CUDA Ant Colony Optimization per TSP simmetrico

Antonio Belotti
mat. 960822

1 Introduzione

In questo progetto ho implementato un algoritmo ant colony optimization per la risoluzione del traveling salesman problem. Lo scopo è di confrontare le performance tra l'implementazione sequenziale classica e l'implementazione parallela su CUDA, cercando uno speed-up significativo.

2 Traveling salesman problem

Ho considerato il problema del TSP simmetrico su grafo completo. L'obiettivo è trovare un cammino hamiltoniano di costo minimo. Il problema è NP-hard e quindi spesso trattato con algoritmi euristici. Per eseguire test significativi, ho usato delle istanze dal dataset benchmark TSPlib[5]. In particolare: att48, a280, att532, d1291, d2103, fl3795.

3 Ant colony optimization

Ant colony optimization (ACO) è una meta-euristica introdotta da Dorigo [2] che simula una colonia di formiche e si presta molto bene per risolvere problemi su grafo. Questi algoritmi funzionano definendo il comportamento delle formiche in modo opportuno per la risoluzione del problema considerato. In generale, ogni formica costruisce iterativamente una soluzione, ed il suo comportamento è influenzato dal peso degli archi e dalle quantità di feromone presenti sugli archi. Il feromone è un segnale che attira le formiche; un arco sarà quindi più papabile per una formica se avrà una grande quantità di feromone. Ogni formica deposita su ogni arco su cui passa una quantità di feromone proporzionale alla bontà della sua soluzione; ad ogni iterazione tutte le tracce di feromone evaporano di una quantità prefissata.
Per applicare la meta-euristica ad un problema specifico bisogna definire la strategia di deposito ed evaporazione del feromone, l'importanza del peso degli archi, l'importanza del feromone sugli archi ed il criterio di arresto.

3.1 Algoritmo utilizzato

L'algoritmo implementato è ad opera di Akihiro Uchida, Yasuaki Ito, and Koji Nakano [6].

Consideriamo il grafo pesato non orientato $G = (V, E)$ dove $V = \{v_0, ..., v_n\}$ è l'insieme dei nodi e $E = \{e_{i,j} : v_i, v_j \in V\}$ è l'insieme degli archi. Il numero di nodi è $n = |V|$. Ogni arco $e_{i,j}$ ha associato un costo $d_{i,j}$ ed una quantità di feromone $\tau_{i,j}$. Il costo è fisso, mentre il feromone varia durante l'esecuzione dell'algoritmo.
Si applicano iterativamente i seguenti passaggi.

Costruzione percorsi Vengono posizionate n formiche su altrettanti nodi scelti uniformemente a caso. Ogni formica costruisce un percorso iterativamente, scegliendo un nodo alla volta. Consideriamo una formica attualmente sul nodo v_i. Il *fitness* dell'arco $e_{i,j}$ è definito dall'equazione 1.

$$f_{i,j} = \frac{\tau_{i,j}^{\alpha}}{d_{i,j}^{\beta}} \tag{1}$$

Gli iperparametri α e β definiscono l'importanza rispettivamente della traccia di feromone e del peso degli archi. La probabilità che la formica in questione, attualmente sul nodo v_i scelga il nodo v_j come prossimo nodo è data da

$$p_{i,j} = \begin{cases} \frac{f_{i,j}}{F} & v_j \in U \\ 0 & altrimenti \end{cases} \tag{2}$$

dove U è l'insieme delle città non ancora visitate dalla formica e $F = \sum_{v_j \in U} \frac{\tau_{i,j}^{\alpha}}{d_{i,j}^{\beta}}$ come fattore di normalizzazione.

Quando tutte le formiche avranno terminato di costruire il proprio percorso si cercherà il percorso meno costoso. In caso sia meno costoso del miglior percorso delle iterazioni precedenti, se ne terrà memoria come soluzione migliore fino a questo punto.

Aggiornamento feromone Per consentire la scoperta di soluzioni migliori e scoraggiare la creazione di soluzioni pessime è necessario aggiornare le quantità di feromone presenti sugli archi. Sono necessari due passaggi:

1. Evaporazione: $\tau_{i,j} \leftarrow (1-p)\tau_{i,j}$ dove p è il coefficiente di evaporazione deciso a priori.
2. Deposito:

$$\tau_{i,j} \leftarrow \tau_{i,j} + \sum_{k=1}^{n} \sum_{(v_i,v_j)\in T_k} \frac{1}{L(T_k)}$$

   dove T_k è il percorso costruito dalla k-esima formica e $L(T_k)$ è suo costo.

Per evitare di rendere un arco completamente non appetibile per una formica, si forza la traccia di feromone a non scendere mai sotto un lower bound costante definito a priori. Una volta aggiornate le traccie di feromone, si può ricominciare con la costruzione di nuovi percorsi fino al verificarsi del criterio di arresto.

Arresto L'algoritmo termina quando sono state eseguite le iterazioni prefissate oppure quando non ci sono miglioramenti per un numero prefissato di iterazioni.

4 Implementazione parallela

Strutture dati L'implementazione è basata esclusivamente su array monodimensionali. Gli array bidimensionali sono salvati in array monodimensionali in modo row-wise. Gli array sono sufficienti per risolvere il problema ma sono anche "comodi" per sfruttare al meglio la gpu. È consigliabile infatti organizzare i dati in zone di memoria contigue per favorire la programmazione di kernel che sfruttano l'accesso ai dati coalesced. Nella maggior parte dei casi, la cella in posizione i in un array monodimensionale contiene una qualche informazione relativa al nodo i. Nel caso di una matrice, in generale l'i-esima riga è relativa all'i-esima formica, oppure la cella in posizione (i, j) contiene un informazione per l'arco (i, j), come ad esempio il feromone o il peso degli archi.
Un path sul grafo è rappresentato in un array di lunghezza n come in figura 1.

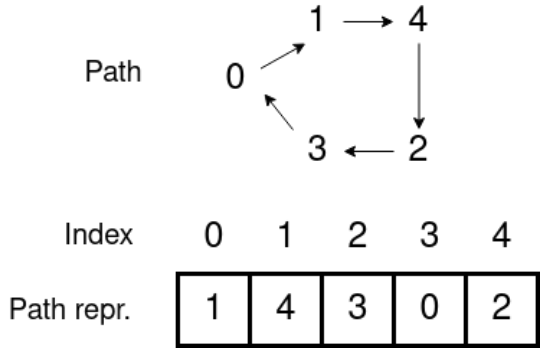

Figure 1: Rappresentazione percorso

Roulette-wheel selection La selezione di ogni nodo nella costruzione del percorso di ogni formica è la parte più critica dell'algoritmo per complessità ed il numero di volte che viene eseguita. Per ogni formica, si crea un array q di lunghezza n che contiene il fitness di ogni nodo (Eq. (1)). Si applica a q la bit-mask *unvisited_nodes* che indica i nodi ancora da visitare. Si esegue poi una somma prefissa inclusiva su q: in questo modo si creano dei bucket, ovvero dei range numerici che sono grandi proporzionalmente al fitness di ogni nodo. Si estrae poi uniformemente a caso un numero r nel range $[0, q_n)$ e si seleziona il nodo j che identifica il bucket in cui il numero casuale è contenuto. Un esempio è riportato in figura 2. La somma prefissa è un problema risolvibile efficientemente in parallelo. La ricerca del bucket in cui ricade il numero casuale è fatta con n thread che confrontano ogni cella con la precedente.

Dato che eseguire la somma prefissa per ogni singola formica distintamente è oneroso per la GPU, viene eseguita la somma prefissa sull'intera matrice $n \times n$ del fitness di tutte le formiche. I range in cui estrarre r saranno quindi diversi per ogni formica.

Utilizzo Memoria Gli iperparametri del problema sono salvati in constant memory, dato che sono

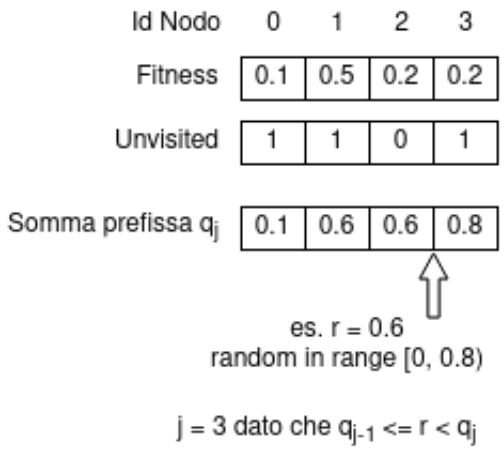

Figure 2: Roulette wheel selection

valori costanti singoli richiesti da molti thread. Dove possibile (o necessario) si sfrutta la shared memory.

4.1 CUDA Prefix Sum

La performance dell'algoritmo è dettata dalle performance della somma prefissa. Per calcolare la somma prefissa ho utilizzato l'algoritmo scan proposto da Blelloch [1] e descritto per architettura CUDA da Harris et.al. [3]. L'idea è di scorrere l'array come se fosse un albero binario. I nodi collegati indicheranno quali elementi saranno da sommare per trovare la somma totale dell'array e successivamente ottenere in ogni cella (nodo dell'albero) il valore di somma prefissa corretto. Il procedimento è diviso in due passate, una *up-sweep* (figura 3) ed una *down-sweep* (figura 4). Nell'up-sweep si scorre l'albero dalle foglie verso la radice e si sommano i nodi collegati. Al termine dell'up-sweep si ottiene nell'ultima cella il valore della radice, ovvero la somma di tutti gli elementi nell'array. L'up-sweep è una reduction in parallelo. Nel down-sweep si scorre l'albero dalla radice verso le foglie, usando le somme parziali presenti nell'array dalla fase precedente per far risultare in ogni cella il valore di somma prefissa corretto.
Per scorrere i livelli di profondità dell'albero si usa la variabile d nel range $[0, log(n)]$.

Nella formulazione normale, questo approccio calcola la somma prefissa esclusiva. Ho leggermente modificato la procedura per consentire anche la somma inclusiva, scrivendo il risultato finale spostato a sinistra di una cella.

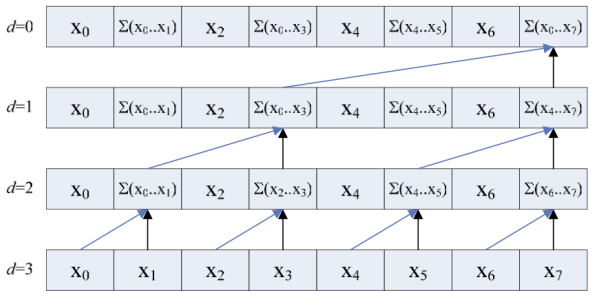

Figure 3: up-sweep phase (da Harris[3])

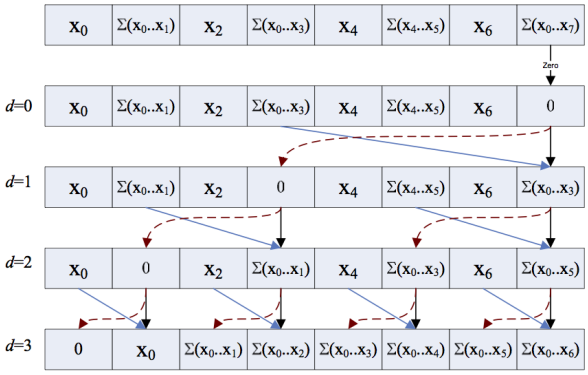

Figure 4: down-sweep phase (da Harris[3])

Per trattare un array di n elementi è necessario lanciare $\frac{n}{2}$ thread, perché nel primo passaggio di up-sweep si sommano altrettante coppie di elementi. Si ottiene quindi che nella pratica, un blocco di thread può eseguire la somma prefissa per un array di dimensione $\leq (2 \cdot \text{max thread per block})$.

Utilizzo memoria Per ottenere migliori performance, ha senso caricare all'inizio l'array in memoria condivisa, eseguire la somma e poi scrivere in memoria globale il risultato. In questo modo si legge e si

scrive solo una volta in memoria globale. Ogni thread si occupa quindi di trasferire 2 elementi da memoria globale a condivisa e viceversa.

Dimensione array Questo algoritmo richiede che l'array dato in input abbia una lunghezza potenza di due, perché viene "costruito" un albero binario. Per eseguire la somma su array di lunghezza arbitraria, è necessario fare padding alla destra dell'array inserendo degli zeri, che non influiscono nella somma, fino ad ottenere una lunghezza potenza di due. Nell'implementazione, lo zero-padding è inserito durante il caricamento dei valori in shared memory.

In caso l'array originale sia più grande del limite dovuto al numero massimo di thread per blocco, è necessario calcolare la somma prefissa in chunk. L'approccio usato è riportato nella figura 5. Come primo passo si esegue la somma prefissa su ogni chunk e si memorizza il valore massimo di ogni chunk in un array di appoggio. Si calcola quindi la somma prefissa esclusiva sull'array di appoggio ottenendo l'incremento da applicare ad ogni elemento in ogni chunk. Sommato l'incremento corretto ad ogni elemento di ogni chunk si ottiene il risultato desiderato. Se l'array di appoggio è anch'esso più grande della dimensione massima di un blocco, sarà necessario eseguire ricorsivamente questa procedura per calcolarne la somma prefissa.

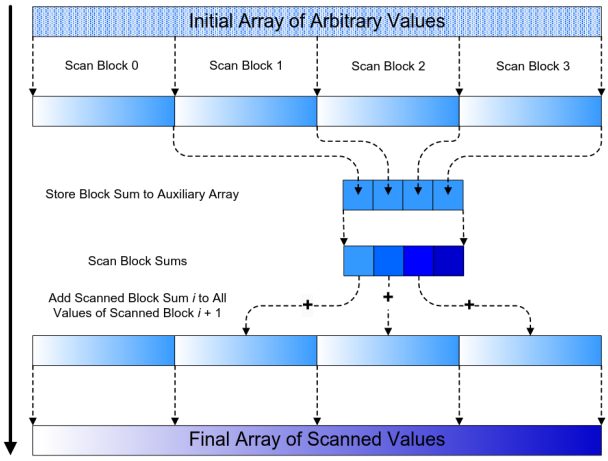

Figure 5: Somma prefissa su array di arbitraria dimensione (da Harris[3])

4.2 Aggiornamento feromone

La seconda operazione più onerosa nel programma è l'aggiornamento delle tracce di feromone. Pur essendo un operazione semplice, non sfrutta a pieno il parallelismo offerto dalla GPU perché richiede l'utilizzo di operazioni atomiche, accesso in memoria uncoalesced e non è possibile sfruttare la memoria condivisa (almeno non in maniera immediata). Si lancia una griglia di thread di dimensione n^2. Ogni thread legge una cella dalla matrice dei percorsi delle formiche e identifica quindi un arco che la data formica attraversa. Ogni thread incrementerà la cella relativa all'arco corretto nella matrice edge fitness. Questo incremento va fatto con operazioni atomiche perché potrebbero esserci accessi concorrenti. Non è

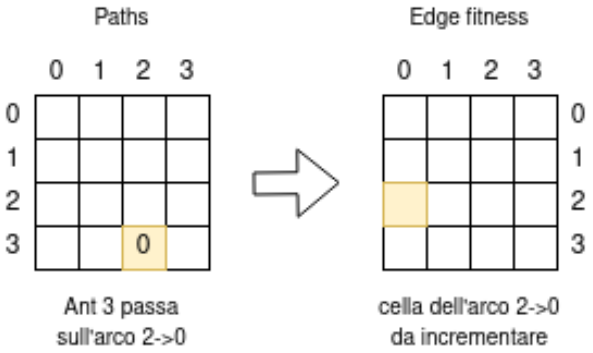

Figure 6: Aggiornamento feromone

possibile creare semplicemente un risultato intermedio in memoria shared perché servirebbe allocare una matrice n^2 per ogni blocco.

Dato che il problema è riconducibile alla creazione di un istogramma, si potrebbe cambiare approccio.
Provando un approccio simile ma partendo dalla matrice *edge fitness* invece che dalla matrice *path* ho ottenuto performance peggiori.
Avevo pensato di calcolare la trasposizione di *path* in modo da avere sulla riga i tutti gli archi $i \to x$ attraversati dalle formiche. Contando le occorrenze dei vari elementi riga per riga si otterrebbe il numero di formiche che attraversano ogni arco ma poi non si potrebbe fare l'update del feromone correttamente perché ogni formica deposita una quantità variabile di feromone.

5 Implementazione sequenziale

Per avere un riscontro dell'effettivo miglioramento delle performance, ho implementato una versione sequenziale. Le operazioni sono identiche all'algoritmo parallelo ma eseguite in versione sequenziale. L'algoritmo è ragionevole anche se non dovesse essere confrontato con una versione parallela, dato che la somma prefissa è un buon modo per fare roulette wheel selection anche in ambiente sequenziale.

6 Performance

Nella tabella 6 sono riportati i tempi di esecuzione delle due implementazioni sulle istanze considerate ed il relativo speed-up. Il programma è stato compilato con NVCC 11.2 ed eseguito su una macchina di Google Colab con CPU single core Intel Xeon@2.20GHz e GPU Tesla T4. Istanze più grandi richiedevano troppo tempo (su CPU) e l'esecuzione veniva terminata da Colab.

Istanza	CPU runtime	GPU runtime	Speedup
att48	0.041 sec	0.352 sec	0.116477
a280	6.683 sec	0.848 sec	7.880896
att532	16.847 sec	1.304 sec	12.919478
d1291	160.237 sec	7.690 sec	20.837061
d2103	877.242 sec	28.914 sec	30.339697
fl3795	2611.220 sec	84.595 sec	30.867310

6.1 Considerazioni - profiling

Il programma contiene alcuni kernel che applicano una semplice operazione su ogni elemento di un array. In base alla dimensione dell'istanza da risolvere, questi array potrebbero essere di dimensioni troppo piccole per sfruttare appieno le capacita della GPU, oppure troppo grandi e quindi diventare memory-bound. Nel primo caso, data la piccola dimensione dell'istanza, l'utilizzo della GPU non è giustificato perché è impossibile sfruttare le capacità del device (come si può vedere anche dai risultati sull'istanza att48). Nel secondo caso si potrebbe pensare di fare kernel-fusion ma l'ordine di esecuzione dei vari kernel non lo consentirebbe agilmente. Inoltre, questi kernel influiscono marginalmente sulle performance totali, che sono dettate in larga parte dalla somma prefissa ed in piccola parte dalla procedura di update del fitness sugli archi.

Il profiler CUDA riporta che il kernel della somma prefissa raggiunge un occupancy $> 90\%$. Dato che il kernel lavora su un array di dimensione n^2 è possibile (per istanze non troppo piccole) lanciare una griglia composta da molti blocchi, molto superiore al numero di SM sul device e quindi si ottiene buona occupancy. Per come sono strutturate up-sweep e down-sweep si ha thread divergence con dei thread che restano inattivi ed aspettano alle barriere di sincronizzazione. Un possibile miglioramento per questo kernel potrebbe venire dall'implementazione di strategie per evitare shared memory bank conflicts[3]. Ciò richiederebbe però una quantità maggiore di memoria condivisa per ogni blocco.

Il kernel di aggiornamento del feromone ha anch'esso high occupancy perché vengono lanciati parecchi blocchi, ma ha performance pessime a causa delle operazioni che svolge. Atomic add causa thread divergence e thread stall e l'accesso a memoria globale uncoalesced è inefficiente.

7 Conclusioni

Ant colony optimization può essere significativamente accelerato utilizzando una GPU. È sicuramente possibile sviluppare ulteriormente l'implementazione in cerca di performance migliori, per esempio implementando la somma prefissa con l'approccio decoupled lookback [4].

Il notebook con il codice di questo progetto è disponibile su github[1].

References

[1] E Blelloch Guy. *Prefix Sums and Their Applications*. Tech. rep. Tech. rept. CMU-CS-90-190. School of Computer Science, Carnegie Mellon . . ., 1990.

[2] M DORIGO. "Optimization, Learning and Natural Algorithms". In: *Ph. D. Thesis, Politecnico di Milano* (1992).

[3] Mark Harris. *Parallel Prefix Sum (Scan) with CUDA*. `https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/scan/doc/scan.pdf`. 2007.

[4] Duane Merrill and Michael Garland. "Single-pass Parallel Prefix Scan with Decoupled Look-back". In: 2016.

[5] Gerhard Reinelt. "TSPLIB–A Traveling Salesman Problem Library". In: *ORSA Journal on Computing* 3.4 (1991), pp. 376–384.

[6] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. "Accelerating ant colony optimisation for the travelling salesman problem on the GPU". In: *International Journal of Parallel, Emergent and Distributed Systems* 29.4 (2014), pp. 401–420. DOI: 10.1080/17445760.2013.842568. eprint: `https://doi.org/10.1080/17445760.2013.842568`. URL: `https://doi.org/10.1080/17445760.2013.842568`.

[1] `https://github.com/antoniobelotti/gpu_aco_tsp`