

Assignment 5 - SVILUPPO E
VALUTAZIONE DELLE
PERFORMANCE DI UN
SEMPLICE SIMULATORE DI
SCORRIMENTO
NON-INERZIALE DI UN FLUIDO
IN CUDA E MPI

Fazio Francesco Matricola: 227758

October 4, 2021

Contents

1	Introduction	7
2	Parallel Implementations	10
2.1	Straightforward parallelization	10
2.1.1	Check	10
2.2	Tiled parallelization with Halo Cells	11
2.2.1	Check	11
2.3	Tiled parallelization without Halo Cells	12
2.3.1	Check	13
2.4	MPI Parallelization	13
3	Computational Performance	14
3.1	Straightforward	14
3.1.1	CUDA Occupancy Calculator	15
3.1.2	sciddicaTWidthUpdate kernel	18
3.2	Tiling con celle halo	21
3.2.1	CUDA Occupancy Calculator	21
3.3	Tiling senza celle halo	28
3.3.1	Cuda Occupancy Calculator	30
4	Roofline Assessment	36
4.1	Applicazioni	39
4.2	Casi di studio	40
5	Conclusion	41

List of Figures

1	codice hash md5 in output del programma in versione Straightforward	10
2	codice hash md5 in output del programma in versione Halo	12
3	codice hash md5 in output del programma in versione Whithout Halo cells	13
4	caratteristiche gtx 980	14
5	configurazioni 1 per l'algoritmo straightforward . .	14
6	configurazioni 2 per l'algoritmo straightforward . .	15
7	configurazioni 3 per l'algoritmo straightforward . .	15
8	risorse utilizzate dal computation kernel	16
9	Threads per Block	16
10	Calcolo registri per thread	17
11	occupazione della GPU-SciddicaTFlowsComputation kernel	18
12	risorse utilizzate dall' sciddicaTWidthUpdate kernel	18
13	Threads per Block	19
14	Calcolo registri per thread	20
15	occupazione della GPU update kernel	20
16	varie configurazioni per l'algoritmo tiling con celle halo	21
17	varie configurazioni per l'algoritmo tiling con celle halo	21
18	risorse utilizzate dal computation kernel	22
19	Threads per Block Halo	23
20	Registri per Thread Halo	24
21	Occupazione della GPU	24
22	Calcolo Shared Memory	25

23	risorse utilizzate dall'SciddicaTWidthUpdate ker- nel	25
24	calcolo thread per blocco	26
25	calcolo registri per thread	27
26	occupazione della GPU	27
27	occupazione della GPU	28
28	configurazione per l'algoritmo tiling senza celle halo con tw pari a 8	29
29	configurazione per l'algoritmo tiling senza celle halo con tw pari a 4	29
30	configurazione per l'algoritmo tiling senza celle halo con tw pari a 2	30
31	risorse utilizzate dal sciddicaTFlowsComputationker- nel	31
32	calcolo thread per blocco sciddicaTFlowsCompu- tationkernel	31
33	calcolo registri per thread sciddicaTFlowsCompu- tationkernel	32
34	occupazione della GPU sciddicaTFlowsComputa- tionkernel	32
35	calcolo shared memory sciddicaTFlowsComputa- tionkernel	33
36	risorse utilizzate dal sciddicaTWidthUpdate kernel	33
37	calcolo thread per blocco sciddicaTWidthUpdate kernel	34
38	calcolo registri per thread sciddicaTWidthUpdate kernel	35
39	occupazione della GPU sciddicaTFlowsComputa- tionkernel	35
40	calcolo shared memory sciddicaTFlowsComputa- tionkernel	36

41	formula Flops	38
42	roofline model	38
43	algoritmo Straightforward	40
44	algoritmo tiling con celle halo	40
45	algoritmo tiling senza celle halo	40

Abstract

Gli automi cellulari sono un potente strumento per la modellazione naturale e sistemi artificiali, che possono essere descritti in termini di interazioni locali delle loro parti costitutive. Alcuni tipi di smottamenti, come i flussi di detriti/fango, soddisfano questi requisiti. Sono stati ottenuti adattando il Modello di Automi Cellulari denominato SCIDDICA, che è stato validato per frane molto veloci. SCIDDICA è stata applicata modificando il modello generale alle peculiarità della frana ticinese. I risultati ottenuti hanno identificato le condizioni di alto rischio che interessano le frazioni di Funes e Lamosano e mostrano che questo approccio agli automi cellulari può avere una vasta gamma di applicazioni per diversi tipi di flussi di fango/detriti. Nelle successive sezioni andremo a descrivere più nel dettaglio l'algoritmo sciddicaT nella sua versione seriale successivamente le 4 versioni in parallelo facendo una comparazione dei vari algoritmi sui punti forti/debole di ogni approccio e delle varie performance ottenute.

1 Introduction

In questa presentazione verranno discussi nel dettaglio lo sviluppo dei vari algoritmi in parallelo le comparazioni tra i vari algoritmi e una serie di valutazioni delle performance del modello **SciddicaT**, SciddicaT è un modello basato su un paradigma computazionale come quello degli automi cellulari, e su un algoritmo detto di "minimizzazione delle differenze". SciddicaT è un modello definito da una serie di fattori, $\langle \mathbf{R}, \mathbf{X}, \mathbf{S}, \mathbf{P}, \sigma \rangle$, dove:

- **R**: è il dominio bidimensionale del modello rappresentato da una matrice di celle;
- **X**: è un pattern geografico di celle vicine " von Neumann neighborhood", secondo il quale i vicini sono situati nelle quattro direzioni principali rispetto alla cella centrale.
- **S**: è lo stato attuale di ogni cella, viene diviso in tre sotto-stati:
 - **Sz**: è l'altitudine topografica di ogni cella
 - **Sh**: è la densità del fluido in ogni punto della matrice
 - **Sf**: rappresenta il valore della fuoriuscita del fluido dalla cella centrale alle quattro celle vicine. Infatti, ogni cella è rappresentata da quattro punti consecutivi;
- **P**: sono le limitazioni per cui il fluido può scorrere o meno attraverso determinati punti del modello bidimensionale. nello specifico, p viene rappresentato da P_{ee} e p_r , che indicano rispettivamente lo spessore minimo sotto il quale il fluido non può fuoriuscire a causa dell'effetto aderenza, e il fattore di smorzamento della fuoriuscita;

- **σ :** definisce le funzioni di transizione del fluido, quindi l'algoritmo vero e proprio. Ci sono diverse fasi:
 - **reset del flusso:** imposta a 0.0 la fuoriuscita dalla cella centrale ai vicini.
 - **computazione del flusso:** calcola il valore di fuoriuscita dalla cella centrale alle celle vicine grazie all'algoritmo di minimizzazione delle differenze, in modo da permettere una distribuzione più equa possibile del fluido sulle quattro celle adiacenti.
 - **update del flusso:** aggiorna la densità del fluido della cella centrale considerando il cambiamento dei vicini avvenuto precedentemente.

Le successive sezioni di questo report si basano sulla parallelizzazione in CUDA delle funzioni di transizione del fluido e di aggiornamento celle. Andremo a vedere nel particolare quattro versioni di parallelizzazione del modello SciddicaT:

- **StraightForward:** permette di parallelizzare in CUDA le funzioni del modello tramite una tecnica chiamata **Grid Stride Loops**.
- **Algoritmo tiling senza celle halo:** in questo algoritmo le celle halo vengono prese in modo diretto dalla memoria globale e non più memorizzate nella memoria condivisa.
- **Algoritmo tiling con celle halo:** Questo algoritmo dimensionale del modello in ulteriori sotto-matrici di dimensione fissata, e si serve della memoria condivisa (**shared memory** per salvare i dati su cui i thread andranno a lavorare, comprese le **halo cells**).

- **Algoritmo MultiGPU con MPI:** in questo algoritmo viene introdotta una libreria di parallelizzazione del lavoro lato CPU e non più GPU, che, come vedremo, dividerà la computazione dell'algoritmo sulle due GTX 980.

2 Parallel Implementations

In questo capitolo discuteremo nel dettaglio le varie implementazioni parallele del modello sciddicaT. ci soffermeremo su i singoli kernel e i vantaggi di ogni parallelizzazione sviluppata.

2.1 Straightforward parallelization

Questa tecnica si basa sulla semplice trasformazione delle funzioni parallele **pragma omp for** della libreria MPI in kernel CUDA. La peculiarità di questo algoritmo è sicuramente la semplicità di implementazione. Oltre lo step iniziale ovvero l'implementazione monolitica è stata aggiunta la possibilità per ogni thread di computare più celle, questa variante è detta **Grid Stride Loop**. In questa tecnica ogni kernel CUDA inizia con un doppio for i cui indici rappresentano la cella da computare, e di conseguenza, essendo un ciclo, ogni thread ha la possibilità di computare più celle.

2.1.1 Check

Per valutare la correttezza del programma abbiamo il codice hash MD5 dell'implementazione sequenziale di sciddicaT, che corrisponde a **8ed78fa13180c12b4d8aeec7ce6a362a** infatti come possiamo vedere dall'immagine qui riportata di seguito corrispondono.

```
user@ip002: progetto$ mvcc -o $ gpidstride.cu
user@ip002: progetto$ ./s ../data/tessina_header.txt ../data/tessina_dem.txt ../data/tessina_source.txt ./tessina_output_serial 4000 && md5sum ./tessina_output_serial && cat ../data/tessina_header.txt ./tessina_output_serial > ./tessina_output_serial_md5
elapsed time: 2.899000 [s]
releasing memory...
8ed78fa13180c12b4d8aeec7ce6a362a ./tessina_output_serial
user@ip002: progetto$
```

Figure 1: codice hash md5 in output del programma in versione Straightforward

2.2 Tiled parallelization with Halo Cells

Questa tecnica è diversa da quella precedente perchè introduce l'uso della memoria condivisa. La tecnica utilizzata fa uso del tiling ovvero si scompone la matrice che comprende tutto il dominio del problema in sotto matrici di dimensioni minore, la dimensioni per una maggiore semplicità corrisponde alla dimensione di ogni blocco. Queste sotto matrici vengono memorizzate nella memoria condivisa. Tuttavia, la dimensione della sottomatrice non corrisponde completamente alla dimensione della matrice calcolabile, perchè include le celle halo, quindi la sua dimensione è leggermente più grande. in questo algoritmo, la memoria condivisa viene utilizzata nei kernel che riguardano la computazione del fluido e l'update delle celle. Il vantaggio di questa tecnica è sicuramente la riduzione degli accessi alla memoria globale, perchè all'inizio di ognuno dei due kernel CUDA le celle su cui quel blocco di thread andrà a lavorare vengono memorizzate nella memoria condivisa, comprese le celle halo, quindi gli accessi alla global memory vengono limitati al reperimento di informazioni. L'implementazione più efficiente è stata ottenuta con una OTILEWIDTH pari a 4 e una MASK pari a 3.

2.2.1 Check

Per valutare la correttezza del programma abbiamo il codice hash MD5 dell'implementazione sequenziale di sciddicaT, che corrisponde a **8ed78fa13180c12b4d8aeec7ce6a362a** infatti come possiamo vedere dall'immagine qui riportata di seguito corrispondono.

```

user@903042 progetto$ nvcc -o cudatex sciddicaTiledHalo.cu
user@903042 progetto$ ./s ../data/tessina_header.txt ../data/tessina_dem.txt ../data/tessina_source.txt ./tessina_output_serial 4000 && md5sum ./tessina_output_serial && cat ../data/tessina_header.txt ./tessina_output_serial > ./tessi
na_output_serial.md5
Elapsed time: 2.908000 [s]
Releasing memory...
bu78f41310c12b4d9a9c7ceda362a ./tessina_output_serial

```

Figure 2: codice hash md5 in output del programma in versione Halo

2.3 Tiled parallelization without Halo Cells

L'algoritmo della tecnica di tiling senza halo cells è simile al precedente, perché in ogni caso fornisce l'uso della memoria condivisa, ma in modo diverso. Infatti, utilizzando la tecnologia di tiling Si prevede che la matrice contenente l'intero dominio del problema sia scomposta in altre sottomatrici di dimensioni comunque inferiori, in questo caso corrisponde anche a la dimensione di ogni blocco. Queste sottomatrici sono memorizzate nella memoria condivisa, Tuttavia, la sua dimensione non è uguale alla dimensione della matrice dell'algoritmo precedente, perché le halo cells non sono incluse in questo caso, quindi ha una dimensione esattamente uguale alla matrice computabile. Anche per questo algoritmo la memoria condivisa Viene utilizzata nei kernel che comportano la computazione del fluido e aggiornamenti delle celle. I principali vantaggi di questa tecnologia sono gli stessi della tecnologia precedentemente descritta, ovvero la riduzione degli accessi alla memoria globale, perchè all'inizio di ognuno dei due kernel CUDA le celle su cui quel blocco di thread andrà a lavorare vengono memorizzate nella memoria condivisa, con una sola differenza: le celle halo vengono reperite di volta in volta dalla memoria globale. L'implementazione più efficiente è stata ottenuta con una TILEWIDTH pari a 8.

2.3.1 Check

Per valutare la correttezza del programma abbiamo il codice hash MD5 dell'implementazione sequenziale di sciddicaT, che corrisponde a **8ed78fa13180c12b4d8aee7ce6a362a** infatti come possiamo vedere dall'immagine qui riportata di seguito corrispondono.

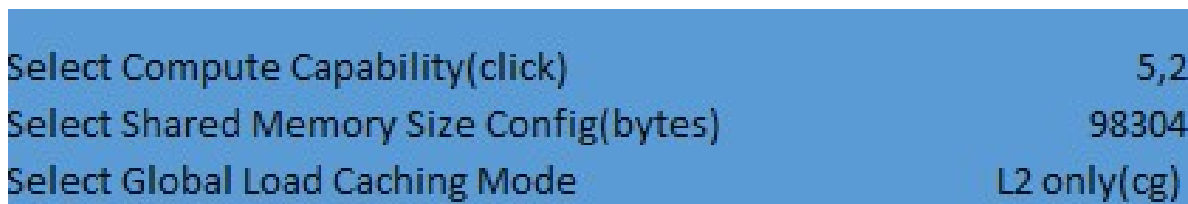
```
user@p002 progetto$ gcc -D S_Accidental_Memo.cu
user@p002 progetto$ ./s ../data/tessina_header.txt ../data/tessina_dem.txt ../data/tessina_source.txt ./tessina_output_serial 4000 && md5sum ./tessina_output_serial && cat ../data/tessina_header.txt ./tessina_output_serial > ./tessi
na_output_serial.qgis
Elapsed time: 3.143000 [s]
Releasing memory...
8ed78fa13180c12b4d8aee7ce6a362a ./tessina_output_serial
user@p002 progetto$
```

Figure 3: codice hash md5 in output del programma in versione Without Halo cells

2.4 MPI Parallelization

3 Computational Performance

In questa sezione discuteremo delle varie performance ottenute attraverso le varie configurazioni di griglie e thread per blocco nelle varie implementazione discusse nel capitolo precedente. in questo capitolo applicheremo il CUDA occupancy calculator per ogni algoritmo implementato per constatare se, dato il nostro algoritmo, il device su cui lo testiamo (GTX 980), è sfruttato al meglio.

A screenshot of the CUDA occupancy calculator interface. It shows three settings: 'Select Compute Capability(click)' set to '5,2', 'Select Shared Memory Size Config(bytes)' set to '98304', and 'Select Global Load Caching Mode' set to 'L2 only(cg)'.

Select Compute Capability(click)	5,2
Select Shared Memory Size Config(bytes)	98304
Select Global Load Caching Mode	L2 only(cg)

Figure 4: caratteristiche gtx 980

3.1 Straightforward

In questa sottosezione andremo a riportare i tempi di esecuzione dei vari kernel calcolati attraverso nvprof. Tutte le configurazioni di griglia di blocchi testati usano numero di thread per blocco $10*10$.

	Straightforward	
Dimensioni griglia	Kernel	Tempo di esecuzione(s)
((496/10)/4)*((610/10)/4)	sciddicaTFlowsComputation	1,42656
	sciddicaTWidthUpdate	1,06914
	SciddicaTResetFlows	0,69345

Figure 5: configurazioni 1 per l'algoritmo straightforward

	Straighforward	
Dimensioni griglia	Kernel	Tempo di esecuzione(s)
((496/10)/8)*((610/10)/8)	sciddicaTFlowsComputation	2,96658
	sciddicaTWidthUpdate	0,73855
	SciddicaTResetFlows	0,70017

Figure 6: configurazioni 2 per l'algoritmo straightforward

	Straighforward	
Dimensioni griglia	Kernel	Tempo di esecuzione(s)
((496/10)/8)*((610/10)/8)	sciddicaTFlowsComputation	1,27632
	sciddicaTWidthUpdate	1,085340
	SciddicaTResetFlows	0,65357

Figure 7: configurazioni 3 per l'algoritmo straightforward

dalle seguenti figure la 3 configurazione con numero di celle per thread uguale a 2, poichè questa configurazione è molto simile al kernel monolitico, siccome risulta molto prestante in quanto più thread lavorano contemporaneamente.

3.1.1 CUDA Occupancy Calculator

qui verranno discussi i calcoli effettuati tramite l'uso del CUDA occupancy calculator, in modo da verificare se con la configurazione usata per l'algoritmo Straightforward gli SM¹ vengono sfruttati a pieno.

SciddicaTFlowsComputation kernel La figura 9 mostra una possibile impostazione della grandezza dei blocchi in modo da raggiungere le prestazioni migliori possibili.

¹symmetric multiprocessor

Threads per block

Registers per thread

Shared memory per block

bytes

Figure 8: risorse utilizzate dal computation kernel

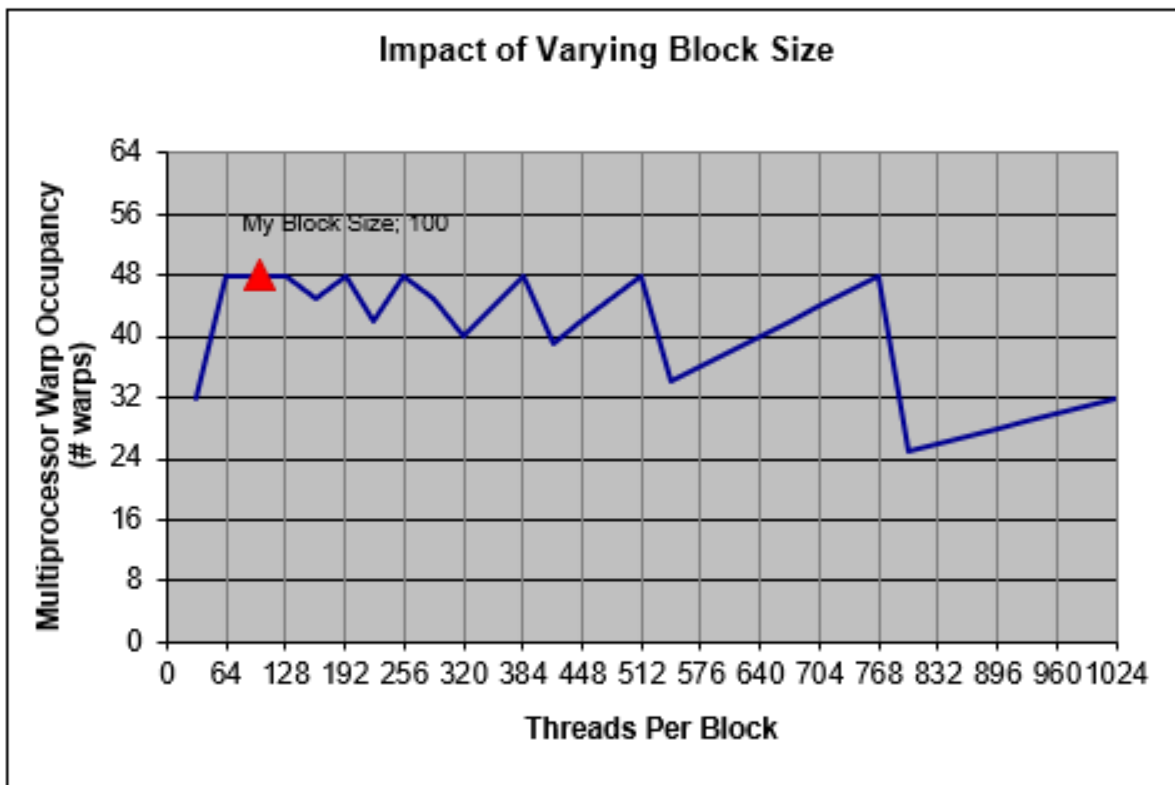


Figure 9: Threads per Block

La figura 10 invece rappresenta un metodo di ottimizzazione riguardo l'uso dei registri che ogni thread utilizza per memorizzare le variabili.

Impact of Varying Register Count Per Thread

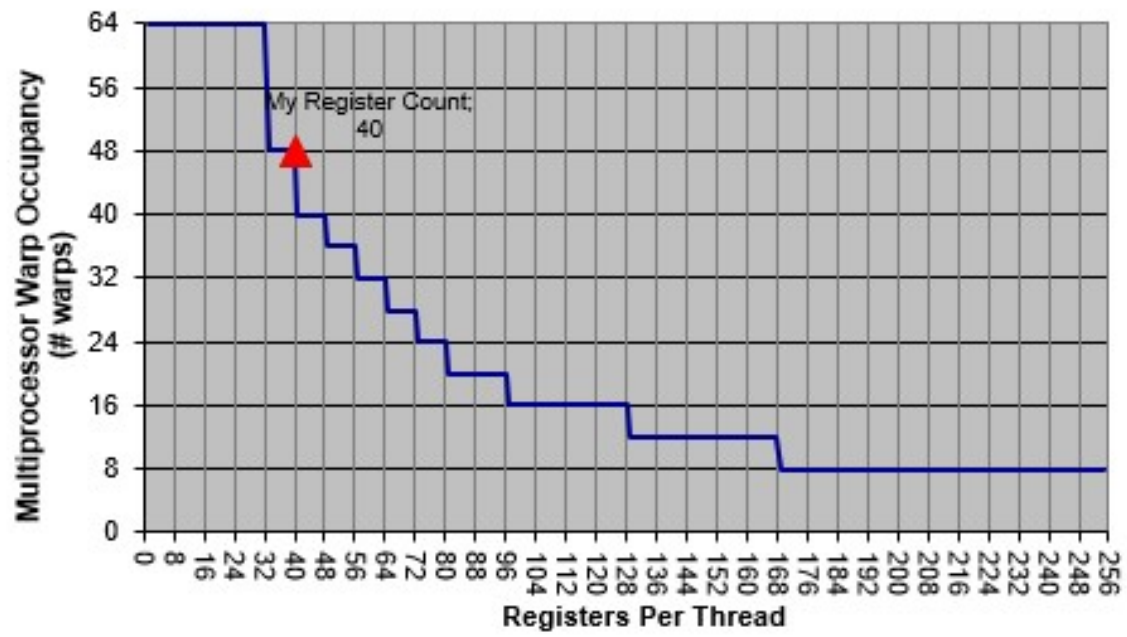


Figure 10: Calcolo registri per thread

Da questa figura possiamo notare che ogni SM è sfruttato al 75 e fa uso di esattamente 12 blocchi.

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	12
Occupancy of each Multiprocessor	75%

Figure 11: occupazione della GPU-SciddicaTFlowsComputation kernel

3.1.2 sciddicaTWidthUpdate kernel

Threads per block
100

Registers per thread
31

Shared memory per block
0 bytes

Figure 12: risorse utilizzate dall' sciddicaTWidthUpdate kernel

La figura sottostante mostra una possibilità di impostazione della grandezza dei blocchi in modo da raggiungere le prestazioni migliori possibili.

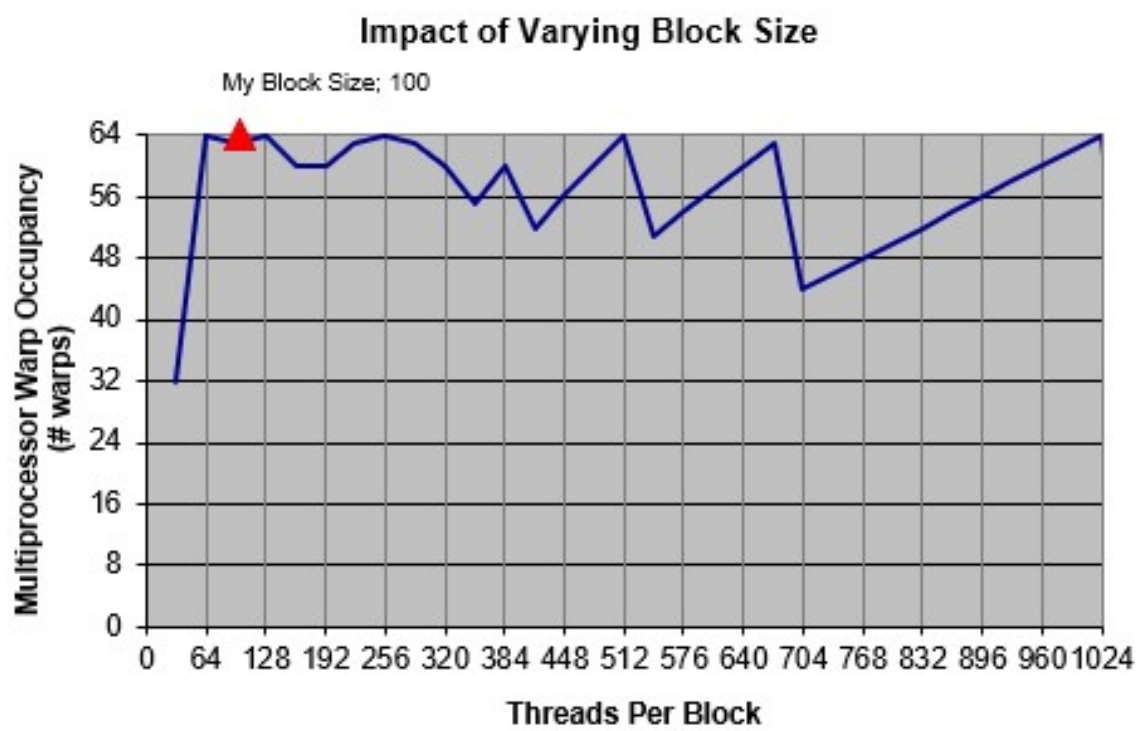


Figure 13: Threads per Block

La figura 14 rappresenta un metodo di ottimizzazione riguardo l'uso dei registri che ogni thread utilizza per memorizzare le variabili. Possiamo notare che l'utilizzo dei nostri 31 registri sfrutta il massimo ottenibile da questa configurazione.

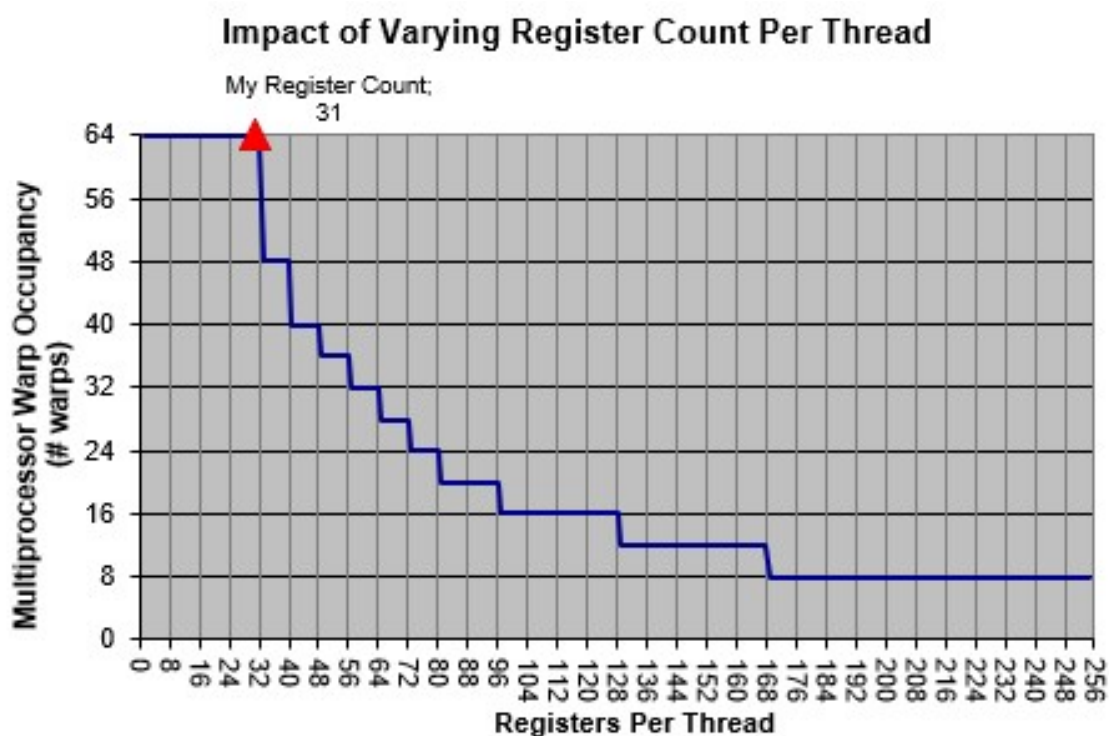


Figure 14: Calcolo registri per thread

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	100%

Figure 15: occupazione della GPU update kernel

possiamo notare nel update kernel che ogni SM è sfruttato al massimo.

3.2 Tiling con celle halo

In questa sezione andremo a riportare in breve i tempi di esecuzione dei vari kernel calcolati attraverso nvprof.

Dimensione Griglia	kernel	tempo di esecuzione
$(496/4)*(610/4)$	SciddicaTResetFlows	0,60143
$(496/4)*(610/4)$	SciddicaTFlowsComputation	2,2136
$(496/4)*(610/4)$	SciddicaTWidthUpdate	0,9725

Figure 16: varie configurazioni per l'algoritmo tiling con celle halo

Dimensione Griglia	kernel	tempo di esecuzione
$(496/5)*(610/5)$	SciddicaTResetFlows	0,58485
$(496/5)*(610/5)$	SciddicaTFlowsComputation	2,371716
$(496/5)*(610/5)$	SciddicaTWidthUpdate	0,99404

Figure 17: varie configurazioni per l'algoritmo tiling con celle halo

dalle figure 16-17 possiamo notare che l'esecuzione che ha avuto più successo è quella con OTileWidth uguale a 4 e MASK uguale a 3 anche se la differenza è minima.

3.2.1 CUDA Occupancy Calculator

qui verranno discussi i calcoli effettuati tramite l'uso del CUDA occupancy calculator, in modo da verificare se con la configurazione usata per l'algoritmo Straightforward gli SM vengono sfruttati a pieno.

sciddicaTFlowsComputation kernel Poichè la computazione viene effettuata tramite l'uso di matrici memorizzate nella shared memory in cui vengono riportate anche le celle halo, loro assumono una dimensione pari a BLOCKWIDTH pari a OTILEWIDTH+MASK 1 perciò 6. quindi i thread per blocco sono 36 e non 16 come ci aspettavamo da un OTILEWIDTH pari a 4. La figura sottostante mostra una possibilità di impostazione della grandezza dei blocchi in modo da raggiungere le prestazioni migliori possibili.

2.) Enter your resource usage:	
Threads Per Block	36
Registers Per Thread	32
User Shared Memory Per Block (bytes)	576

Figure 18: risorse utilizzate dal computation kernel

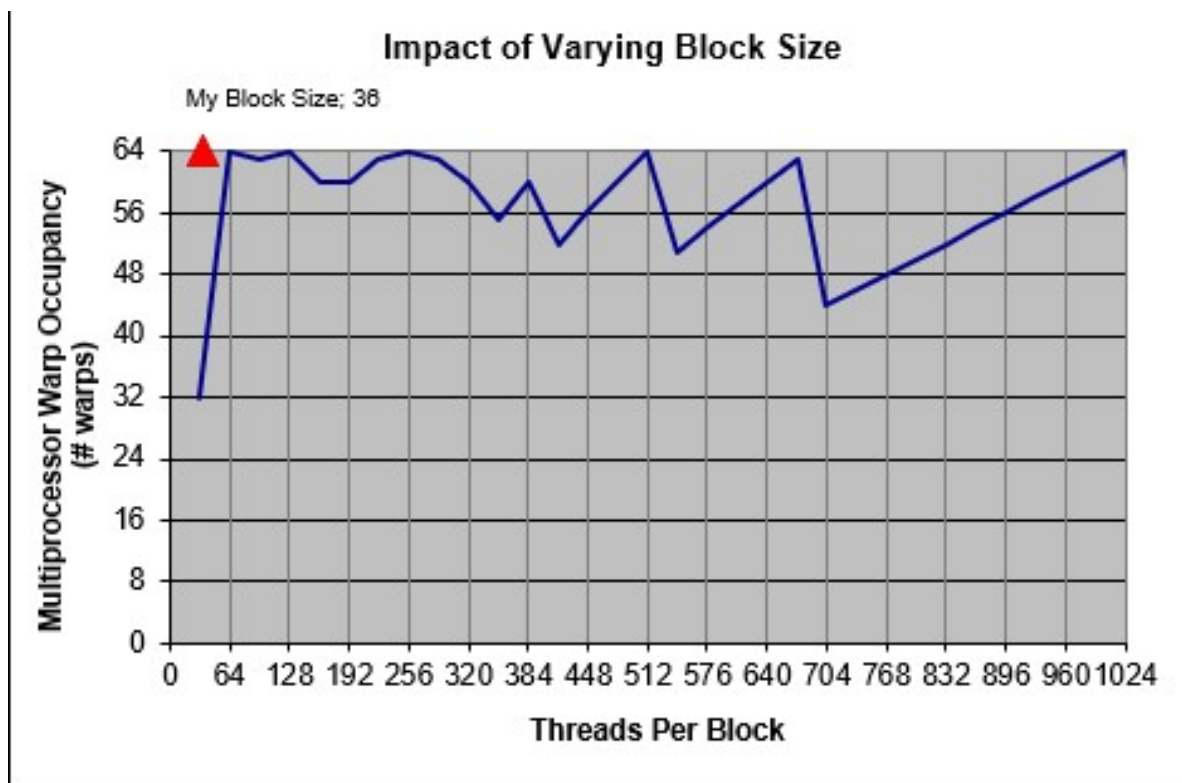


Figure 19: Threads per Block Halo

La figura sottostante mostra invece un metodo di ottimizzazione riguardo l'uso dei registri che ogni thread utilizza per memorizzare le variabili.

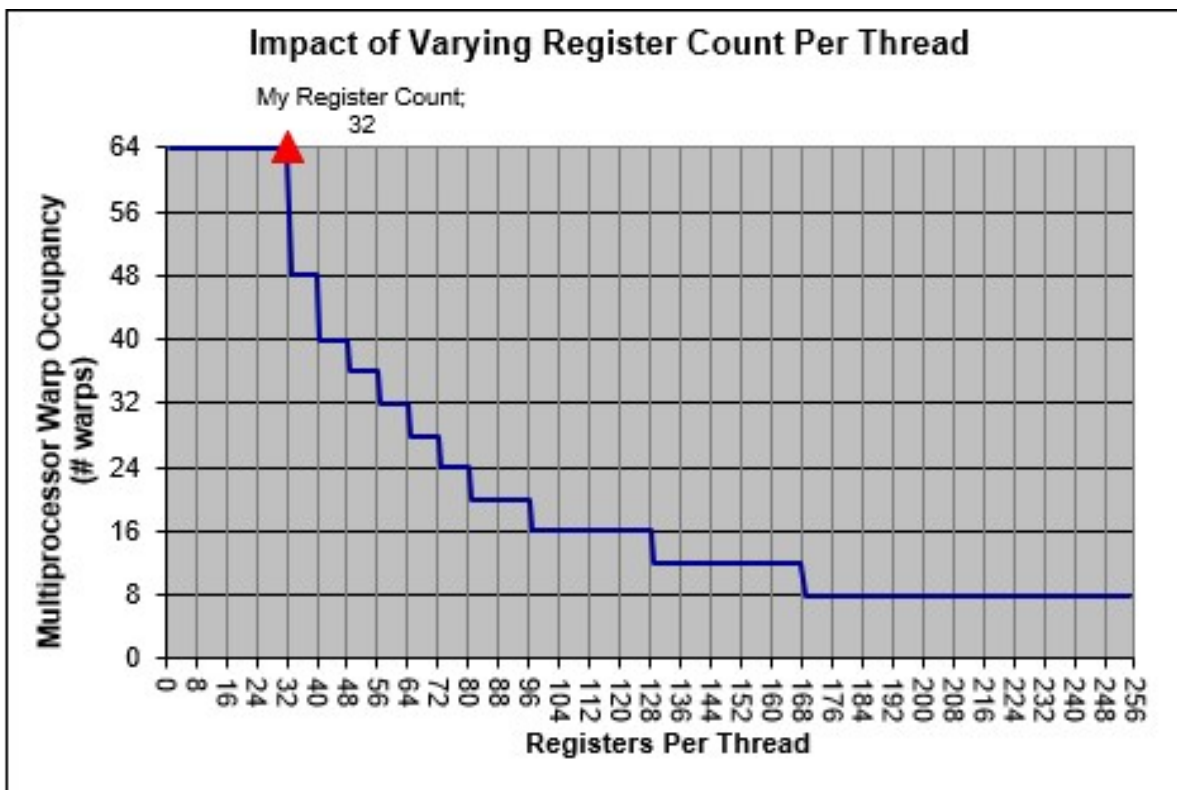


Figure 20: Registri per Thread Halo

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	32
Occupancy of each Multiprocessor	100%

Figure 21: Occupazione della GPU

In questa sezione andremo a parlare di Shared Memory, poichè il kernel ne fa uso. Notiamo che nella figura 22 il nostro utilizzo di shared memory è sufficiente a sfruttare al massimo il dispositivo

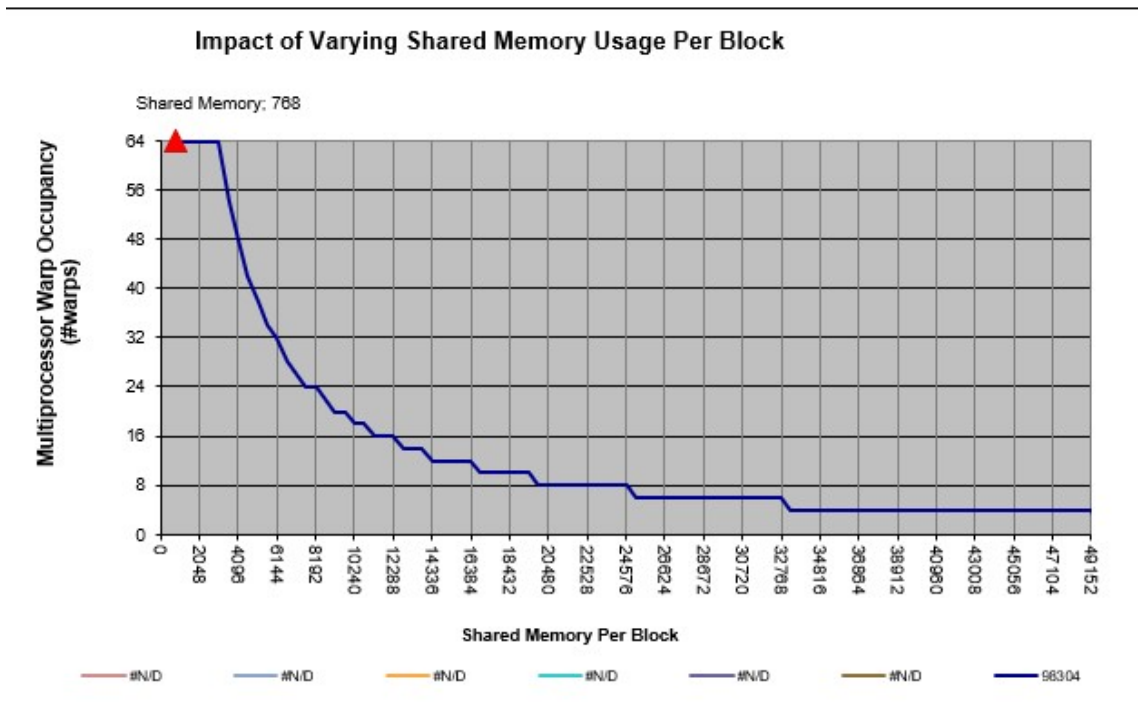


Figure 22: Calcolo Shared Memory

SciddicaTWidthUpdate kernel La figura 24 mostra varie possibilità di configurazione per raggiungere le prestazioni migliori possibili, e, in questo caso, riusciamo a raggiungere le migliori prestazioni.

2.) Enter your resource usage:	
Threads Per Block	36
Registers Per Thread	24
User Shared Memory Per Block (bytes)	1250

Figure 23: risorse utilizzate dall'SciddicaTWidthUpdate kernel

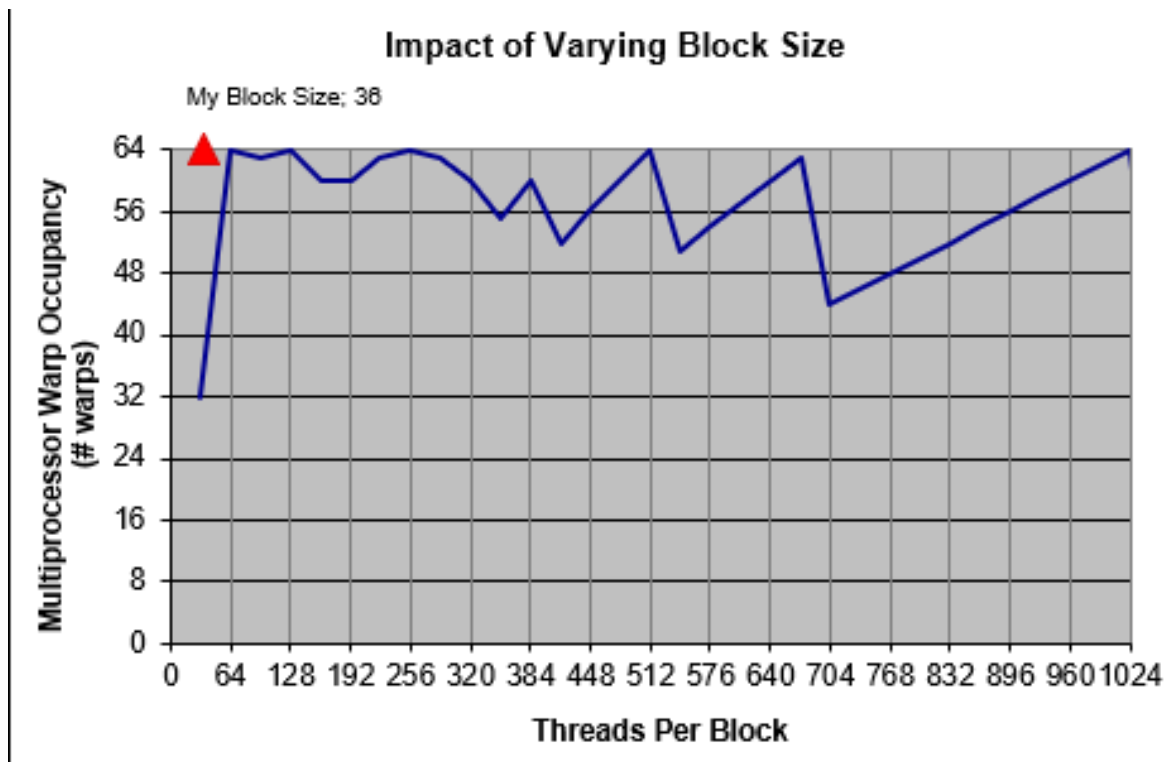


Figure 24: calcolo thread per blocco

La figura sottostante mostra invece un metodo di ottimizzazione riguardo l'uso dei registri che ogni thread utilizza per memorizzare le variabili.

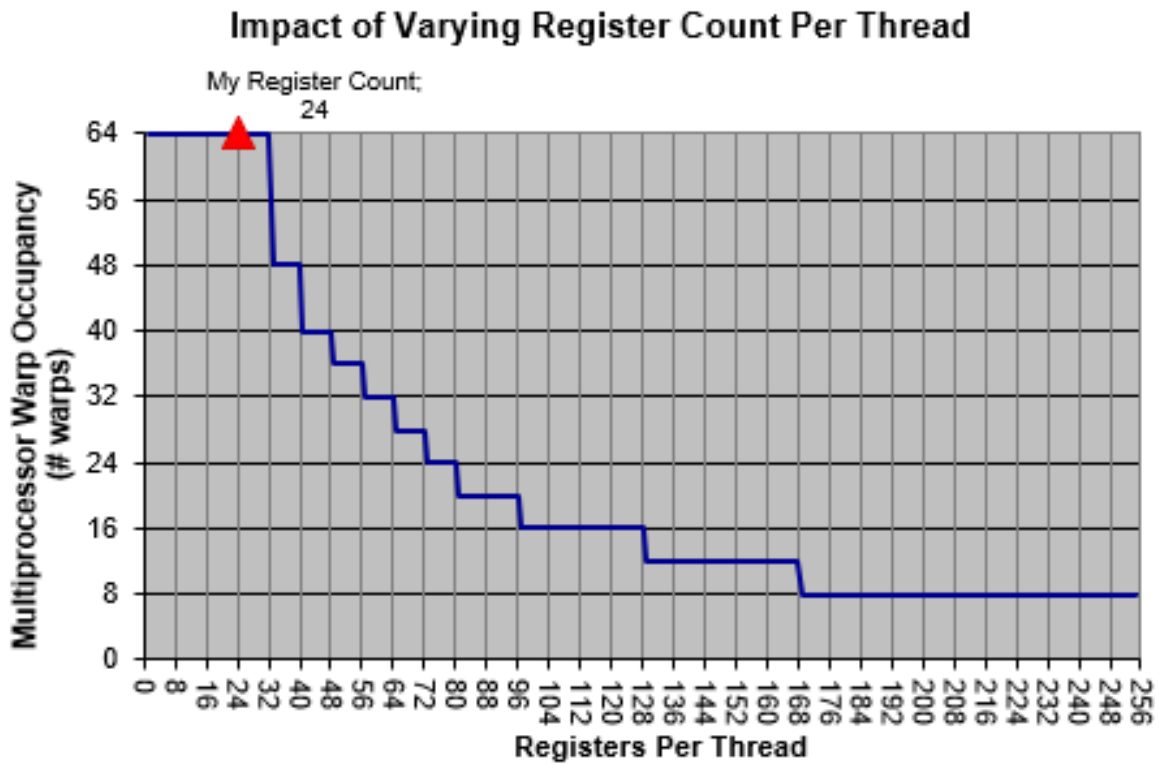


Figure 25: calcolo registri per thread

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	32
Occupancy of each Multiprocessor	100%

Figure 26: occupazione della GPU

per quanto riguarda l'update kernel, possiamo notare che ogni SM è sfruttato al 100.

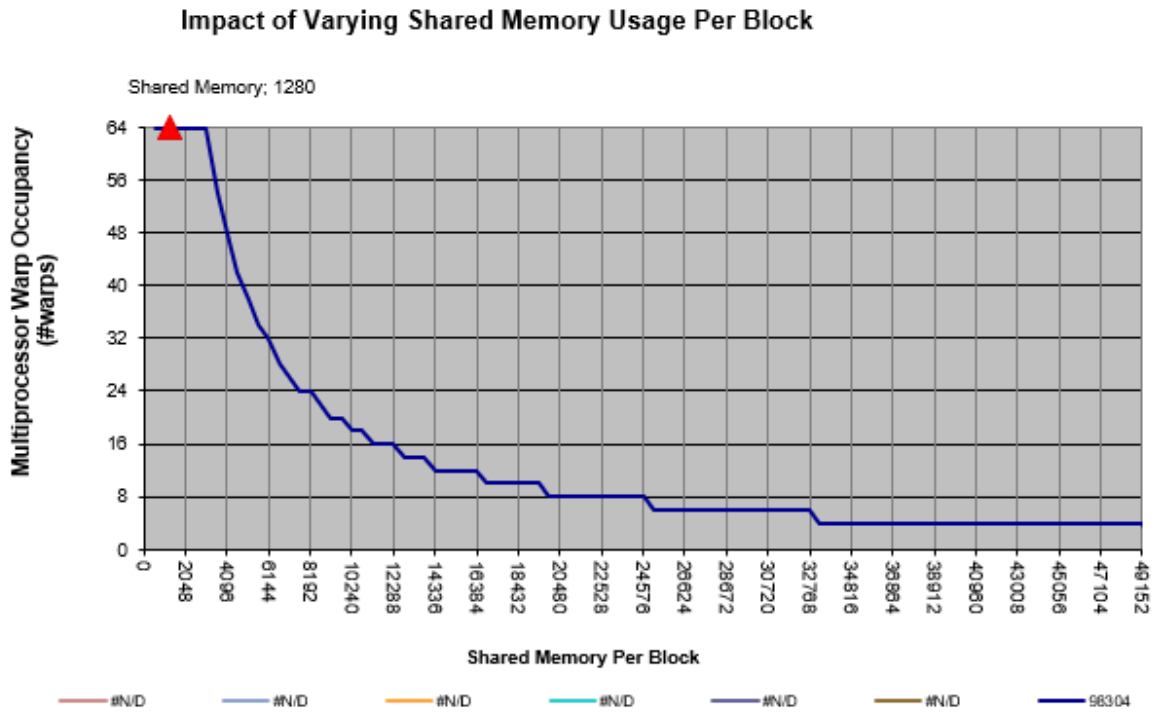


Figure 27: occupazione della GPU

Come dimostra la figura in alto , il nostro utilizzo di shared memory è sufficiente a sfruttare al massimo ogni multiprocessor.

3.3 Tiling senza celle halo

in basso i tempi di esecuzione dei vari kernel calcolati tramite nvprof.

	Tiling senza Halo Cells	
DimGrid	kernel	Time(s)
$(496/8)*(610/8)$	SciddicaTResetFlows	0,62246
$(496/8)*(610/8)$	SciddicaTFlowsComp	1,20931
$(496/8)*(610/8)$	SciddicaTWidthUpdate	0,73241

Figure 28: configurazione per l'algoritmo tiling senza celle halo con tw pari a 8

	Tiling senza Halo Cells	
DimGrid	kernel	Time(s)
$(496/4)*(610/4)$	SciddicaTResetFlows	0,63758
$(496/4)*(610/4)$	SciddicaTFlowsComp	2,11396
$(496/4)*(610/4)$	SciddicaTWidthUpdate	1,07783

Figure 29: configurazione per l'algoritmo tiling senza celle halo con tw pari a 4

	Tiling senza Halo Cells	
DimGrid	kernel	Time(s)
$(496/2)*(610/2)$	SciddicaTResetFlows	0,54132
$(496/2)*(610/2)$	SciddicaTFlowsComp	7,50385
$(496/2)*(610/2)$	SciddicaTWidthUpdate	2,89063

Figure 30: configurazione per l'algoritmo tiling senza celle halo con tw pari a 2

Dalle seguenti figure possiamo notare che in termini di tempistica ha avuto più successo la configurazione in cui TileWidth è uguale a 8. questo è logico visto che avendo matrici più grandi che vengono memorizzate nella memoria condivisa , il numero di blocchi effettivamente utilizzati è minore, e i kernel lavorano con più efficienza. Dalle figure in alto notiamo che più aumentiamo le dimensioni dei blocchi, più l'algoritmo diventa efficiente, e vediamo che vale per ogni kernel.

3.3.1 Cuda Occupancy Calculator

qui verranno discussi i calcoli effettuati tramite l'uso del CUDA occupancy calculator, in modo da verificare se con la configurazione usata per l'algoritmo Straightforward gli SM vengono sfruttati a pieno.

sciddicaTFlowsComputationkernel Dalla figura 31 notiamo che poichè la computazione viene effettuata tramite l'uso di matrici salvate nella memoria condivisa in cui non vengono riportate le celle halo, queste hanno una dimensione pari a TileWidth.

2.) Enter your resource usage:	
Threads Per Block	64
Registers Per Thread	40
User Shared Memory Per Block (bytes)	1000

Figure 31: risorse utilizzate dal sciddicaTFlowsComputationkernel

La figura 32 ci mostra come settare òa grandezza deo bòpcchi in modo da raggiungere le prestazioni più efficienti.

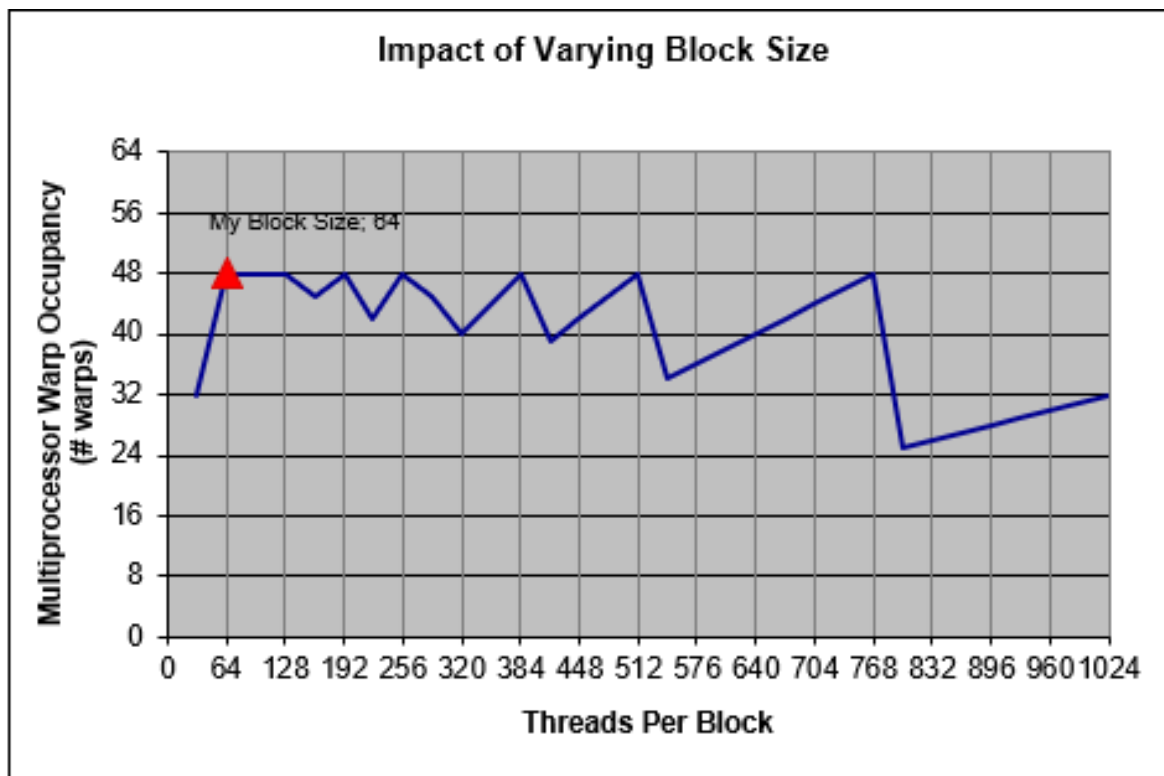


Figure 32: calcolo thread per blocco sciddicaTFlowsComputationkernel

La figura 33 mostra invece un metodo di ottimizzazione riguardo l'uso dei registri che ogni thread utilizza per memorizzare le variabili.

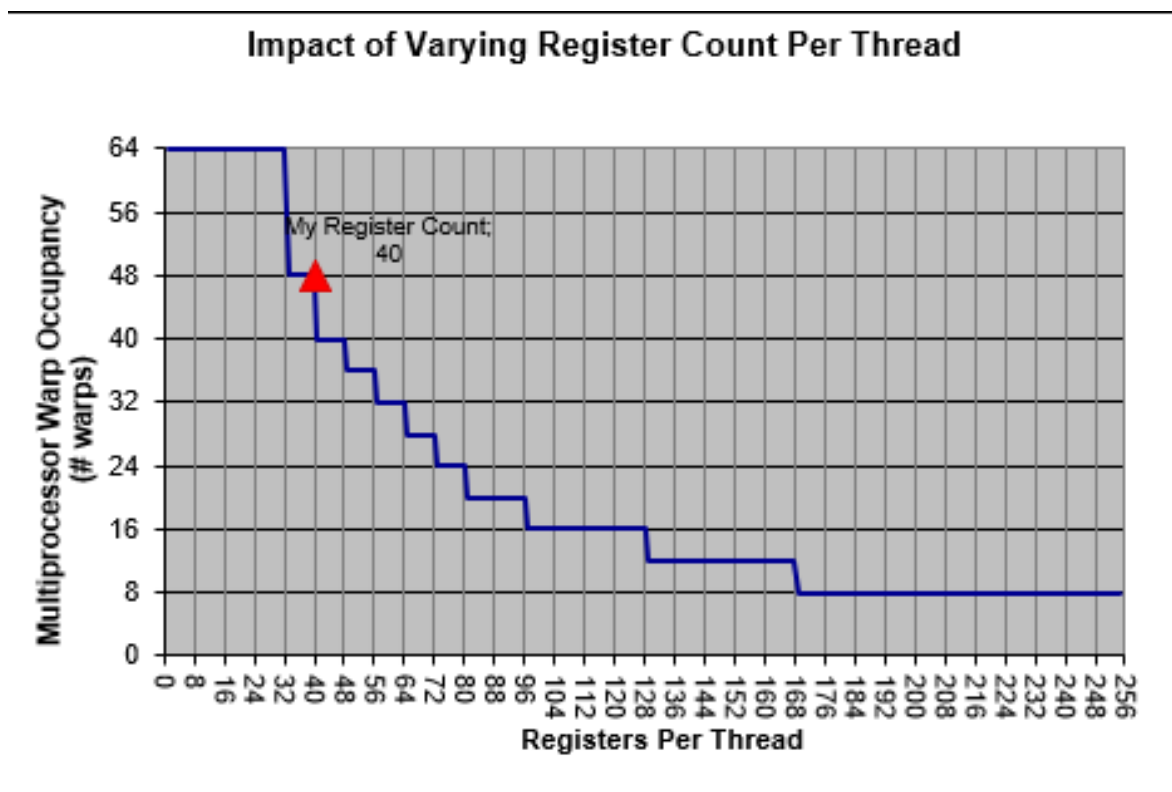
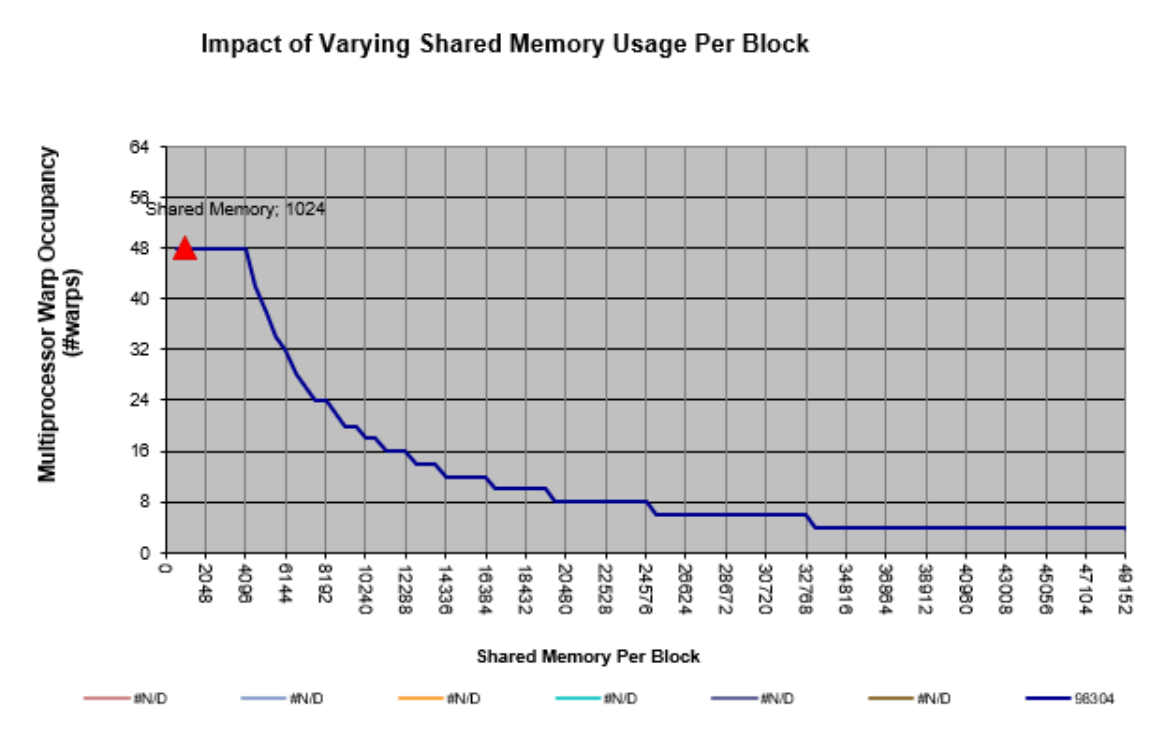


Figure 33: calcolo registri per thread sciddicaTFlowsComputationkernel

Possiamo notare che l'utilizzo dei nostri 40 registri non porta al massimo dei risultati, ma è comunque accettabile. possiamo notare che ogni SM è sfruttato al 75.

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	24
Occupancy of each Multiprocessor	75%

Figure 34: occupazione della GPU sciddicaTFlowsComputationkernel



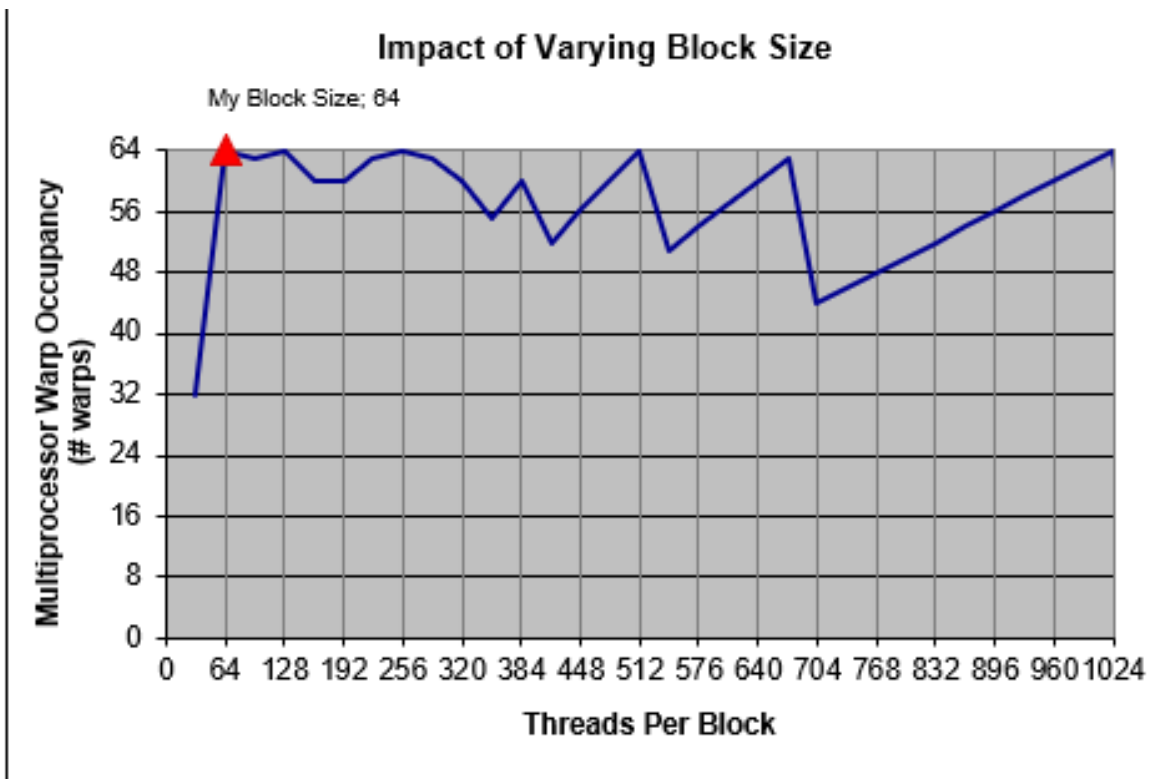


Figure 37: calcolo thread per blocco sciddicaTWidthUpdate kernel

La figura 37 mostra invece un metodo di ottimizzazione riguardo l'uso dei registri che ogni thread utilizza per memorizzare le variabili.

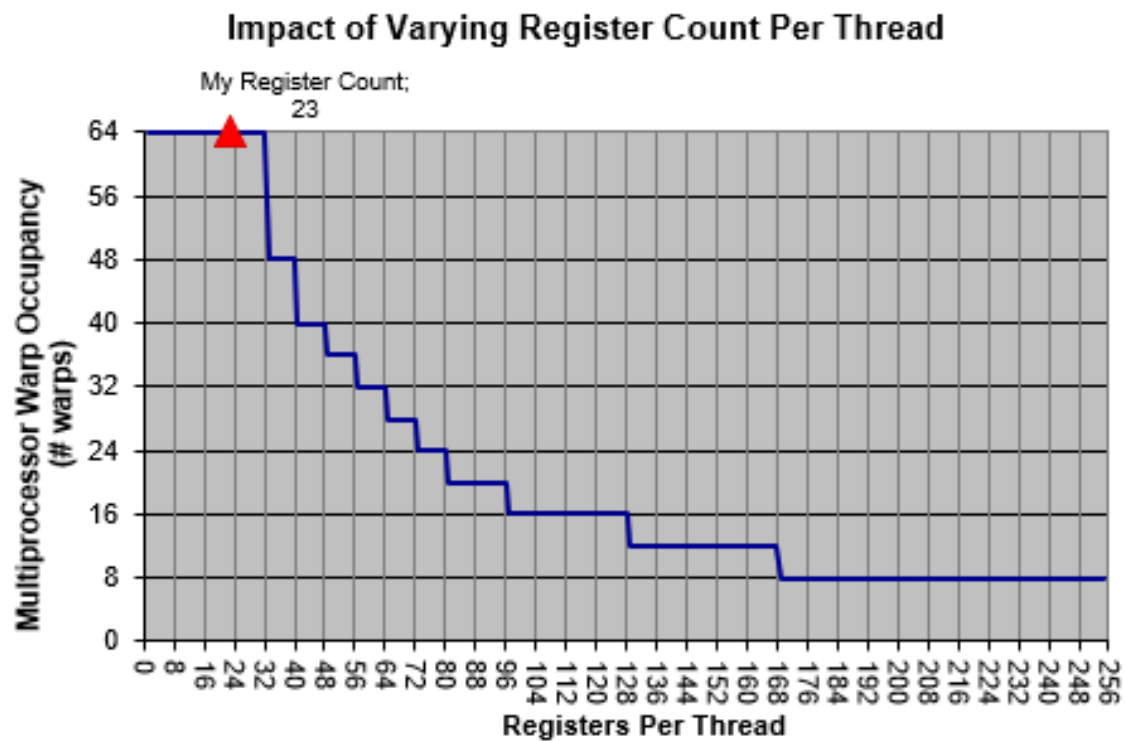


Figure 38: calcolo registri per thread sciddicaTWidthUpdate kernel

Possiamo notare che l'utilizzo dei nostri 23 registri è perfetto per le massime prestazioni. possiamo notare che ogni SM è sfruttato al 100.

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	32
Occupancy of each Multiprocessor	100%

Figure 39: occupazione della GPU sciddicaTFlowsComputationkernel

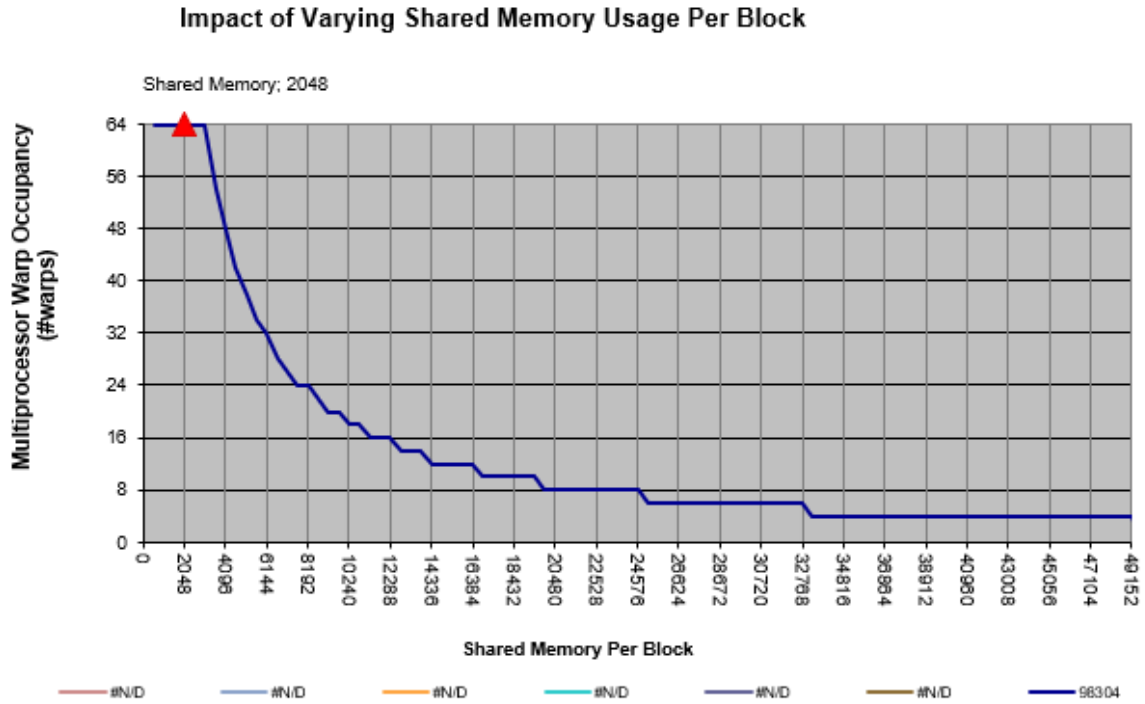


Figure 40: calcolo shared memory sciddicaTFlowsComputationkernel

anche se si vede un utilizzo più intenso della memoria condivisa in quanto la dimensione della TileWidth è raddoppiata, sfrutta comunque a pieno la GPU.

4 Roofline Assessment

Il modello Roofline (in italiano letteralmente "linea del tetto") è un modello di performance visuale che consente in maniera intuitiva di stimare le performance di un dato kernel computazionale o di una applicazione che esegue su architetture di calcolo di tipo multi-core, many-core o su acceleratori, mostrando graficamente le limitazioni inerenti all'hardware usato e le potenziali ottimizzazioni da poter applicare, nonché la priorità con cui esse necessitano di essere applicate. Facendo convergere aspetti riguardanti

la località dei dati, la banda, e paradigmi di parallelizzazione in un'unica raffigurazione, il modello di fatto costituisce una valida alternativa al semplice utilizzo della percentuale del picco di performance per valutare la qualità delle prestazioni ottenute, dal momento che è in grado di modellare analiticamente sia dettagli implementativi che limitazioni alle prestazioni dovute a caratteristiche intrinseche dell'architettura considerata. Il modello Roofline nella sua versione base può essere visualizzato graficando le prestazioni relative all'esecuzione di istruzioni a virgola mobile (FLOPS) come funzione del picco di performance, il picco di banda della macchina, e l'intensità operativa. La curva risultante, chiamata appunto Roofline, costituisce un limite superiore alle prestazioni ottenibili, al di sotto della quale esistono le effettive performance per il dato kernel o applicazione. Suddetta curva include due limiti massimi specifici della piattaforma: un limite derivato dalla banda di memoria e un limite derivato dal picco di performance dell'unità di computazione. Il picco di computazione è espresso in FLOPS e per la GTX 980 che è stata utilizzata per questo esperimento, il valore è di circa 4,612 GFLOP/s, mentre la bandwidth massima è di 224.32 GB/s. Infatti, il picco di computazione rappresenta il numero massimo di operazioni floating point che possono essere eseguite in un secondo, mentre la bandwidth rappresenta la quantità massima di dati che può essere processata in un secondo. Il grafico del roofline model rappresenta, sull'asse X, l'operational intensity (numero di floating point operations per unità di dato), e sull'asse Y, il numero di operazioni al secondo di un certo kernel. Numero che è calcolato come segue:

Arithmetic Intensity, Algorithm dependent Bandwidth, HW dependent

$$FLOPS = \frac{\#FLOP}{\text{time (Sec)}} = \frac{\#FLOP}{\text{Byte}} \times \frac{\text{Byte}}{\text{Sec}}$$

$$FLOPS = AI \times BW$$

Figure 41: formula Flops

considerando il logaritmo, otteniamo: $\log(FLOPS) = \log(AI) + \log(BW)$, che rapportato all'equazione di una retta ($y = mx + c$), si otterrebbe: $y = \log(FLOPS)$, $m = 1$, $x = \log(AI)$. Di seguito viene riportato il grafico generale del roofline model:

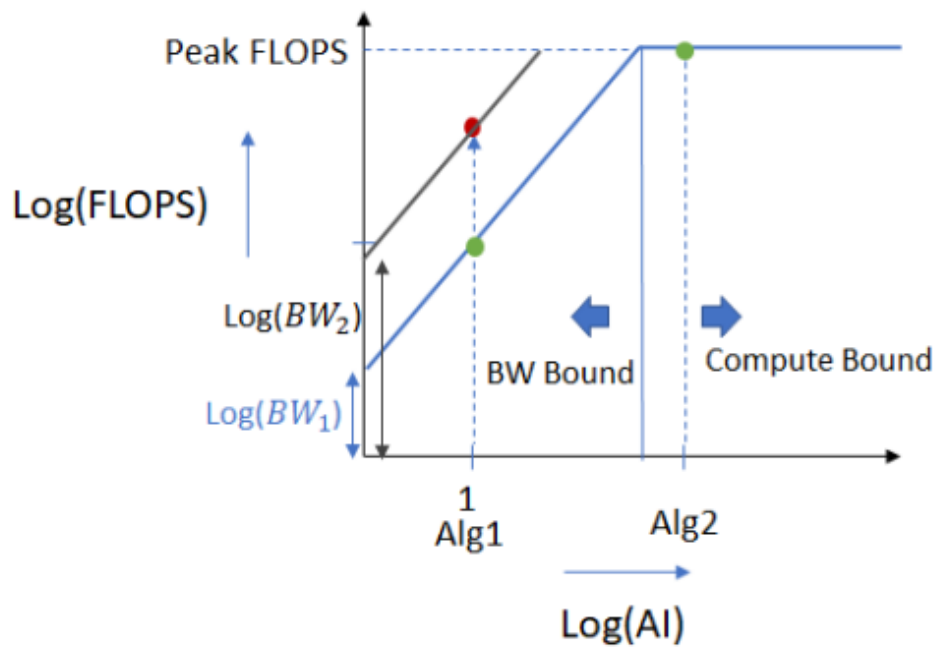


Figure 42: roofline model

Come possiamo notare, il grafico è diviso in due regioni: la prima, quella di sinistra, contiene tutti gli algoritmi "BW bound",

che non hanno raggiunto il massimo della computazione in termini di FLOPS, in quanto la memory bandwidth non viene sfruttata al suo massimo, mentre la seconda, quella di destra, contiene gli algoritmi "Compute bound" che sfruttano al massimo il device. Per migliorare gli algoritmi che stanno nella zona di sinistra, è possibile ottimizzarli preferendo il cosiddetto "riuso delle risorse", oppure migliorando l'hardware in termini di bandwidth.

4.1 Applicazioni

Il caso generale riguardante tutti gli algoritmi testati vede un grafico del roofline model che raggiunge un picco di computazione pari a 4.612 GFLOPS/s, che sotto logaritmo sono poco più di 3,5. Per quanto riguarda la bandwidth, il picco della global memory bandwidth raggiunge un massimo di 224 GB/s, mentre quello della shared memory raggiunge circa 2121 GB/s (calcolato con `gpumembench`). Per quanto riguarda i test che sono stati effettuati, nei vari grafici che vengono visualizzati potremo notare dei punti verdi, che indicano i vari kernel testati. Quindi, tramite il comando `nvprof ./alg.out` profiliamo il tempo di computazione dei kernel dell'algoritmo in questione, mentre con il comando `srunkvprof -metrics flopcountsp -metrics flopcountdp -metrics gld-transactions -metrics gsttransactions -metrics sharedloadtransactions -metrics sharedstoretransactions ./alg.out` profiliamo il numero di operazioni floating point e il numero di operazioni load e store per ogni kernel. Il valore è calcolato come segue:

- asse x: $(\text{floating point operations} / (\text{load} + \text{store}) * 8)$
- asse y: $\log(\text{floating point operations} / \text{tempo di esecuzione kernel})$

4.2 Casi di studio

Per quanto riguarda le prime tre implementazioni (a singola GPU) i valori per calcolare i due punti sull'asse x e sull'asse y sono riportati nelle tabelle sottostanti:

		StraightForward		
Kernel	Floating point op	Tempo di esec (s)	Load Operations	Store Operations
sciddicaTFlowsComp	11539176	1,32402	3147602	1554
SciddicaTWidthUp	2832842	0,89906	2832842	302507
SciddicaTResetFlows	0	0,62955	2	1209923

Figure 43: algoritmo Straightforward

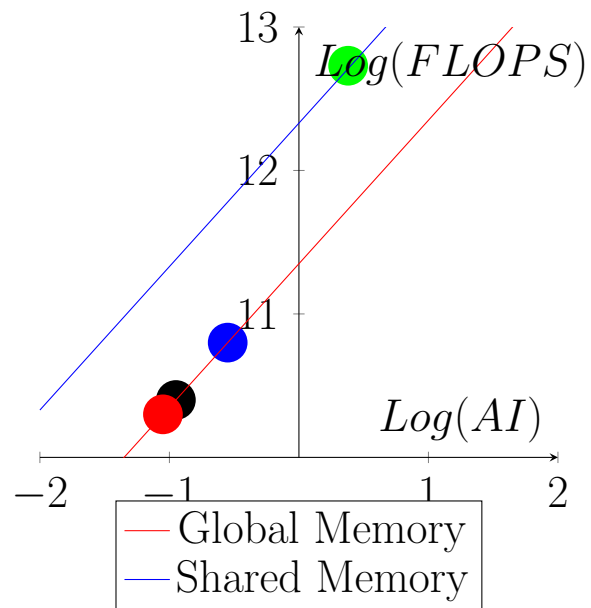
		Tiling With Halo Cells		
Kernel	Floating point op	Tempo di esec (s)	Load Operations	Store Operations
sciddicaTFlowsComp	13051653	2,22671	2649942	225947
SciddicaTWidthUp	2375520	0,96627	6605218	969685
SciddicaTResetFlows	0	0,60556	2	1105235

Figure 44: algoritmo tiling con celle halo

		Tiling Without Halo		
Kernel	Floating point op	Tempo di esec (s)	Load Operations	Store Operations
sciddicaTFlowsComp	12994718	1,21423	4219626	149888
SciddicaTWidthUp	2364872	0,73598	4480128	889017
SciddicaTResetFlows	0	0,62497	2	1182436

Figure 45: algoritmo tiling senza celle halo

Il grafico sul roofline plot, inoltre, è stato disegnato confrontando i tre kernel sulla computazione della fuoriuscita del fluido e tre kernel sull'update delle celle, in quanto il numero di floating point operations negli altri kernel risultava essere 0.



5 Conclusion

References