Programmazione su architetture parallele

2SAT

Galvan Matteo

19 aprile 2023

Sommario

L'obiettivo del progetto è di trovare le prime k soluzioni al problema 2SAT che viene fornito in input.

INTRODUZIONE

Il 2-satisfability, chiamato anche 2SAT, è un problema in cui l'obiettivo è trovare una soluzione date delle clausole booleane. Ogni clausola è formata da 2 letterali.

Sia il numero di letterali che il numero di clausole è conosciuto: se vengono creati 10 letterali, oltre a quelli da 1 a 10, esisteranno anche quelli da -1 a -10. La particolarità e che se 1 è vero allora -1 è falso. Questi letterali li chiamerò nodi fratelli.

Ogni clausola contiene due letterali in OR e ogni clausola è in AND con le altre. Nella risoluzione del problema se una delle clausole è falsa, rende falsa tutta la soluzione che si sta andando a verificare.

Per definizione di come è costruito l'operatore booleano OR, una clausola diventa falsa se entrambi i letterali all'interno sono falsi.

I letterali positivi e negativi possono essere pensati come dei nodi in un grafo e le clausole come l'arco che collega due nodi. Due nodi collegati non possono essere entrambi Falsi perché questo renderebbe la clausola falsa e quindi falso tutto il problema. Il 2SAT, a differenza del 3SAT, è risolvibile in tempo polinomiale.

1.1 Vincoli

Le istanze 2SAT sono espresse in un formato chiamato DIMACS. Il file contenente le clausole contiene una prima riga con scritto "p cnf n m" in cui n rappresenta il numero di letterali e m il numero di clausole che formano il problema.

Ogni riga del file di vincoli è composta da 3 numeri: il primo numero rappresenta il primo letterale, il secondo numero il secondo letterale e il terzo numero è sempre uno 0, che significa la fine della riga.

1.2 Soluzione Seriale

Nel caso di una risoluzione seriale, tramite la presenza di un grafo, è possibile trovare una soluzione andando a visitare il grafo, dando dei valori ai nodi e verificando che non esistano due nodi entrambi Falsi. In tutto questo deve sempre essere verificato il fatto che due nodi fratelli abbiano valore opposto.

La soluzione seriale verrà utilizzata nel capitolo 4.5 per confrontare i tempi di esecuzione tra di essa e la soluzione parallela.

2. OPTIMIZATION

Per sviluppare la soluzione parallela, con l'obiettivo di risolvere il 2SAT in parallelo ottimizzando le risorse, ho implementato diverse ottimizzazioni.

2.1 Controllo di correttezza

Per migliorare la velocità di esecuzione è possibile eseguire un controllo in più. Nel momento in cui andiamo a visitare il grafo, e troviamo un vicino che è già stato visitato, se volevamo dargli True ed era già False, interrompiamo. Infatti il 2SAT esige che i nodi siano veri o falsi e se un nodo era già vero, quella che stiamo eseguendo non è una soluzione.

2.2 Regola di risoluzione

La regola di risoluzione permette di andare a creare nuove clausole che non esistevano in precedenza. L'obiettivo principale nell'utilizzo di questa regola è di andare a incrementare il numero di archi all'interno

del grafo (liste di adj) per muoverci più velocemente e cercare la soluzione in maniera ottimale.
Dato (1 or 2) and (3 or -2) otteniamo una nuova clausola (1 or 3) che corrisponde ad un nuovo arco, nel nostro grafo. Nella lista di adiacenza del grafo che sto creando segniamo sia adj[1,3] che adj[3,1] in modo tale da velocizzare la creazione di nuovi vincoli. Più clausole sono presenti e maggiori saranno le informazioni che si andranno ad ottenere.

Nel momento della creazione di nuove clausole viene effettuato un controllo, in cui, se una clausola entra in conflitto con un'altra, non sono presenti soluzioni. Se esisteranno entrambe le clausole (1 or 1) and (-1 or -1), non ci saranno soluzioni, poiché una delle due clausole sarà sicuramente falsa.

3. IMPLEMENTAZIONE

Nell'implementazione il codice è diviso in due file, main.cu e device.cu. Il main contiene tutte le istruzioni che vengono svolte dalla CPU e le chiamate alle funzioni che invece lavorano con la GPU.
L'implementazione durante la sua costruzione è stata migliorata svariate volte.
Il programma, all'avvio, riceve in input due elementi: un numero k, che rappresenta il numero massimo di soluzioni da trovare, e un file in formato txt che contiene le clausole come descritto nel cap. 1.1. Dal file .txt il programma legge il numero di letterali e il numero di vincoli che sono contenuti e di cui vogliamo trovare una soluzione.
Per esempio, il file contiene:

p cnf 3 4
-2 -3 0
-2 -1 0
-1 -3 0
-1 2 0

Dal .txt ricaviamo che il numero di letterali è 3 e il numero di vincoli 4. Dal numero di letterali calcoliamo il numero totale di tipologie possibili di nodi del grafo.

3.1 Rappresentazione dei dati

Il passo successivo è il salvataggio di tutti i vincoli in un array booleano di lunghezza 2*numero di letterali. Questo array lo possiamo immaginere come formato dalle celle bianche della matrice nella figura sottostante.

-	1	2	3	-1	-2	-3
1	0	0	0	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	0	0
-1	0	1	0	0	1	1
-2	0	0	0	1	0	1
-3	0	0	0	1	1	0

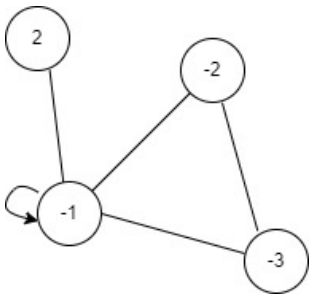
Come si vede nella matrice, il valore dato dall'incrocio tra una riga e una colonna determina, tramite 1 o 0, l'esistenza o meno di un vincolo/arco che collega i due nodi. Tramite una visita di tutti i vincoli del file, ottenuti dalla sua lettura, inseriamo 1 nella matrice quando esiste un vincolo tra due nodi.
Per ottenere un'ulteriore ottimizzazione, segniamo in un array, grande come il numero di letterali totali, la presenza o meno di quel letterale all'interno dei clausole. Questa operazione permetterà di svolgere in modo efficace dei controlli che snelliranno la ricerca di soluzioni.

3.2 Creazione nuovi vincoli

Il passo successivo è anche il più pesante che la GPU deve compiere. L'obiettivo è di applicare la Regola di Risoluzione vista al capitolo 2.2, che permette di ottenere nuove clausole. Più vincoli abbiamo e maggiori saranno le possibilità di trovare una soluzione nel minor tempo possibile.
Quindi, tramite una funzione che lavora sulla GPU, andiamo a far controllare ogni cella dell'array di adiacenza. Nel nostro caso, il thread 3 controllerà tutti i nodi con cui -1 è in collegamento e per ognuno di questi visiterà il nodo fratello. Per esempio -1 è connesso a 2, quindi il thread 3 andrà a controllare i nodi connessi a -2. -2 è connesso a -1 e -3 e seguendo la Regola di Risoluzione possiamo dire che anche -1 è connesso a -1. Il vincolo -1 -3 è già presente e per questo non aggiorniamo la matrice nella posizione corrispondente.
Dopo aver effettuato una chiamata alla funzione che svolge questo compito ed aver trovato nuovi vincoli, la matrice risultante da questo processo è:

-	1	2	3	-1	-2	-3
1	0	0	0	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	0	0
-1	0	1	0	1	1	1
-2	0	0	0	1	0	1
-3	0	0	0	1	1	0

Abbiamo ottenuto un array dei vincoli aggiornato e con questo possiamo creare visivamente un grafo.



Data l'aggiunta del vincolo -1 -1, le soluzioni possibili dovranno avere tutte -1 con valore True, poiché, se fosse False, il vincolo diventerebbe falso e di conseguenza la soluzione in questione sarebbe errata.
Nella funzione checkDiagonale, verificata l'esistenza di (-1 or -1) e della non esistenza di (1 or 1), andremo a garantire che tutte le soluzioni che cercheremo partiranno dando al nodo -1 il valore true.

Tramite questo array di 0 e 1 siamo in grado di iniziare a trovare delle possibili soluzioni.

3.3 Strutture dati

Per salvare e ricercare le k possibili soluzioni, oltre all'array di adiacenza e all'array per l'esistenza dei letterali, sono necessarie altre strutture dati. Ho utilizzato:

- sol: un array per la soluzione corrente;
- alternativeSol: array per la soluzione alternativa a quella corrente;
- finalSol: un array delle soluzioni finali grande nLett * k;
- solReg: un array che contiene tutte le prossime soluzioni da controllare;
- prox: una lista di double in cui inserisco la posizione da cui riprendere la soluzione salvata in solReg;
- status: array che viene modificato dalla GPU e serve come controllo all'interno delle procedure su scheda video.

3.4 Funzionamento

Dobbiamo immaginare il core del nostro programma di risoluzione come un albero con una radice che possiede un array di lunghezza 2*n, con n numero di letterali letto da file, tutto vuoto. Questa radice possiede due figli, formati da sol e da alternativeSol. sol è un array, come detto prima, che contiene la soluzione che stiamo verificando, mentre alternativeSol è una array alternativo che verrà copiato all'interno di solReg. Nella ricerca della soluzione, dentro sol, inserisco -1 e 1 nei primi due letterali fratelli che incontro che sono entrambi 0 e che esistono. Nell'array alternativeSol invece inserisco 1 e -1 nei corrispondenti letterali.
La soluzione prosegue ripetendo lo stesso step, in cui ora, a noi, interessano il figlio sinistro e destro di sol. Il figlio sinistro è la nostra soluzione da continuare a controllare, mentre il figlio destro viene salvato dentro solReg. Oltre a questo, ogni volta che salvo una soluzione in solReg, vado a salvare anche l'indice i, da cui riprendere quella soluzione, all'interno della lista prox.
Alla fine del processo di controllo dell'ultimo figlio sinistro, recupero la soluzione salvata più recentemente dentro solReg, in modo tale da continuare il processo. Ogni qual volta trovo una soluzione, la confronto con le altre dentro finalSol per avere certezza che essa sia nuova. Nel caso in cui una soluzione abbia una incongruenza, viene terminata e si passa alla successiva, recuperata direttamente da solReg. Nel caso in cui abbia trovato k soluzioni il programma termina. Per funzionare ed effettuare dei controlli ulteriori, il programma utilizza un array status, che viene modificato dalla GPU in caso di errori e incongruenze.

Al termine del programma vengono scritte le soluzioni all'interno di un file .txt.

Nel nostro esempio l'array sol, dopo tutti i precedenti passaggi, è rappresentato così:

1	2	3	-1	-2	-3
0	0	0	1	0	0

Durante il primo controllo in cui la i=0, troviamo che 1 ha 0 ma il -1 ha già un valore, quindi aumentiamo semplicemente i a 1. Nel controllo successivo invece, sia 2 che -2 hanno valore 0:

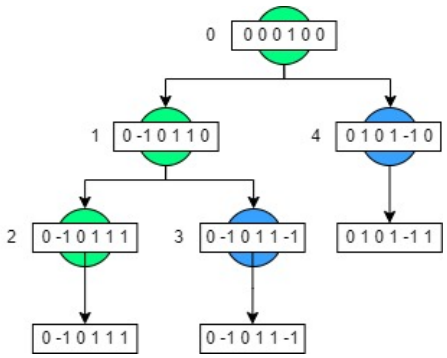
- sol: siccome sia 2 che -2 sono presenti nei letterali, assegniamo -1 alla posizione del 2 e 1 alla posizione del -2;

- alternativeSol: dopo aver copiato il sol iniziale (dopo il checkDiagonale) in alternativeSol, assegniamo valori opposti rispetto a quello dati a sol, in modo tale da avere tutte le possibili soluzioni. Assegniamo quindi 1 alla posizione del 2 e -1 alla posizione del -2. Salvo anche la i per continuare la soluzione senza ricominciare da capo.

Quello che otteniamo quindi è l'array **sol** così:

1	2	3	-1	-2	-3
0	-1	0	1	1	0

Il passo successivo è assegnare ai letterali connessi al 2 tutti 1. Infatti, l'unica soluzione accettata in questo senso, è che ai nodi con -1 siano collegati solo nodi con 1. Dopo aver dato anche i valori ai nodi fratelli, se la soluzione non è terminata, si continua ad assegnare valori alla soluzione corrente e a salvare quella alternativa; se invece è terminata si passa a recuperare una soluzione alternativa in solReg, copiandola in sol e poi in alternativeSol.
Il processo continua iterativamente, continuando a creare il figlio sinistro e destro di un nodo. Nel nostro esempio possiamo immaginare l'albero che si crea, così:



Al termine del programma, come spiegato sopra, salveremo le soluzioni trovate in un file. In questo caso sono tre:

-	1	2	3	-1	-2	-3
1	0	-1	0	1	1	1
2	0	-1	0	1	1	-1
3	0	1	0	1	-1	1

Possiamo notare come siano tutte e tre differenti.

4. RISULTATI SPERIMENTALI

Per studiare le prestazioni durante l'esecuzione, le variabili da valutare sono: il numero di letterali, il numero di vincoli, il numero di blocchi, il numero di thread e il numero k di soluzioni che sono state richieste. Infatti la variazione di uno di questi parametri, modifica il tempo di esecuzione, aggiungendo complessità, abbassando il lavoro richiesto ad ogni Multi-Processor, oppure richiedendo un numero maggiore di soluzioni e di fatto andando a richiedere un lavoro maggiore.

Per comprendere se l'utilizzo di un approccio parallelo ha portato dei benefici e i termini entro i quali questo tipo di approccio ha dei vantaggi, ho effettuato dei test facendo variare tutti i parametri sopra citati.

4.1 Hardware utilizzato

Per la creazione e sviluppo dei test ho utilizzato le GPU messe a disposizione da Google Colab: questo ha comportato una certa limitazione sulla quantità di risorse utilizzabili ma ha anche permesso l'accesso a macchine di per se molto potenti.

Specifiche tecniche:

CPU	Intel Xeon
Frequenza	2.20Ghz
Core	2
Thread	15
RAM	12 GB
GPU	Tesla T4
V-RAM	15 GB
SM	40
Thread	1024

4.2 Numero di vincoli e letterali

Nello studio, per comprendere le valutazioni e la scelta sui numeri proposti come test, è necessario osservare prima alcuni aspetti. Come spiegato nel capitolo 1, i letterali sono i nodi del grafo mentre i vincoli sono gli archi che collegano i nodi. I vincoli determinano l'esistenza dei nodi. Infatti se un nodo non è presenta in nessun vincolo non esiste nel grafo. Di conseguenza, la variazione del numero di vincoli porta a un differente tempo di esecuzione.

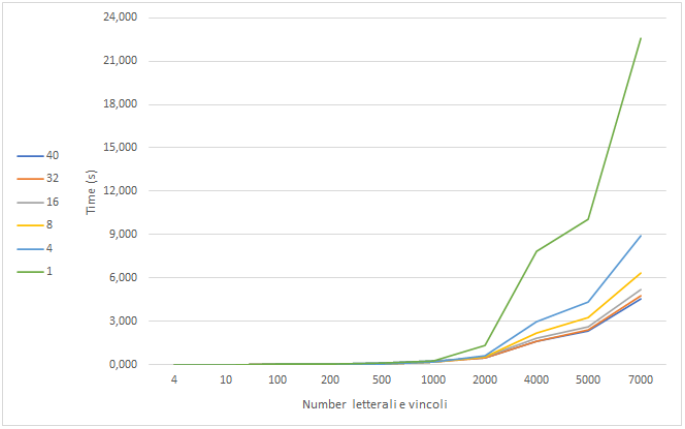
Quindi, dato un numero n di nodi e un numero m di vincoli:

- se n = m: ottengo un grafo che potenzialmente ha delle soluzioni;
- se n « m: ottengo una grande ripetizione di vincoli uguali tra loro e nessuna soluzione. Infatti otterrò molti nodi collegati a se stessi che porteranno quasi sicuramente ad una incongruenza.
- se n » m: ottengo molto probabilmente delle soluzioni per il numero ridotto di archi tra i pochi nodi che sono presenti nei vincoli. Il tempo di esecuzione diminuisce, con pochi vincoli ho pochi nodi connessi e di conseguenza un grafo più piccolo.

Nei test di esecuzione, il numero di letterali sarà uguale al numero di vincoli.

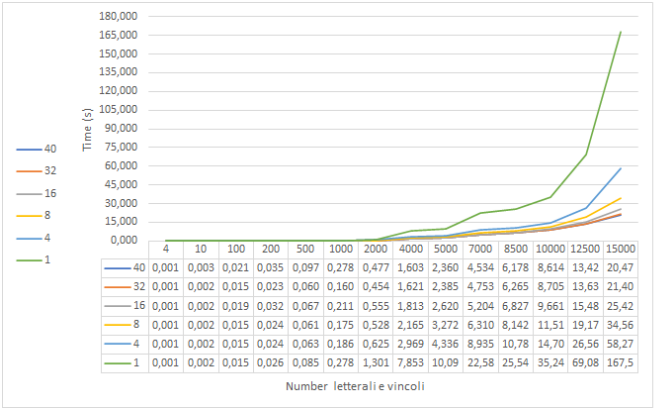
4.3 Esecuzione

Il grafico sotto riportato raccoglie i tempi globali di esecuzione con k costante, al variare del numero di letterali e di vincoli. Inoltre, sono stati raccolti i tempi facendo variare anche il numero di blocchi. Dal grafico possiamo notare che non c'è una variazione importante tra le varie serie. Ogni linea rappresenta l'esecuzione con un numero di blocchi differenti. Se andiamo a visualizzare fino a 1000 letterali e vincoli, i tempi non variano molto, poiché non c'è praticamente serializzazione.



Se visualizziamo il grafico, e osserviamo un numero di letterali e vincoli sopra i 2000 letterali, notiamo una differenza molto elevata con 5000 e poi con 7000. Da questi numeri, i thread dei blocchi devono necessariamente andare a svolgere più compiti e di conseguenza serializzare le operazioni.

Nel grafico sottostante, andando a valutare invece valori fino a 15000 letterali e vincoli, la differenza è evidente. Più blocchi ho e più sarà veloce l'esecuzione.



	4	10	100	200	500	1000	2000	4000	5000	7000	8500	10000	12500	15000
40	0,001	0,003	0,021	0,035	0,097	0,278	0,477	1,603	2,360	4,534	6,178	8,614	13,42	20,47
32	0,001	0,002	0,015	0,023	0,060	0,160	0,454	1,621	2,385	4,753	6,265	8,705	13,63	21,40
16	0,001	0,002	0,019	0,032	0,067	0,211	0,555	1,813	2,620	5,204	6,827	9,661	15,48	25,42
8	0,001	0,002	0,015	0,024	0,061	0,175	0,528	2,165	3,272	6,310	8,142	11,51	19,17	34,56
4	0,001	0,002	0,015	0,024	0,063	0,186	0,625	2,969	4,336	8,935	10,78	14,70	26,56	58,27
1	0,001	0,002	0,015	0,026	0,085	0,278	1,301	7,853	10,09	22,58	25,54	35,24	69,08	167,5

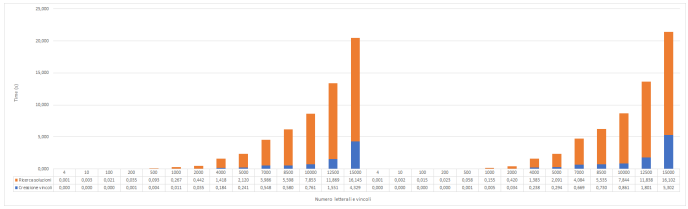
Possiamo notare che tra 1 blocco con 1024 thread e 40 blocchi con 1024 ognuno, la velocità è molto differente. Con 40 blocchi infatti la velocità scende ad

un decimo di quella con 1 blocco. Risulta evidente quindi il risparmio di tempo ottenuto andando ad aumentare in modo significativo il numero di blocchi. Dai valori del grafico si evince che, quando il numero di letterali*2 raggiunge il numero di blocchi*thread, allora abbiamo un significativo peggioramento. Infatti, nella creazione di vincoli, avremo molto lavoro da svolgere in qualsiasi caso, mentre quando dovrò accedere all'array che contiene una soluzione per lavorarci, ogni thread dovrà svolgere almeno due operazioni. Questo porta ad un peggioramento evidente.

4.4 Creazione vincoli e ricerca soluzioni

Per svolgere un'analisi approfondita, ho raccolto il tempo di esecuzione totale diviso tra tempo di creazione dei vincoli e tempo di ricerca delle soluzioni. Nel grafico sottostante sono presenti due raccolte dati:

- sinistra: risoluzione con 40 blocchi e 1024 thread per blocco;
- destra: risoluzione con 32 blocchi e 1024 thread per blocco.

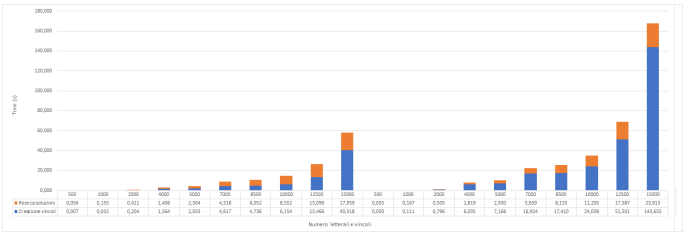


Dai grafici a barre è possibile notare che, se nei primi casi non ci sono grandi differenze sui tempi, quando si utilizzano numeri più grandi, si vedono delle differenze. Osserviamo che la parte che viene impattata maggiormente riguarda la creazione di vincoli, dove infatti la matrice delle adiacenze inizia ad assumere valori importanti. Di conseguenza, un numero di blocchi maggiore, aumenta di netto l'efficienza.

Nel grafico che segue invece, abbiamo lo stesso studio, con un numero di blocchi differente.

- sinistra: risoluzione con 4 blocchi e 1024 thread per blocco;
- destra: risoluzione con 1 blocco e 1024 thread per blocco.

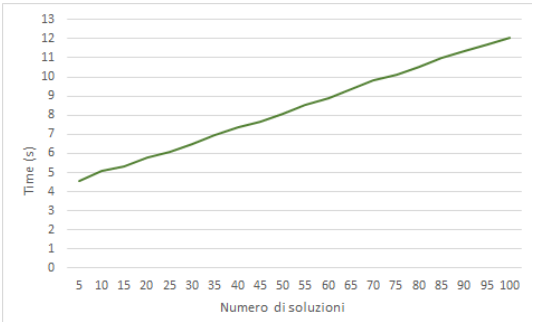
Si può notare che i tempi di esecuzione sono nettamente peggiori rispetto al grafico precedente e questo è dovuto principalmente al piccolo numero di blocchi che abbiamo stanziato. Un altro fattore che risalta e fortifica l'analisi precedente è l'impatto enorme che il tempo di creazione di nuovi vincoli (CV) ha rispetto al tempo di ricerca delle soluzioni (RS). Il tempo di RS non è molto maggiore del tempo trovato quando avevamo 40 blocchi. Invece il tempo di CV è enorme ed è dovuto alla differenza di grandezza della matrice di adiacenza rispetto al numero dimGrid*blockSize.



Risulta utile capire che migliorando sensibilmente il tempo di creazione vincoli si potrebbe abbassare in modo elevato il tempo di esecuzione.

4.5 Variazione di k

Per studiare ogni possibile cambiamento nella ricerca delle soluzioni, ho misurato i tempo di esecuzione totale variando il numero di soluzioni cercate. I dati sono stati raccolti considerando come possibile il trovare tutte le k soluzioni, dati il numero di vincoli e il numero di letterali. Ultimo vogliamo studiare come varia il tempo di ricerca soluzioni se variamo il numero di k.

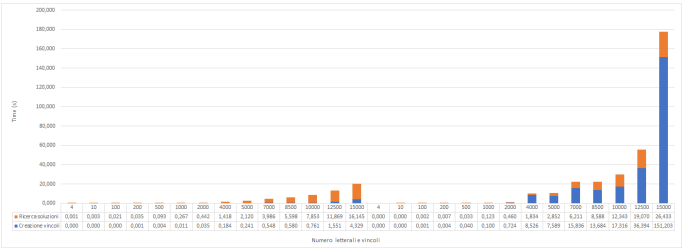


Dal grafico si evince che il tempo cresce linearmente al variare del numero k di soluzioni cercate, che varia da 5 a 100 e che dimostra che l'impatto della ricerca delle soluzioni è lineare a parità di numero di blocchi e di thread.

4.6 CPU solution

Per valutare in modo accurato se l'utilizzo di una soluzione parallela abbia portato in modo significativo dei miglioramenti rispetto ad una soluzione serializzata, ho sviluppato una versione che non utilizza la GPU.

Nel grafico sottostante abbiamo i tempi di esecuzione in GPU con 40 blocchi e 1024 thread e i tempi di esecuzione nella CPU. I tempi di esecuzione sono divisi in tempi di creazione di nuovi vincoli nel grafo e tempi di ricerca effettiva delle soluzioni.



Dal grafico si vede che, se nei grafi relativamente piccoli, quindi con massimo n nodi e n vincoli, la differenza tra esecuzione in GPU e CPU è irrisoria, ad un certo punto c'è una importante differenza.

Nei tempi di esecuzione notiamo subito una cosa interessante. La ricerca di nuovi vincoli risulta molto pesante per la CPU, poiché richiede oggettivamente molto lavoro. Invece la ricerca di nuove soluzioni, siccome nella soluzione in GPU è comunque localizzata sia in CPU che GPU, ha dei valori molto bassi quando lavoriamo solo su CPU. Infatti, un aspetto positivo di lavorare solamente nella CPU è che non c'è scambio di dati tra CPU e GPU, che seppur veloce, richiede tempo e risorse.

Nel complesso, quindi, è comunque meglio la soluzione in parallelo, poiché abbassa in modo drastico i tempi di esecuzione. Tuttavia nella ricerca delle soluzioni, fino a 4000 vincoli e letterali i tempi sono simili.

5. CONSIDERAZIONI

La soluzione da me costruita dimostra di essere valida quando, come visto nel grafico al paragrafo 4.6, il numero di vincoli e letterali diventa importante. Infatti, le richieste del problema erano dai 10000 letterali ai 50000 e dai 500000 vincoli al milione. In questi casi la mia soluzione parallela risulta buona rispetto ad una esecuzione sulla CPU. Un aspetto da considerare è che la mia soluzione non funziona oltre ai 22000 letterali poiché diventando 44000 tra negativi e positivi, la matrice delle adiacenze occupa molta memoria. Resta però utilizzabile un numero di vincoli anche sopra il milione, che però come spiegato nel paragrafo 4.2, porta sempre all'interruzione dell'esecuzione per una incongruenza dei vincoli, se il numero di clausole è troppo grande rispetto al numero di letterali.

Come evidenziato dai dati, la parte più pesante dell'esecuzione è la creazione di nuovi vincoli, che ho fatto solamente una volta poiché molto stressante. La matrice, se partiamo da 22000 letterali, diventa grande 44000*44000. Un possibile miglioramento sarebbe quello di utilizzare una struttura dati differente o cercare di ottimizzare l'array delle adiacenze per occupare molto meno spazio nei casi in cui questa sia effettivamente piena di zeri.

Un ulteriore possibilità di miglioramento è dimostrata dai tempi di esecuzione della ricerca delle soluzioni. Come abbiamo notato prima, il tempo di ricerca delle soluzioni tra GPU con 40 blocchi e il tempo di ricerca delle soluzioni sulla CPU, è molto simile. Questo è dovuto al passaggio di dati tra memorie, ma anche al fatto che seppure la ricerca delle soluzioni avvenga in parallelo, quello che succede è che lavoriamo con un'unica soluzione alla volta. Questo aspetto potrebbe essere migliorato andando a creare e cercare più soluzioni nello stesso momento, rendendo la ricerca delle soluzioni totalmente parallela e di conseguenza abbassando il tempo di esecuzione.

6. Pseudo-codice

L'implementazione e una esecuzione simulata del programma sono state spiegate nel capitolo 3.
Nelle prime righe del main vengono dichiarate le strutture dati che verranno utilizzate nell'esecuzione del programma. Oltre a tutte quelle sull'host, verranno stanziate anche le corrispondenti nel device.
Il main poi continua andando a popolare, tramite la lettura dei vincoli, l'array delle adiacenze (adj). Vengono create variabili utili alla esecuzione e, alla riga 12, viene creato l'array alreadyVisited che permette di risparmiare lavoro durante l'esecuzione.

Algorithm 1 Main - Creazione vincoli

```
1:  Read litteral from file
2:  Read constraint from file
3:  Read from console k
4:  len ← 2 * numberLitteral
5:  Create adj array, littExist array (bool), sol array, alternSol array, regSol array, finalSol array (int) in both memory.
6:  Inizialize adj to zero, sol/alternSol to zero, regSol to zero.
7:  for all (i,j) in constraints do
8:      adj[(i, j)] ← 1
9:      adj[(j, i)] ← 1
10: end for
11: Copy all data host vector to device vector.
12: Create index, cSol, alrVisit array (bool).
13: Launch createConstraint(adj, len, len*len, status) on device
14: Launch checkDiagonal(adj, len, sol, status) on device
15: Copy status to Host
16: if status[0] == 1 then
17:     return
18: end if
19: Memset status to 0 and copy to device
```

Nel codice, alla riga 13, viene chiamato createConstraint. Questo algoritmo permette di andare a creare nuovi vincoli, come spiegato nel capitolo 3, grazie alla regola di risoluzione.
All'interno di createConstraint si procede scandagliando ogni posizione dell'array delle adiacenze, per trovare, dato (1 or 2) AND (3 or -2), il nuovo vincolo (1 or 3).

Algorithm 2 createConstraint(adj, len, sizeAdj, status)

```
1:  thid ← getThreadBlockID
2:  if thid < sizeAdj then
3:      if adj[thid] then
4:          sec ← (thid%len)
5:          if sec > (len/2) then
6:              sec ← sec − (len/2)
7:          else
8:              sec ← sec + (len/2)
9:          end ifpri = floor(thid / len)
10:         for i from (sec*len) to ((sec+1)*len), i <- i+1 do
11:             pos ← (pri * len) + (i%len)
12:             if adj[pos] != 1 then
13:                 if adj[i] AND ((i % len) + 1)) then
14:                     adj[pos] ← 1
15:                     a ← (pos%len) * len
16:                     b ← floorf(pos/len)
17:                     adj[a + b] ← 1
18:                 end if
19:             end if
20:         end for
21:     end if
22: end if
```

L'algoritmo di checkDiagonal, alla riga 14 del main, permette di determinare se esiste un conflitto all'interno della diagonale della nostra matrice di adiacenza. Se esiste sia un vincolo (1 or 1) che (-1 or -1), non ci sono soluzioni. Inoltre, se esiste (1 or 1) e non esiste (-1 or -1), sicuramente 1 avrà valore Vero nella nostra soluzione. Questo permette di velocizzare la risoluzione. Vediamo ora il codice di checkDiagonal:

Algorithm 3 checkDiagonal(adj, len, sol, status)

```
1:  thid ← getThreadBlockID
2:  if thid % (len + 1) == 0 AND thid < (len * len) then
3:      thid2 ← (len + 1) * (len/2) + thid
4:      if adj[thid] == 1 AND adj[thid] == 1 then
5:          status[0] ← status[0]OR1
6:          if adj[thid] == 1 AND adj[thid2] == 0 then
7:              sol[thid%len] ← 1
8:          else if adj[thid] == 0 AND adj[thid2] == 1 then
9:              sol[thid2%len] ← 1
10:         end if
11:     end if
12: end if
```

Nell'esecuzione andiamo a modificare un valore nell'array status se troviamo un errore. Questo passo ci permette poi di interrompere l'esecuzione al principio. Dopo le chiamate a queste due procedure, se abbiamo un conflitto, il programma termina completamente. Nel caso invece che non ci siano conflitti prosegue.
Per comprende al meglio la seconda parte del main, che vedremo dopo, vediamo alcune procedure che esso utilizza.
La procedura saveNextSol si occupa di andare a salvare la soluzione alternativa in regSol.

Algorithm 4 saveNextSol(regSol, altSol, len, cSol)

```
1:  thid ← getThreadBlockID
2:  if thid < len then
3:      regSol[(cSol * len) + thid] = altSol[thid]
4:  end if
```

La procedura completeSol si occupa di andare a completare la soluzione che riceve in input. Se vede che 3 è vero e -3 esiste nei vincoli, assegna falso a -3.

Algorithm 5 completeSol(sol, len, littExist)

```
1:  thid ← getThreadBlockID
2:  if thid < len/2 then
3:      if sol[thid] == 0 AND sol[thid + (len/2)] != 0 AND littExist[thid] == 1 then
4:          if sol[thid + (len/2)] == 1 then
5:              sol[thid] ← −1
6:          else if sol[thid + (len/2)] == -1 then
7:              sol[thid] ← 1
8:          end if
9:      end if
10:     if sol[thid] != 0 AND sol[thid + (len/2)] == 0 AND littExist[thid + (len/2)] == 1 then
11:         if sol[thid] == 1 then
12:             sol[thid + (len/2)] ← −1
13:         else if sol[thid] == -1 then
14:             sol[thid + (len/2)] ← 1
15:         end if
16:     end if
17: end if
```

La procedura checkRow assegna 1 ai nodi/letterali connessi a -1. Infatti sappiamo che un nodo che ha come valore -1 (falso) può essere connesso solamente a nodi con valore 1 (vero). Se trova incongruenze lo segnala.

Algorithm 6 checkRow(adj, sol, len, status, alrVisit, littExist)

```
1: thid ← getThreadBlockID
2: if thid < len then
3:     if sol[thid] == -1 AND alrVisit[thid] == 0 then
4:         for i from 0 to len, i <- i+1 do
5:             if adj[thid * len+i] == 1 then
6:                 if sol[i] == 0 then
7:                     sol[i] ← 1
8:                     if i >= (len/2) then
9:                         if sol[i-(len/2)] == 1 then
10:                            status[1] ← status[1] OR 1
11:                        end if
12:                    else if i<(len/2) then
13:                        if sol[i+(len/2)] == 1 then
14:                            status[1] ← status[1] OR 1
15:                        end if
16:                    end if
17:                    status[0] ← status[0] OR 1
18:                end if
19:                if sol[i] == -1 then
20:                    status[1] ← status[1] OR 1
21:                end if
22:            end if
23:        end for
24:        d_alreadyVisited[thid] ← 1;
25:        if thid < len/2 then
26:            d_alreadyVisited[thid + len/2] ← 1
27:        end if
28:    end if
29: end if
```

La procedura checkNewSol si occupa di controllare che la soluzione trovata sia nuova, confrontandola con quelle in finalSol.

Algorithm 7 checkNewSol(sol, finalSol, len, indexSol, status)

```
1: thid ← getThreadBlockID
2: if thid < indexSol then
3:     i ← 0
4:     while i < len AND finalSol[thid*len + i] == sol[i] do
5:         i ← i + 1
6:     end while
7:     if i == len then
8:         status[2] ← status[2]OR1
9:     end if
10: end if
```

La procedura copyNextSol si occupa di recuperare l'ultima soluzione alternativa salvata.

Algorithm 8 copyNextSol(regSol, sol, len, cSol)

```
1: thid ← getThreadBlockID
2: if thid < len then
3:     sol[thid] ← regSol[(cSol * len) + thid]
4: end if
```

Ora procediamo con il codice della seconda parte del main, che utilizza tutte le procedure viste prima.

Algorithm 9 Main 2 - Ricerca soluzioni

```
1: Create resSol ← false, esiste ← false, continua ← false, i ← 0, Create list prox (double)
2: repeat
3:     continua ← false, Copy sol to device
4:     repeat
5:         Memset status to 0 and copy to device
6:         if sol[i] == 0 AND sol[i+l] == 0 AND resSol == false then
7:             Copy sol in altSol
8:             if littExist[i] then
9:                 sol[i] ← −1, alternativeSol[i] ← 1, esiste ← true
10:            end if
11:            if littExist[i + l/2] then
12:                sol[i + l/2] ← 1, alternativeSol[i + l/2] ← −1, esiste ← true
13:            end if
14:            if esiste then
15:                prox[0].push_back(i), CopyaltSoltodevice
16:                saveNextSol(regSol, altSol, len, cSol) on device
17:                cSol ← cSol + 1, Copysoltodevice, resSol ← true, esiste ← false
18:            end if
19:        end if
20:        if !resSol then
21:            i ← i + 1
22:        end if
23:        if resSol then
24:            checkRow(adj, sol, len, status, alrVisit, littExist) on device
25:            completeSol(sol, len, littExist) on device
26:            Copy status do Host, Copy sol to Host
27:            if status[0] == 0 then
28:                resSol ← false
29:            end if
30:            if status[1] == 1 then
31:                break
32:            end if
33:        end if
34:    until i < l
35:    if status[1] == 0 then
36:        Copy sol to Host
37:        if indexSol > 0 then
38:            checkNewSol(sol, finalSol, len, indexSol, status) on device
39:        end if
40:        if status[2] == 0 OR indexSol == 0 then
41:            k ← k − 1
42:            for cu from 0 to len, cu <- cu+1 do
43:                finalSol[indexSol ∗ len + cu] = sol[cu]
44:            end for
45:            indexSol ← indexSol + 1, Copy finalSol to device
46:        end if
47:    end if
48:    Memset status to 0, Copy status to device
49:    if cSol > 0 then
50:        Memset sol to 0, i = prox[0].back(), cSol ← cSol − 1, Copy regSol to Host
51:        copyNextSol(regSol, sol, len, cSol) on device
52:        Copy sol to host, Copy alreadyVis to device, resSol ← true, continua ← true
53:    end if
54:    Memset to 0 alreadyVisited, Copy to device
55: until continua AND k > 0
```

7. Compilazione ed Esecuzione

Per compilare il progetto è possibile posizionarsi nella directory src e utilizzare il comando **make compile**, che compilerà il progetto. Per eseguire il programma si utilizza il comando **make run**. Se si vuole compilare e eseguire insieme si utilizza il comando **make**.

All'interno della directory src sono già presenti i file compilati. Di conseguenza, in assenza di ambiente di compilazione si può eseguire direttamente.

L'esecuzione prende in input 5 file di vincoli, presenti nella directory **vincoli**.

Dopo l'esecuzione, le soluzioni vengono salvate dentro la cartella soluzioni e le durata nel file **duration.txt**.

Per eseguire singolarmente su un file a scelta è possibile utilizzare:

- Su gpu in Linux: **./main -k=4 -file=v1.txt**
- Su cpu in Linux: **./maino -k=4 -file=v1.txt**

7.1 Creazione e Controllo soluzioni

Per andare a creare dei file dei vincoli in modo automatico, ho scritto un programma in python **create.py**. Da cmd, utilizzando il comando **python create.py name.txt n m**, con name il nome del file, n il numero di letterali e m il numero di clausole, il programma scriverà in formato DIMACS un file name con le specifiche richieste.

Inoltre, è possibile controllare che le soluzioni trovate siano tutte differenti, tramite l'esecuzione di **python compare.py solv1.txt**.

Tramite l'esecuzione di **python check.py v1.txt solv1.txt** è possibile verificare se le soluzioni trovate siano effettivamente delle soluzioni per i vincoli da cui siamo partiti.