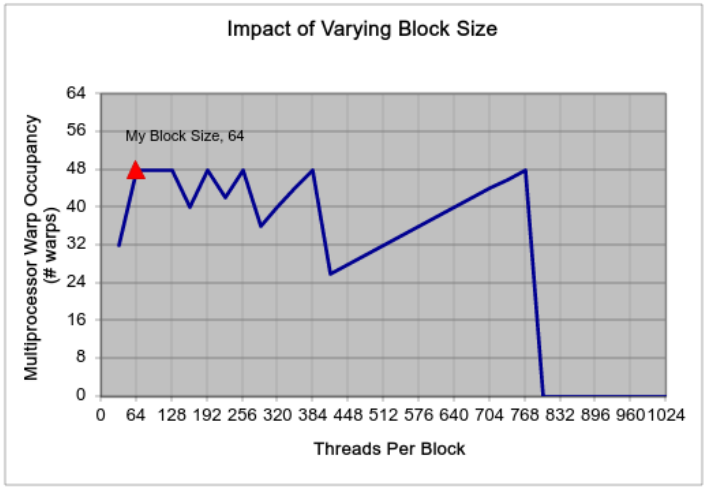
Si noti che non sono state rilevati cambiamenti nel numero di registri usati al variare della dimensione dei blocchi di thread.

Figura 12: Computazione del flusso. In grigio il flusso computato dall'implementazione MPI.

Al penultimo posto si colloca la versione multi-GPU, con un peggioramento di più del doppio rispetto alla versione singola GPU dell'algoritmo. Ciò perchè una singola GTX 980 è più che sufficiente per processare sciddicaT, date le dimensioni relativamente ristrette del dominio. Il tempo aggiunto è dovuto all'overhead dato dalla comunicazione tra processi. Infatti, ad ogni ciclo, vengono effettuate due operazioni di invio e ricezione per processo, per un totale di 16000 operazioni extra. L'overhead è reso peggiore dal fatto che la versione di OpenMPI utilizzata su JPDM2 non è CUDA-aware, dunque per ogni operazione di invio, il buffer dev'essere copiato dalla GPU all'host, e poi ricopiato dall'host alla GPU; non è possibile inviare direttamente buffer di memoria GPU.

3.1 Implementazione a Monolithic Kernel

Come si può notare, il kernel resetFlows impiega sempre lo stesso tempo a prescindere dalla dimensione del blocco: questo è perchè ogni thread usa il kernel semplicemente per impostare il valore della propria cella a 0. Non vengono effettuate operazioni di calcolo di nessun tipo. Il kernel flowsComputation invece ottiene prestazioni ottimali nel caso di blocco 8x8 o 16x16, in quanto l'occupazione dei multiprocessori scende al 50% nel caso di blocco 4x4. Il kernel widthUpdate performa leggermente meglio con blocco 16x16, ma di pochissimo. Entrambe le affermazioni sono corroborate dal fatto che la dimensione 8x8 porta alla maggiore occupazione contemporanea possibile dei multiprocessori.



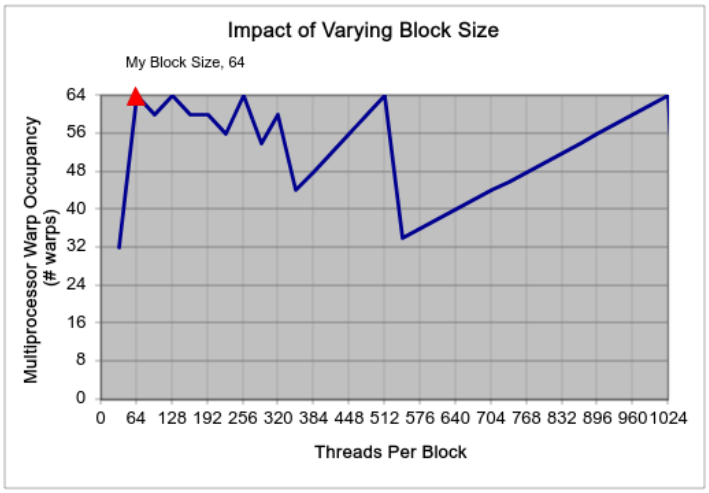


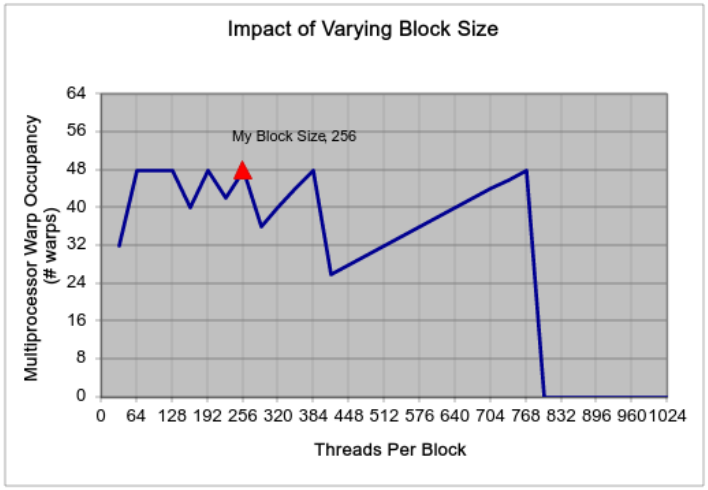
Figura 13: Monolithic. Sopra: flowsComputation, sotto, widthUpdate.

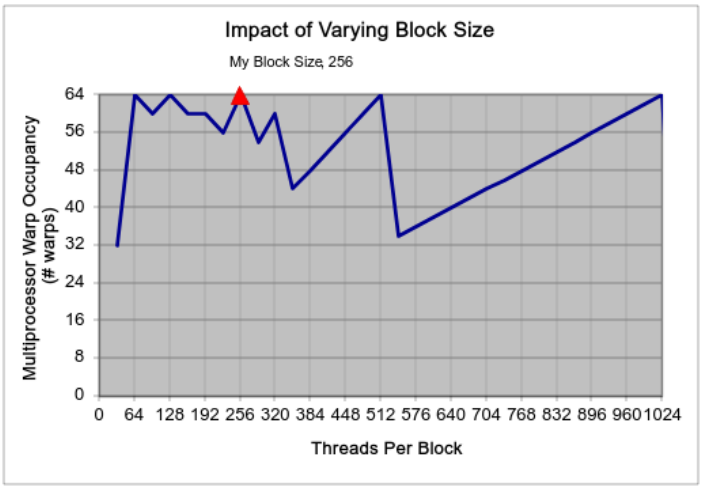
Kernel (monolithic)	Tempo (4x4)	Tempo (8x8)	Tempo (16x16)
resetFlows	0.218s	0.218s	0.223s
flowsComputation	1.594s	0.856s	0.873s
widthUpdate	0.383s	0.368s	0.328s

Tabella 2: Tempi di esecuzione dell'implementazione a monolithic kernel.

3.2 Implementazione a Grid-stride loops

Il kernel resetFlows si mantiene pressochè costante (non tanto quanto la versione monolithic) attraverso tutte e 3 le dimensioni di blocco, con un leggero svantaggio verso la dimensione 4x4. Questo è dovuto al fatto che la occupazione degli SM non è massima nel caso di dimensione 4x4. Per questo kernel, nel caso di blocco 4x4, l'occupazione dei multiprocessori si ferma al 50%, con 1024 *thread*, 32 *warp* e 32 blocchi attivi per multiprocessore, mentre arriva al 100% nel caso 8x8 o 16x16. Riguardo il kernel flowsComputation, le prestazioni sono notevolmente più basse nel caso 4x4 (i multiprocessori sono a un tasso di occupazione del 50%), dove 8x8 si conferma ancora una volta una giusta via di mezzo: tuttavia, l'occupazione dei multiprocessori è la stessa (75%) sia nel caso 8x8 che 16x16. Stesso dicasi per widthUpdate, dove però l'occupazione nei casi 4x4 e 8x8 raggiunge il 100%.





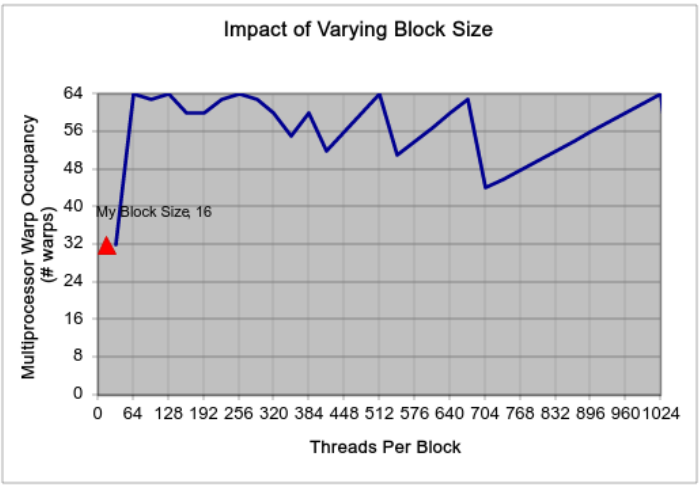


Figura 14: Grid-stride. Sopra: flowsComputation, centro: widthUpdate, sotto: resetFlows.

Kernel (grid-stride)	Tempo (4x4)	Tempo (8x8)	Tempo (16x16)
resetFlows	0.604s	0.693s	0.676s
flowsComputation	1.927s	1.090s	1.326s
widthUpdate	0.933s	0.859s	0.831s

Tabella 3: Tempi di esecuzione dell'implementazione a grid-stride loops.

3.3 Implementazione a tile, no halo

E' possibile notare come il kernel resetFlows rimanga pressochè costante nelle prestazioni sull'arco di tutte e 3 le dimensioni blocco (che, si noti, corrispondono direttamente all'ampiezza dei tile utilizzati). Ciò perchè il kernel non fa assolutamente uso della memoria shared, dunque non ottenendo vantaggi prestazionali rispetto alle altre versioni. I vantaggi vengono visti sopratutto sugli altri kernel. In particolare, come si evince dalla Figura 15, l'utilizzo della memoria condivisa è sufficiente a sfruttare quasi al massimo la concorrenza dei multiprocessori. Nel caso del kernel flowsComputation (dove vengono utilizzati 512 byte di memoria shared), con 1536 *thread*, 48 *warp* e 24 blocchi attivi per multiprocessore, si raggiunge una occupazione del 75%. Per raggiungere una occupazione del 100% anche in questo caso, i registri utilizzati dovrebbero essere al massimo 32, ma *nvcc* ha allocato l'uso di 36 registri (si veda la Figura 17). Nel caso del kernel widthUpdate (dove si usano 2048 byte di memoria shared) si raggiunge una occupazione del 100%, avendo 2048 *thread*, 64 *warp* e 32 blocchi attivi per multiprocessore. Il quantitativo di memoria shared utilizzato, inoltre, permette di sfruttare al massimo la concorrenza dei multiprocessori (si veda la Figura 16).

Kernel (tiled, no halo)	Tempo (4x4)	Tempo (8x8)	Tempo (16x16)
resetFlows	0.218s	0.218s	0.223s
flowsComputation	1.738s	0.840s	0.006s
widthUpdate	0.475s	0.362s	0.006s

Tabella 4: Tempi di esecuzione dell'implementazione a tile, no halo.

3.4 Implementazione a tile, con halo

L'implementazione che fa uso delle celle halo è prestazionalmente pressochè identica a quella che non fa uso delle celle halo. Ciò è probabilmente dovuto alle dimensioni ristrette del dominio. Anche qui, nel caso di dimensione blocco 4x4, si ottiene una occupazione dei multiprocessori pari al 50%, mentre nel caso di 8x8 e 16x16 si ha una occupazione del 75%, questa volta per entrambi i kernel: si veda la Figura 18. E anche qui, il quantitativo di memoria shared utilizzata (rispettivamente 1024 e 3072 byte per il kernel flowsComputation e widthUpdate), non infliusce sul tasso di occupazione del 75%. Per influire in maniera sostanziale, l'uso da parte dell'utente della memoria shared dovrebbe superare i 4096 byte, scenario non contemplato in quanto l'algoritmo usa già in maniera più efficiente possibile la memoria shared, dal punto di vista dello spazio (grafico non riportato per brevità). Ad influire (in maniera uguale per entrambi i kernel), come si evince in figura 19, è il numero di registri usati - tuttavia, questo numero viene determinato a tempo di compilazione dal compilatore CUDA, e non può essere modificato.

Kernel (tiled, halo)	Tempo (4x4)	Tempo (8x8)	Tempo (16x16)
resetFlows	0.218s	0.218s	0.222s
flowsComputation	1.667s	0.839s	0.006s
widthUpdate	0.626s	0.397s	0.006s

Tabella 5: Tempi di esecuzione dell'implementazione a tile, con halo.

3.5 Implementazione multi-GPU tiled, no halo

L'implementazione multi-GPU, basata sulla versione tiled senza halo (in quanto risultata la più performante) presenta un forte decremento delle prestazioni rispetto alla versione a GPU singola: infatti,

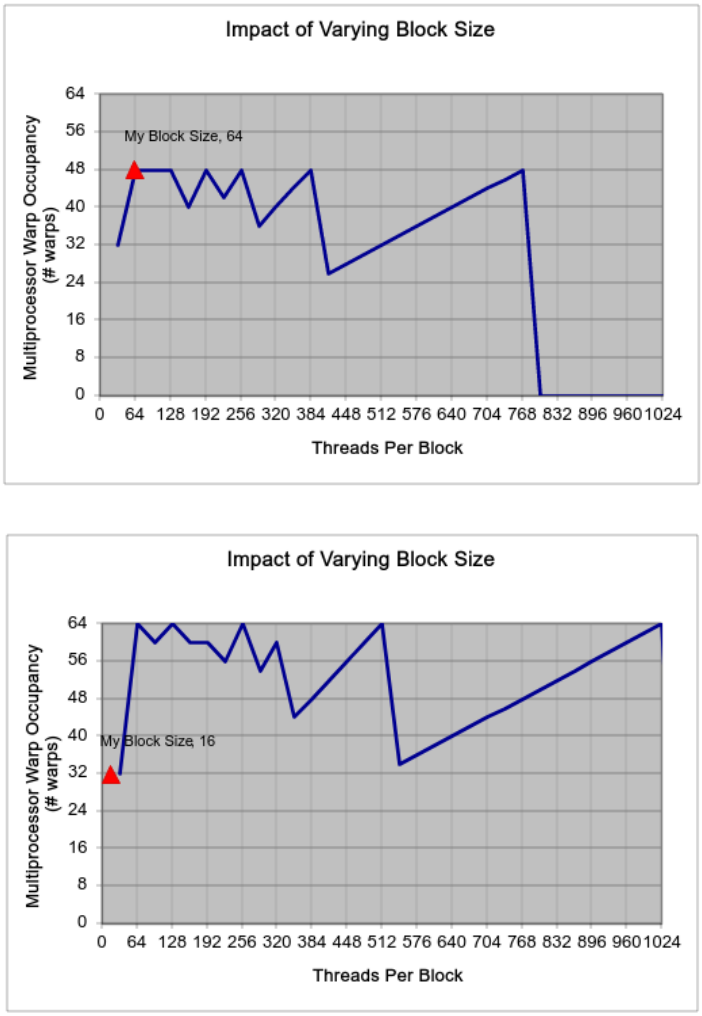


Figura 15: Utilizzo multiprocessori (tiled, no halo) al variare della dimensione del blocco. Sopra: flowsComputation, sotto: widthUpdate.

nonostante la variazione di dimensione del blocco thread, le prestazioni non arrivano mai ai livelli della versione tiled, no halo a GPU singola. Di seguito un breve riassunto delle tempistiche ottenute, con riferimento ad una sola GPU (i tempi di esecuzione dei due processi sono identici, con al più qualche millisecondo di scarto). Si ricorda che il decremento delle prestazioni è dovuto all'overhead che incorre nel momento in cui c'è molta comunicazione tra un processo e l'altro. Intuitivamente dividere il lavoro tra due GPU dovrebbe migliorarne le prestazioni, ma non è sempre così, specie se si parla di domini piuttosto ristretti dal punto di vista delle dimensioni, come sciddicaT.

E' possibile notare come il kernel flowsComputation sia decisamente il più "pesante" dal punto di vista computazionale. Ciò succede perchè questo è l'unico kernel che fa uso delle celle halo, sia per il buffer S_z che S_h, e quindi per ognuno dei 4000 passi della simulazione, vengono effettuate 8 operazioni di invio/ricezione, un notevole collo di bottiglia sulle prestazioni.

Kernel (multiGPU, tiled, no halo)	Tempo (4x4)	Speedup	Tempo (8x8)	Speedup
resetFlows	0.365s	0.597	0.334s	0.652
flowsComputation	2.509s	0.690	2.004s	0.419
widthUpdate	0.551s	0.862	0.486s	0.744

Tabella 6: Tempi di esecuzione dell'implementazione multiGPU a tile, no halo.

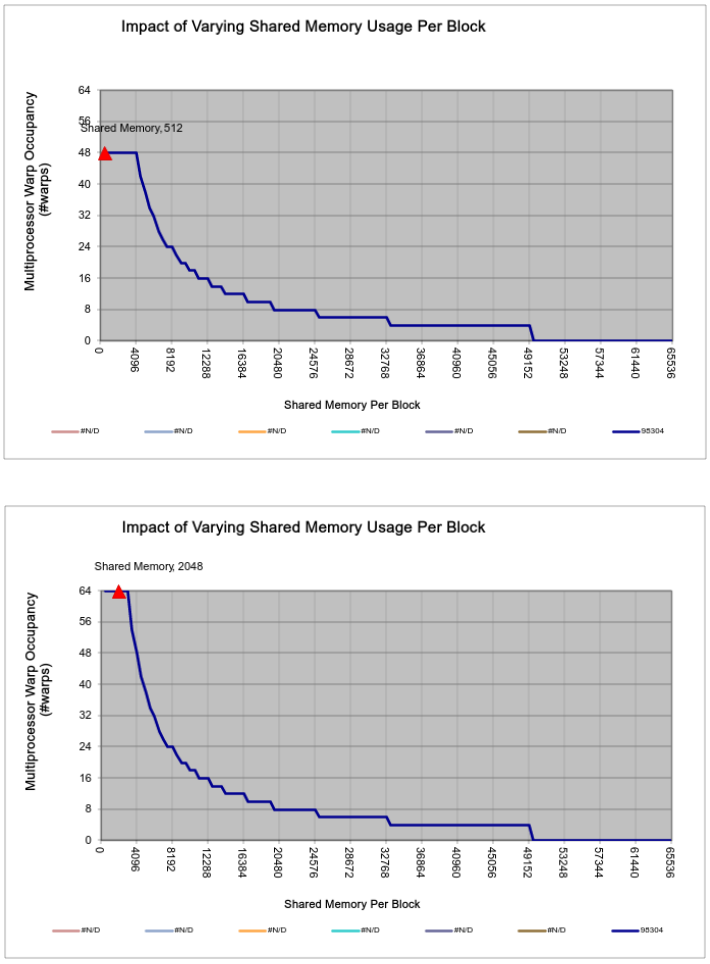


Figura 16: Utilizzo multiprocessori (tiled, no halo) al variare dell'uso della memoria condivisa. Sopra: flowsComputation, sotto: widthUpdate.

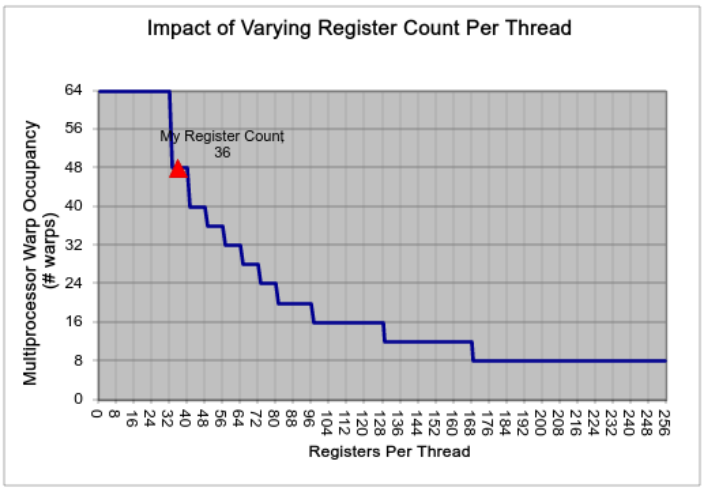


Figura 17: Occupazione multiprocessori (tiled, no halo) al variare del numero di registri. Kernel flowsComputation.

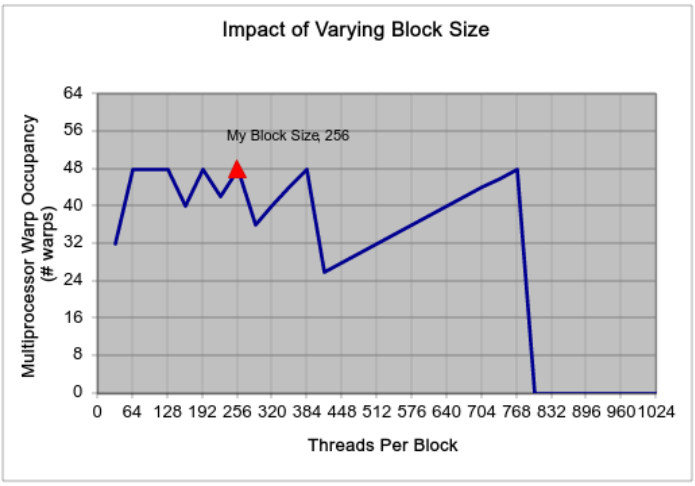


Figura 18: Utilizzo multiprocessori (tiled, halo) al variare della dimensione del blocco.

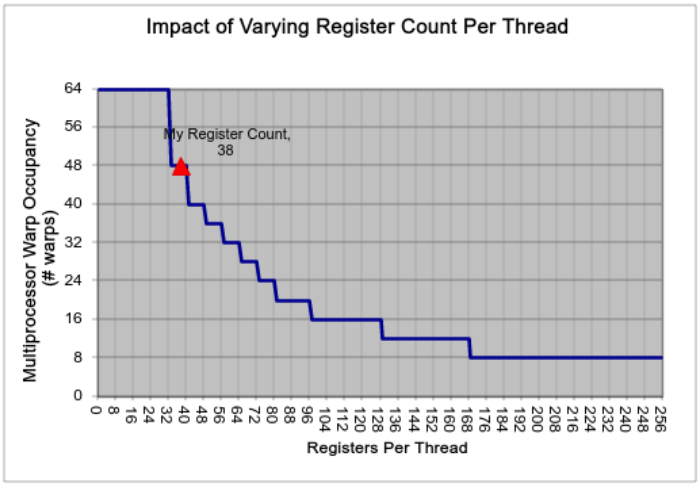


Figura 19: Utilizzo multiprocessori (tiled, halo) al variare del numero di registri.

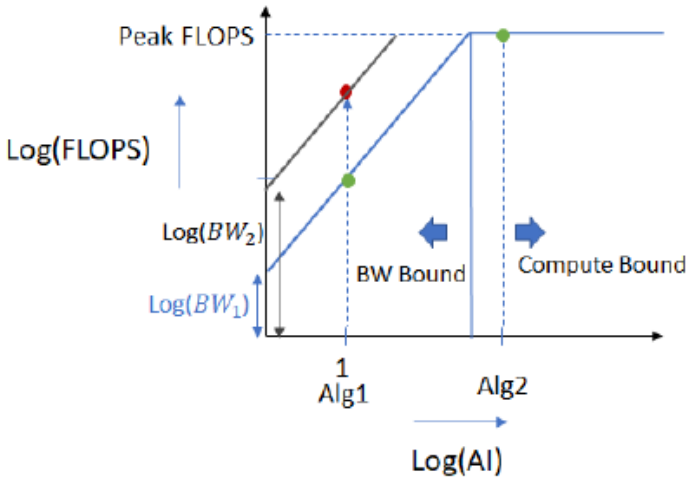


Figura 20: Esempio di modello roofline per due algoritmi.

Kernel (100 step)	Tempo	FLOP	Op. load	Op. store
widthUpdate	0.005s	1968128	2214146	61504
flowsComputation	0.023s	10219620	2460162	532
resetFlows	0.009s	0	2	246016

Tabella 7: Metriche di prestazioni, straightforward.

4 Il roofline model

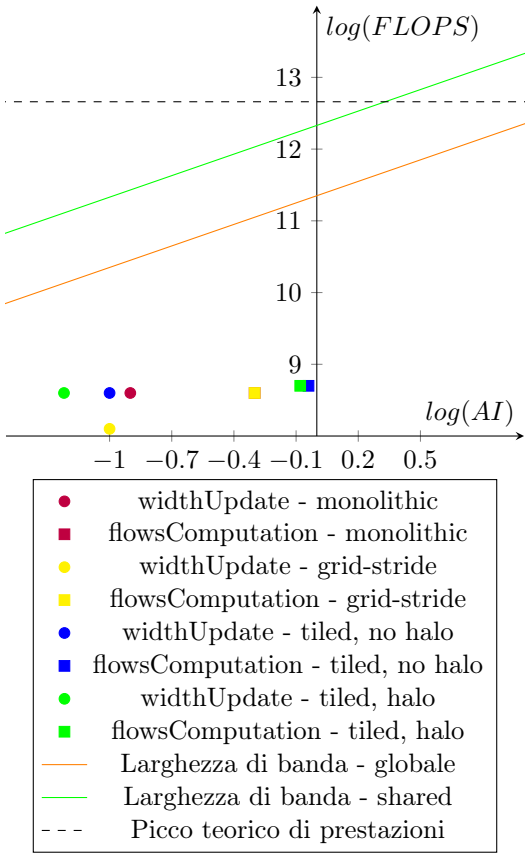
Il roofline model è un modello matematico che permette, intuitivamente, di stimare le performance (sia computazionali che di memoria) di un algoritmo in maniera visiva. Infatti, il modello prevede la creazione di un grafico cartesiano che mostra le limitazioni intrinseche dell'hardware usato e le potenziali ottimizzazioni da poter applicare. Da un punto di vista computazionale, un dispositivo è caratterizzato da due parametri: il picco teorico di prestazioni (il numero massimo di operazioni floating-point effettuate in un secondo, FLOPS) e il picco di banda (il numero massimo di byte che possono essere reperiti/memorizzati al secondo, BW). Nello specifico, la GTX 980 ha un picco di 4.612 GFLOPs dove la larghezza di banda massima è 224.32GB/s (per quanto riguarda la memoria globale) e 2119.35 GB/s (per quanto riguarda le operazioni a 64-bit della memoria condivisa).
Un algoritmo (e quindi un kernel CUDA) può essere caratterizzato da una quantità chiamata Intensità Aritmetica, data dal numero di operazioni floating-point per byte. $AI = [\frac{FLOP}{byte}]$. Sapendo che $FLOPS = AI \times BW$ (da $\frac{FLOP}{s} = \frac{FLOP}{byte} \times \frac{byte}{s}$), passando tutto al logaritmo otteniamo $log(FLOPS) = log(AI) + log(BW)$. Interpretando la formula ottenuta come l'equazione di una linea $y = mx + c$ sul piano, otteniamo che $y = log(FLOPS)$, $m = 1$, e $c = log(BW)$. Nel plot roofline, il picco di performance è un punto sull'asse verticale, mentre il picco di memoria è il punto d'intersezione tra la retta descritta sopra e l'asse verticale, come si vede in Figura 20.
Nelle Tabelle 7-8-9-10 è possibile osservare i dati a partire dai quali calcolare il roofline plot dei vari kernel per le 4 implementazioni svolte. Si noti che tutti i dati sono stati presi con dimensioni blocco 8x8 in quanto si è dimostrata essere la dimensione a grandi linee ottimale (si veda il Cap. 3). Nel caso di implementazioni shared si considera una TILE_WIDTH=8, pari quindi alla dimensione del blocco. Inoltre, i dati raccolti fanno riferimento a una esecuzione della simulazione limitata a 100 passi - effettuare il profiling per l'intera esecuzione di 4000 passi risulta impossibile, in quanto la macchina JPDM2 sul quale è stata eseguita questa operazione, non consente di eseguire processi che superano i 6 minuti in durata.

Kernel (100 step)	Tempo	FLOP	Op. load	Op. store
widthUpdate	0.018s	2420480	2731970	301816
flowsComputation	0.029s	12404292	3035522	1625
resetFlows	0.022s	0	2	1207264

Tabella 8: Metriche di prestazioni, grid-stride.

Kernel (100 step)	Tempo	FLOP	Load (G)	Store (G)	Load (S)	Store (S)
widthUpdate	0.005s	1968128	1837186	61504	353648	152768
flowsComputation	0.020s	10255620	1230082	532	153760	30752
resetFlows	0.009s	0	2	246016	0	0

Tabella 9: Metriche di prestazioni, tiled, no halo.



Si noti che non viene considerato il kernel resetFlows in quanto non effettua operazioni floating point di nessun tipo e, nel concreto, le tempistiche di esecuzioni sono identiche sull'arco di tutte le versioni. E' possibile notare dal grafico come tutti quanti i kernel siano molto fortemente "bandwidth bound". In altre parole, la larghezza di banda utilizzata dai kernel funge da collo di bottiglia per le prestazioni complessive dell'algoritmo. I kernel, semplicemente, non utilizzano una mole sufficiente di dati a raggiungere il picco teorico prestazionale della GPU. Come già menzionato più volte, il dominio di sciddicaT è di dimensioni relativamente ristrette, specialmente per una GPU potente come la GTX 980.

Kernel (100 step)	Tempo	FLOP	Load (G)	Store (G)	Load (S)	Store (S)
widthUpdate	0.005s	1968128	3301378	61504	392088	401016
flowsComputation	0.022s	10255620	1230082	532	215264	123008
resetFlows	0.010s	0	2	246016	0	0

Tabella 10: Metriche di prestazioni, tiled, halo.

Si noti inoltre come le varie implementazioni del kernel widthUpdate cadano tendenzialmente più a sinistra delle implementazioni del kernel flowsComputation. Ciò è dovuto al fatto che quest'ultimo è il kernel decisamente più intenso dal punto di vista computazionale, ed è quello che impiega più tempo ad essere eseguito. Si ponga particolare attenzione sul kernel widthUpdate in versione grid-stride e si noti come le operazioni floating-point effettuate in corrispondenza siano molto basse rispetto alla media; ciò coincide perfettamente con le osservazioni fatte nel Cap. 3 sulla lentezza dell'implementazione grid-stride rispetto alle altre. Si può dedurre che ad aver decrementato le prestazioni complessive di quell'implementazione sia stato proprio questo kernel.

5 Conclusioni

Nel presente documento è stato discusso il modello matematico sciddicaT, utilizzato per simulare lo scorrimento non inerziale di un fluido su una superficie. Una volta brevemente introdotti agli elementi principali del modello, si è passati alla discussione delle varie implementazioni parallele dell'algoritmo alla base del modello, tramite grafici e tabelle. In seguito, sono state confrontate le prestazioni delle varie implementazioni parallele, e da ciò è emerso che l'implementazione più performante è stata quella a tile senza halo, e quella meno performante è stata quella multi GPU. Si potrebbe sostenere che questa conclusione fosse prevedibile, in quanto come discusso più volte nel presente documento, il dominio di sciddicaT è semplicemente troppo ristretto per beneficiare veramente da una implementazione a multi GPU; in tal caso, gli svantaggi legati all'overhead di comunicazione superano nettamente i vantaggi, portando a un risultato globalmente peggiore. Un possibile miglioramento potrebbe essere apportato abilitando il supporto a CUDA sulla versione di OpenMPI installata su JPDM2. Ciò permetterebbe l'utilizzo degli stream CUDA e dunque di effettuare computazione e comunicazione in parallelo, attualmente impossibile. Nonostante ciò, tuttavia, i vantaggi ottenibili da una GTX 980 sono non nulli (i vantaggi prestazionali sono evidenti dato il tempo d'esecuzione originale di 67.5 secondi) ma comunque ben al di sotto delle capacità totali dell'hardware.

Riferimenti bibliografici

[Kin15] Donald Kinghorn. GTX 980 Ti Linux CUDA performance vs Titan X and GTX 980. https://www.pugetsystems.com/labs/hpc/GTX-980-Ti-Linux-CUDA-performance-vs-Titan-X-and-GTX-980-659/, 2015.