Progetto relativo al corso di Programmazione su architetture parallerle
a.a. 2021-2022

Gianluca Macrì
mat. 142213

Indice

1 Problema affrontato

Lo scopo di questo lavoro è stato quello di implementare mediante il linguaggio di programmazione CUDA una versione parallela dell'*Auction algorithm*, originariamente proposto da Bertsekas (1979), per la risoluzione del problema del massimo matching su grafi. In particolare, analogamente all'approccio seguito da Vasconcelos e Rosenhahn (2009), ci si limiterà al caso semplificato dei grafi bipartiti per il quale è sempre possibile ottenere una soluzione ottimale, quando questa esista.

Per grafo bipartito si intende un grafo $G = (V, E)$ con $X, Y \subseteq V$ partizione dei nodi di V, ossia $X \cup Y = V$ e $X \cap Y = \emptyset$, e con archi che collegano esclusivamente nodi appartenenti a partizioni differenti, i.e. $E \subseteq X \times Y$. Un matching, o accoppiamento, M è quindi definito come un sottoinsieme di archi $M \subseteq E$ tale che per ciascun nodo $v_1 \in V$ esiste al più un arco in M incidente in esso, ossia $\forall v_1 \in V\, ((\exists v_2 \in V : (v_1, v_2) \in M) \implies \neg (\exists v_3 \in V : v_3 \neq v_2\, (v_1, v_3) \in M))$.

Quando si parla di *matching problem*, si tende tipicamente ad associare un'interpretazione semantica ai due insiemi X ed Y, ad esempio nel caso dell'*Auction algorithm* indicandoli rispettivamente come partecipanti ad un asta e lotti (od oggetti) in vendita. In tale scenario, inoltre, si associano agli archi dei pesi a_{ij} per $(i, j) \in E$ che, corrispondono al valore che ciascuna persona attribuisce soggettivamente ai diversi lotti, e si cerca di associare ad ogni persona esattamente un oggetto in modo tale da massimizzare la funzione di profitto $\sum_{(i,j)\in M} a_{ij}$. Chiaramente, detti m ed n rispettivamente in numero di persone (o la cardinalità di X) e il numero di oggetti (o la cardinalità di Y), dovrà valere $m \leq n$ affinché una soluzione possa esistere.

Per risolvere tale problema l'*Auction algorithm* prevede quindi di "simulare" lo svolgimento di un'asta nella quale a ciascun oggetto j viene associato un prezzo p_j (inizialmente 0) e le persone competono al rialzo per aggiudicarsi il lotto da cui poter ottenere il miglior profitto. In particolare l'algoritmo procede per round successivi, ciascuno suddiviso in due fasi distinte: la fase di offerta (o *bidding phase*) e quella di assegnamento (*assignment phase*). All'inizio della k-ma iterazione, ciascuna persona i non ancora assegnata ad alcun oggetto confronterà i prezzi correnti p_j con i valori che associa a ciascun lotto e proporrà un incremento $\delta(i)$ per l'oggetto j_i che ritiene più conveniente.

Formalmente, detti

$$N(i) = \{j \mid (i, j) \in E\}$$

$$j_i^k = \underset{j \in N(i)}{\operatorname{argmax}} \left(a_{ij} - p_j^k\right)$$

$$g_1^k(i) = \max_{j \in N(i)} \left(a_{ij} - p_j^k\right) = a_{ij_i^k} - p_{j_i^k}^k$$

$$g_2^k(i) = \begin{cases} \max_{j \in N(i),\ j \neq j_i^k} \left(a_{ij} - p_j^k\right) & \textit{se } |N(i)| > 1 \\ -\infty & \textit{altrimenti} \end{cases}$$

dove l'apice k indica il round, definiamo $\delta^k(i) = g_1^k(i) - g_2^k(i) + \epsilon$ come l'incremento di offerta sull'oggetto j_i^k che la persona i propone all'iterazione k-ma. Detto in altre parole la persona i al round k propone un rialzo sul prezzo dell'oggetto per cui avrebbe attualmente il maggior margine di guadagno, pari alla differenza tra i due maggiori guadagni possibili, più un valore ϵ positivo per evitare di avere incrementi nulli nel caso di pareggi.

Derivati tali valori, si conclude la fase di offerta e inizia quella di assegnamento in cui ogni oggetto viene assegnato al miglior offerente del round. In particolare, definendo $P^k(j)$ come l'insieme degli offerenti per j al round k, ogni oggetto j per il quale vale $P^k(j) \neq \emptyset$ è assegnato all'offerente $i_j^k = argmax_{i \in P^k(j),\ j_i^k = j}\ \delta^k(i)$, lasciando disaccoppiato l'eventuale possessore precedente, e il prezzo di j viene incrementato diventando $p^{k+1}(j) = p^k + \delta^k(i_j)$. L'algoritmo si conclude non appena tutte le persone risultino assegnate ad un oggetto.

Bertsekas (1979) dimostra che, nel caso in cui una soluzione esista, la terminazione di tale procedura avviene in un numero finito di passi grazie a due elementi. Per prima cosa il fatto che ogni oggetto, una volta che viene assegnato ad una qualche persona, non può più tornare ad essere libero, facendo quindi crescere in maniera monotona il numero di persone accoppiate. A ciò si unisce il costante incrementare dei prezzi per gli oggetti che ricevono almeno un offerta, aspetto che porta gli offerenti a considerarli via via meno vantaggiosi. Inoltre, si può verificare che nel caso in cui $a_{ij} \in \mathbb{N}$, utilizzare un valore $\epsilon < \frac{1}{min(m,n)}$ dove $m = |X|$ e $n = |Y|$, permette all'algoritmo di raggiungere una soluzione ottimale.

Nel frammento 1 si propone lo pseudocodice per l'*auction algorithm* in versione seriale, in cui si utilizzano una matrice A per rappresentare i pesi degli archi tra persone ed oggetti, utilizzando $-\infty$ per gli archi inesistenti, e diversi vettori per rappresentare i costrutti necessari. Rispetto a questi ultimi, si noti come, a differenza della descrizione precedente dove si utilizzava δ^k per indicare gli incrementi proposti dalle diverse persone all'iterazione k-ma, nello pseudocodice si è preferito mantenere tale informazione catalogata per oggetto nel vettore Δ_p, così da velocizzare la fase di assegnamento.

Algorithm 1 Auction algorithm, versione seriale

```
function AUCTION(A, m, n, ε)            ▷ m è il numero di persone, n quello degli oggetti
    peopleMatched ← 0, matching ← 0
    p ← 0                                                    ▷ prezzi per oggetto
    Δ_p ← 0                                     ▷ incrementi dei prezzi per oggetto
    roundBidders ← 0                     ▷ migliori offerenti per oggetto nel round
    while peopleMatched < m do
        UPDATE_ROUND_BIDS(p, matching, roundBidders, Δ_p, A, m, n, ε)
        UPDATE_MATCHING(Δ_p, roundBidders, p, matching, m, peopleMatched)
    return match

procedure UPDATE_ROUND_BIDS(p, matching, roundBidders, Δ_p, A, m, n, ε))
    Δ_p ← 0, roundBidders ← 0
    for person in {1..m} do
        if matching[person] = 0 then        ▷ solo le persone non accoppiate partecipano
            bid, bidTarget ← GET_PERSON_ROUND_BID(p, person, A, m, n, ε)
            if bid > Δ_p[bidTarget] then     ▷ salva solo info su bid massime per oggetto
                Δ_p[bidTarget] ← bid
                roundBidders[bidTarget] ← person

function GET_PERSON_ROUND_BID(p, person, A, m, n, ε)
    maxProfit ← −∞, sndMaxProfit ← −∞, target ← −1
    for object in {1..n} do
        if A[person, object] ≠ −∞ then             ▷ archi non esistenti hanno peso −∞
            profit = A[person, object] − p[object]
            if profit > maxProfit then
                sndMaxProfit ← MaxProfit
                maxProfit ← profit
                target ← object
            else if profit > sndMaxProfit then
                sndMaxProfit ← profit
    return (maxProfit − sndMaxProfit + ε), target
```

procedure UPDATE_MATCHING($\Delta_p, roundBidders, p, matching, n, peopleMatched$)
  for *object* in $\{1..n\}$ **do**
    if $\Delta_p[object] > 0$ **then** ▷ l'oggetto ha ricevuto almeno un incremento
      $p[object] \leftarrow p[object] + \Delta_p[object]$
      $previousHighestBidder \leftarrow person\ s.t.\ matching[person] = object$
      if $previousHighestBidder \leftarrow 0$ **then**
        $peopleMatched \leftarrow peopleMatched + 1$
      else
        $matching[previousHighestBidder] \leftarrow 0$
      $matching[roundBidders[object]] \leftarrow object$

2 Metodologia adottata per la soluzione

Per implementare una versione parallela si è partito dall'osservazione che nell'*auction algoritm* ogni round dipende strettamente dai risultati del precedente, richiedendo una successione seriale. Similmente ogni fase di assegnamento può essere eseguita solo in seguito alla conclusione della fase di offerta che la precede. Infatti, per poter poter determinare il miglior offerente per un certo oggetto è necessario attendere che tutte le persone non assegnate abbiano proposto i propri incrementi.

Nonostante ciò, il lavoro all'interno di ciascuna fase può essere parallelizzato in maniera abbastanza naturale, visto che ciascun offerente sceglie su che oggetto puntare in modo indipendente dagli altri, così come l'assegnamento dei lotti non condiziona i rimanenti. Di conseguenza si è pensato di suddividere il lavoro su due kernel principali, uno per ciascuna fase dell'algoritmo: *bidding kernel* e *assignment kernel*, delegando alla CPU il compito di avviare l'esecuzione di un round finché necessario.

Analogamente al precedente caso seriale, come input si considera di ricevere una matrice di pesi non negativi $A \in \mathbb{Z}^{m \times n}$, dove si utilizza un valore predefinito equiparabile a $-\infty$ per gli archi assenti, e il valore di ϵ. Inoltre, si utilizza una variabile intera, letta da *host* e scritta dal *device* per indicare il numero delle persone accoppiate in un certo round. La funzione ritorna quindi l'assegnamento trovato quando questo esiste.

L'algoritmo 2 descrive in maniera semplificata gli aspetti principali della proposta parallela sviluppata, cercando di trascurare gli aspetti più tecnici. Di seguito si analizzano più nel dettaglio le diverse funzioni, indicando eventuali modifiche implementative presenti nel codice e motivando le scelte effettuate.

Algorithm 2 Auction algorithm, versione parallela

function AUCTION_PARALLEL(A, m, n, ϵ)
  $pplMatched \leftarrow 0$
  while $pplMatched < m$ **do**
    $personBiddedForObj \leftarrow$ **False**
    kernel exec BIDDING($A, p, m, n, \epsilon, objRecBidFrom, bidInc$)
    kernel exec ASSIGNMENT($p, match, pplMatched, m, n, objRecBidFrom, bidInc$)
  return match

Internamente, la funzione "AUCTION_PARALLEL", utilizza diversi vettori e matrici per rappresentare le strutture dati utilizzate dall'algoritmo. La maggior parte di queste viene utilizzata esclusivamente dalla GPU e dunque viene allocata esclusivamente in VRAM. Fanno eccezione il contatore di persone correntemente accoppiate e il vettore corrispondente all'accoppiamento, entrambi interi, che devono essere inizializzati e letti da *host*. Per tali strutture viene quindi utilizzata della memoria *page-locked* così da velocizzare i trasferimenti. Oltre a ciò *host* inizializza un secondo array di interi di lunghezza n che rappresenta l'assegnamento dal punto di vista degli oggetti, utilizzato nell'implementazione della fase di assegnamento per questioni di efficienza (si veda linea 65 dello pseudo codice).

Riguardo al *bidding kernel*, visto che ciascuna persona non accoppiata deve considerare tutti i diversi oggetti per determinare il rialzo più vantaggioso, si è deciso di utilizzare delle riduzioni parallele lungo le righe della matrice dei valori A, diminuiti del prezzo corrente per ciascun oggetto in modo da considerare i profitti. In particolare, si è utilizzata una griglia bidimensionale di blocchi, nella quale la prima dimensione viene utilizzata nella riduzione di una singola persona, mentre la seconda permette di distinguere tra persone diverse. Rispetto allo pseudocodice, in sede implementativa si è preferito limitare il numero di blocchi lanciati, utilizzando dei valori multipli del numero di *Streaming Multiprocessor* (SM) della scheda, iterando sulla dimensione della griglia (*grid stride*) in entrambe le dimensioni per gestire istanze di dimensione arbitraria. Tale aspetto permette, inoltre, di gestire la riduzione lungo le righe di A mediante un numero determinato di fasi, come descritto nel seguito.

```
8:  procedure BIDDING(A, p, m, n, ϵ, objRecBidFrom, bidInc)
9:      for unmatched person in parallel on blockIdx.y do
10:         object ← blockIdx.x ∗ blockDim.x + threadIdx.x
11:         if object ≥ m then          ▷ si assume un numero adeguato di thread lungo l'asse x
12:             return
13:         if A[person, object] ≠ −∞ then
14:             profit ← A[person, object] − p[object]
15:             twoMaxPair ← (profit, −∞, object)  ▷ dato con campi max, sndMax e maxIdx
16:         else
17:             twoMaxPair ← (−∞, −∞, −1)
18:         twoMaxPair ← BLOCK_PARALLEL_REDUCTION_PAIR(twoMaxPair)
19:         if ( thenthreadIdx.x = 0)
20:             aux[person, blockIdx.x] = twoMaxPair
21:         all threads of the block are synchronized
22:         if blockIdx.x is the last block of the reduction then
23:             if threadIdx.x < gridDim.x then          ▷ assumiamo gridDim.x ≤ threadIdx.x
24:                 twoMaxPair ← aux[person, threadIdx.x]
25:             else
26:                 twoMaxPair ← (−∞, −∞, −1)
27:             twoMaxPair ← BLOCK_PARALLEL_REDUCTION_TWO_PAIR(twoMaxPair)
28:             if ( thenthreadIdx.x = 0)
29:                 bidInc ← twoMaxPair.max − twoMaxPair.sndMax + ϵ
30:                 objRecBidFrom[twoMaxPair.maxIdx, person] ← True
```

Nella realizzazione della riduzione parallela, che calcola sia i due valori massimi sia l'indice del massimo, corrispondente con l'oggetto per cui una persona incrementerà l'offerta, si è optato per uno schema in due fasi che aggregasse inizialmente le informazioni di ciascun blocco per poi far ripetere la riduzione ad un unico blocco così da ottenere il risultato finale. Al fine di svolgere entrambe nella stessa funzione, evitando il lancio di più kernel, si è adoperato un contatore ausiliario posto in memoria globale che indicasse per ogni persona il numero di blocchi che che avessero concluso la prima fase. Per la gestione di quest'ultimo si è utilizzata l'operazione atomica *atomicInc*, trattandosi di una variabile condivisa con scritture concorrenti.

Coerentemente con lo pseudo codice, BLOCK_PARALLEL_REDUCTION_TWO_PAIR utilizza internamente due chiamate a WARP_PARALLEL_REDUCTION_TWO_PAIR, sfruttando la memoria condivisa dal blocco per memorizzare i risultati intermedi. Avendo optato per l'utilizzo del tipo *double* per la gestione degli array relativi agli incrementi, così da limitare possibili errori dovuti all'aritmetica a virgola mobile, si è cercato di strutturare la *shared* in modo tale da prevenire l'insorgere di conflitti di banco. In particolare sono stati utilizzati diversi vettori di interi "spacchettando" e "re-impacchettando" i valori di *max* e *sndMax* tramite appositi costrutti forniti da CUDA.

Nella funzione WARP_PARALLEL_REDUCTION_PAIR si richiamano quindi più volte delle funzioni di *shuffle* per realizzare efficientemente la riduzione a livello dei *warp*.

```
function BLOCK_PARALLEL_REDUCTION_TWO_PAIR(twoMaxPair)
    lane ← threadIdx.x mod warpSize, wid = threadIdx.x div warpSize
    twoMaxPair ← WARP_PARALLEL_REDUCTION_TWO_PAIR(twoMaxPair)
    if lane = 0 then
        sharedAux[wid] = twoMaxPair                ▷ sharedAux vettore in memoria shared
    all threads of the block are synchronized
    if threadIdx.x < blockIdx.x div warpSize then
        twoMaxPair ← sharedAux[wid]
    else
        twoMaxPair ← (−∞, −∞, −1)
    if wid = 0 then
        twoMaxPair ← WARP_PARALLEL_REDUCTION_TWO_PAIR(twoMaxPair)
    return twoMaxPair

function WARP_PARALLEL_REDUCTION_PAIR(twoMaxPair)
    candidate ← (−∞, −∞, −1)
    for offset ← 1 to warpSize multiplying by 2 do
        candidate ← __shfl_xor_sync with offset for all twoMaxPair components
        if candidate.max > twoMaxPair.max then
            twoMaxPair.sndMax ← max(twoMaxPair.max, candidate.sndMax)
            twoMaxPair.max ← candidate.max
            twoMaxPair.maxIdx ← candidate.Idx
        else if candidate.max > twoMaxPair.sndMax then
            twoMaxPair.sndMax ← candidate.max
    return twoMaxPair
```

Una miglioria che dovrebbe permettere di bilanciare maggiormente il carico tra i diversi SM, aumentando al contempo la coalescenza negli accessi agli array ausiliari, utilizzati nel *bidding kernel* per memorizzare i risultati intermedi della riduzione parallela, consiste nell'utilizzo di una coda contenente le persone non accoppiate in un certo round. Tale struttura viene implementata attraverso un contatore *unmatchedPeople* e un array di interi di dimensione m dove le prime posizioni contengono gli indici delle persone attualmente disaccoppiate. Inizialmente queste corrispondono con l'intero insieme dei partecipanti all'asta, inizializzato da *host*, mentre successivamente la coerenza della struttura verrà mantenuta dai kernel sulla base dell'accoppiamento.

Terminata la fase di offerta, si passano i risultati sui rialzi e relativi target all'*assignment kernel* che avrà la responsabilità di aggiornare conseguentemente l'assegnamento e i prezzi degli oggetti. Similmente a quanto fatto nella fase di assegnamento anche questo kernel esegue una riduzione parallela per determinare il miglior offerente per ciascuno oggetto. In questo caso però si è preferito l'utilizzo di una griglia monodimensionale, sempre di dimensione multipla del numero di SM, dove ciascun blocco viene legato a oggetti distinti rispetto alla dimensione y

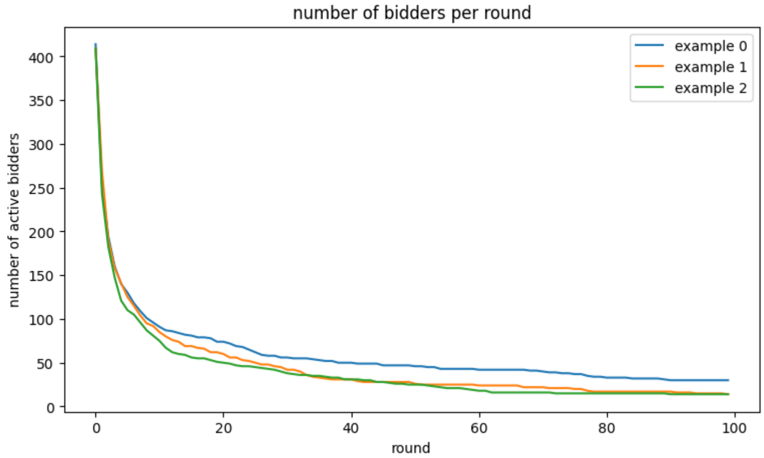

Figura 1: Numero di offerenti attivi durante i primi 100 round di alcune istanze contenenti 1108 persone ed altrettanti oggetti con densità 1.

(utilizzata al posto della x per una migliore distinzione semantica), mentre la riduzione avviene tra i thread dell'unico blocco sulla dimensione x.

Tale scelta è motivata dal fatto che, come si può notare dall'esempio riportato in figura 1, si è osservato che il numero di offerenti tende a decrescere in modo esponenziale al procedere dei round. Questo fenomeno fa sì che ciascun oggetto debba scegliere il miglior offerente tra un numero sempre minore di candidati, rendendo svantaggioso l'utilizzo di una riduzione in più passaggi.

In ogni caso, per garantire la gestione di un numero arbitrario di contendenti, si è utilizzata una fase preliminare di iterazione con *stride* pari alla dimensione x del blocco.

Analogamente a quanto fatto per l'*assignment kernel* anche in questo caso nell'implementazione si utilizza una coda per indicare quali oggetti abbiano ricevuto almeno un offerta e debbano quindi partecipare alla fase di assegnamento. La struttura dati scelta è sempre un array di interi, questa volta di dimensione n, associato ad un contatore, e viene gestita dal precedente kernel durante la fase di offerta.

```
procedure ASSIGNMENT(p, match, pplMatched, m, n, objRecBidFrom, bidInc)
    for unmatched object in parallel on blockIdx.y do
        maxPair ← (−∞, −1)                                  ▷ dato con campi max e maxIdx
        for person ← blockIdx.x ∗ blockDim.x + threadIdx.x with x-blockstride do
            if objRecBidFrom[object, person] then
                if bidInc[person] > maxPair.max then
                    maxPair ← (maxPair.max, maxPair.maxIdx)
        maxPair ← BLOCK_PARALLEL_REDUCTION_PAIR(maxPair)
        if thread.Idx = 0 then
            prevHighestBidder ← person s.t. match[person] = object
            if prevHighestBidder = −1 then
                atomicAdd(pplMatched, 1)
            else
                match[previousHighestBidder] ← −1
            p[object] ← p[object] + maxPair.max
            match[object] ← maxPair.maxIdx
```

BLOCK_PARALLEL_REDUCTION_PAIR e WARP_PARALLEL_REDUCTION_PAIR, come indicato anche nello pseudo codice, sono funzioni analoghe a quelle utilizzate nella fase di offerta, presentando le stesse peculiarità implementative e per tale ragione il loro pseudocodice viene omesso. La differenza principale sta nel fatto che, in questo caso, le strutture gestite comprendono unicamente l'informazione sul valore massimo e sul suo indice, corrispondenti rispettivamente al miglior rialzo e al relativo offerente.

---

function BLOCK_PARALLEL_REDUCTION_PAIR(MaxPair)
    analogous to BLOCK_PARALLEL_REDUCTION_TWO_PAIR considering
    just the *max* and *maxIdx* components, calls WARP_PARALLEL_REDUCTION_PAIR
    similarly to WARP_PARALLEL_REDUCTION_TWO_PAIR

Terminata la fase di riduzione, un unico thread per blocco si occupa di alzare il prezzo dell'oggetto al quale è legato e di aggiornare il vettore *match*, incrementando atomicamente il contatore di persone assegnate quando l'oggetto diventi assegnato per la prima volta. Alternativamente, il precedente possessore rimane disaccoppiato e viene inserito nella coda di offerenti per il round successivo.

Per completare tale coda, aggiungendo le persone attive che non si sono aggiudicate alcun oggetto viene quindi utilizzato un ultimo kernel, non riportato nello pseudocodice, che scandisce parallelamente la coda (vettore) degli offerenti del round corrente alla ricerca di quelli non assegnati.

Quest'ultima esecuzione può essere svolta in parallelo con la copia del contatore di persone assegnate, da *device* a*host*, così come accade tra l'inizializzazione di alcune strutture dati e l'esecuzione dei kernel. Per tale ragione l'implementazione proposta prevede l'utilizzo di più *stream* diversi e la loro sincronizzazione tramite eventi. A tal proposito, la figura 2 descrive schematicamente la ripartizione del lavoro all'interno di un singolo round con le relative dipendenze.

Oltre a ciò, visto che l'algoritmo prevede un l'esecuzione di svariati round con la medesima struttura, si è realizzata una variazione del codice che utilizza il costrutto dei *CUDA graph* in modo da provare a ridurre l'overhead presente nel lancio dei kernel.

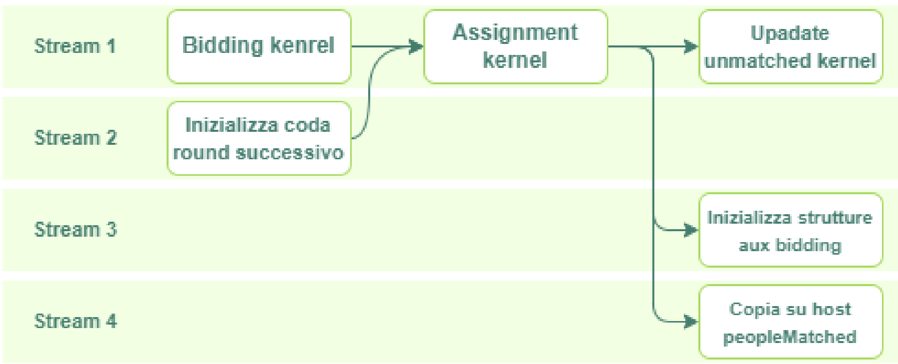

Figura 2: Suddivisione dell'esecuzione di un round tra stream diversi per l'implementazione parallela.

Tabella 1: Specifiche del sistema utilizzato per eseguire i test.

O.S.	Ubuntu 22.04.1 LTS
CPU	i5 4460 @ 3.2GHz
RAM	2x4GB DDR3 1866 MHz
GPU	GeForce GTX 960 @ 1.25GHz
Architettura	Maxwell
C.C.	5.2
SM	8
VRAM	1994 MB
versione CUDA	11.6.112

3 Risultati sperimentali

Per testare il comportamento della soluzione proposta si sono eseguiti diversi test, confrontando la soluzione seriale con quella parallela in più varianti. In particolare, sono state considerate le seguenti versioni:

- non deterministica, che presenta le caratteristiche descritte nella sezione precedente. Di conseguenze, nel caso in cui più offerenti propongano il medesimo rialzo d'offerta per uno stesso oggetto o quando più oggetti presentino un uguale profitto, l'accoppiamento e la scelta dell'obiettivo del rialzo avviene in maniera non deterministica a causa della variabilità insita nella schedulazione dei blocchi;

- una variante deterministica del precedente codice che si comporta come l'algoritmo seriale rispetto al caso di pareggi tra più offerenti, ossia assegnando un oggetto alla persona con indice minore tra quelle che propongono il rialzo maggiore in un certo round;

- una versione del primo codice che sfrutta i *CUDA graph* per l'esecuzione.

In tutti i casi si sono generate in maniera pseudo casuale le matrici dei pesi (positivi) per grafi bipartiti, assicurandosi di utilizzare un *seed* di generazione noto e garantendo per costruzione la presenza di un assegnamento tra persone ed oggetti. Per fare ciò si è generata per ciascuna istanza una permutazione casuale degli oggetti, utilizzandola per definire un assegnamento iniziale e successivamente si sono riempiti i valori dei pesi mancanti in base alla densità desiderata. Tali istanze, salvate in binario su file, sono state fornite in input agli algoritmi d'asta seriali e paralleli, misurando i tempi di esecuzione attraverso gli eventi CUDA.

I test sono stati eseguiti su una macchina con le specifiche riportate in tabella 1.

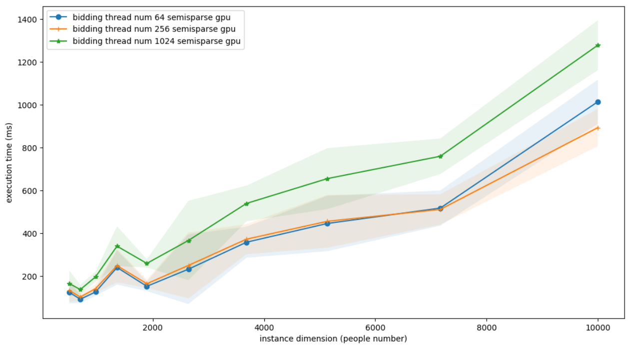

Figura 3: Tempi di esecuzione della versione non deterministica per diversi valori del numero di thread per il bidding kernel

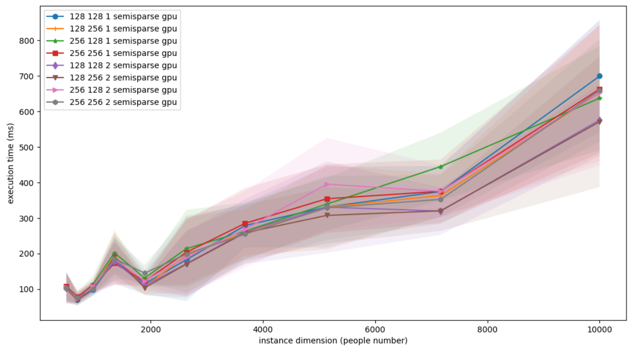

Figura 4: Tempi di esecuzione della terza versione parallela con diversi numero di thread e di blocchi

Durante lo sviluppo si sono considerate diversi valori per il numero dei thread e le dimensioni dei blocchi lanciati. Ad esempio nella figura 3 si sono valutati i valori 64, 256 e 1024 per il

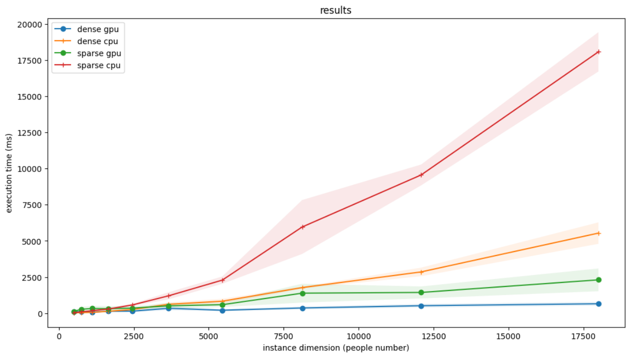

Figura 5: Tempi di esecuzione della versione non deterministica contro quella seriale

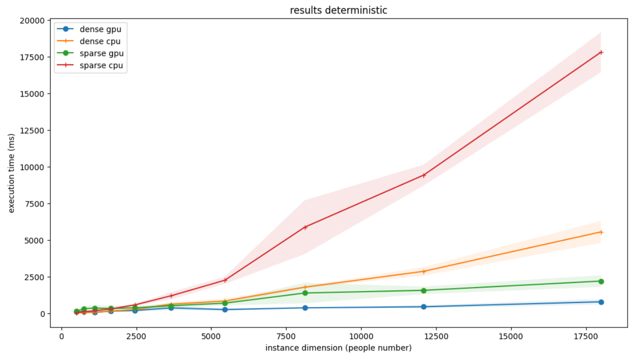

Figura 6: Tempi di esecuzione della versione parallela deterministica contro quella seriale

numero di thread da utilizzare nel lancio del *bidding kernel*, mentre la figura 4 rappresenta i risultati ottenuti utilizzando i *CUDA graph* e variando il numero di thread di entrambe le fasi dell'*auction algorithm* così come la dimensione y della griglia. In particolare, per quest'ultimo parametro si sono considerati valori pari al numero di SM della scheda oppure al doppio di esso.

In entrambi i casi i test hanno considerato delle istanze con un numero variabile di persone, equivalente al numero di oggetti, ed una densità al 50%. Per ciascuna dimensione si sono generate casualmente 3 istanze, considerandone i tempi medi d'esecuzione e utilizzando un valore di ϵ pari a 1.

Si è dunque scelta la configurazione migliore, che utilizza 128 thread per il kernel d'offerta e 256 per quello d'assegnamento, impostando un valore doppio rispetto al numero di SM per la dimensione y delle relative griglie. Tale configurazione viene quindi sottintesa nei risultati seguenti.

Per valutare la differenza di performance tra le versioni parallele si sono eseguiti dei test al variare del numero di persone, utilizzando lo stesso numero di oggetti e due diversi valori di densità: 10% per ottenere un grafo sparso e 100% per uno denso. In particolare si sono valutati 10 valori per m a partire da 500 e fino ad arrivare a 18000 seguendo una progressione geometrica, utilizzando 3 istanze per dimensione e un valore di ϵ pari a 1. Inoltre, per limitare il numero di passi necessari al raggiungimento della convergenza, aspetto che come verrà evidenziato in seguito risulta di particolare importanza, si è deciso di considerare solamente pesi nell'intervallo [1,10000].

Come si può osservare dai risultati riportati nelle figure 6, 5 e 8, riportati in forma numerica

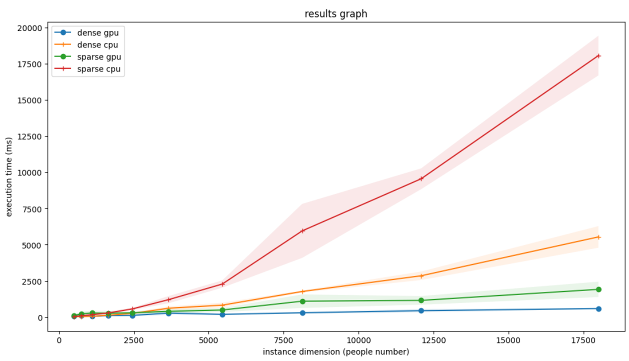

Figura 7: Tempi di esecuzione della versione con i CUDA graph contro quella seriale

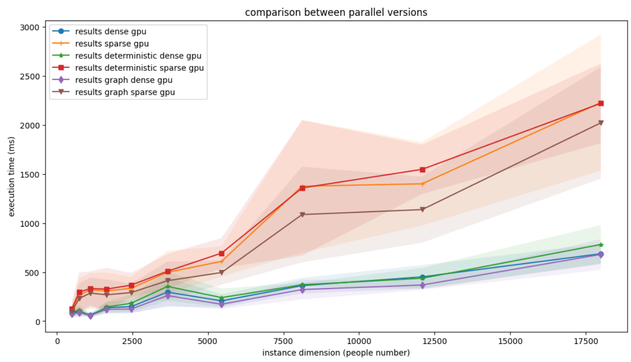

Figura 8: Confronto sui tempi di esecuzione delle varianti parallele

nella sezione B, i tempi di esecuzione della versione seriale crescono più rapidamente dei quelli delle versioni parallele che, al contrario, scalano meglio all'aumentare della dimensione dell'istanza. In particolare, si può notare che in entrambi i casi le performance risultano migliori per i grafi densi, con una differenza notevole per la versione seriale (peggioramento di 7 volte circa sull'istanza maggiore).

Tale comportamento può essere spiegato guardando al numero di round svolti in media al variare della dimensione dell'istanza. Come si evince dalla figura 9, infatti, le istanze con una densità minore richiedono sempre un numero di round inferiore per il raggiungimento di una configurazione finale, riducendo il tempo di esecuzione dell'algoritmo.

Il numero di iterazioni in generale costituisce un fattore determinante per la bontà dei tempi di esecuzione. Ciò può essere constatato guardano ai risultati riportati nella figura 10, riguardanti nuovamente il confronto dell'algoritmo seriale con quello parallelo, sulle medesime istanze. In questo caso però si è utilizzato un valore di ϵ pari a $\frac{1}{m+1}$ con m numero delle persone, che garantisce il raggiungimento di un assegnamento ottimale.

Tale modifica ha portato ad un significativo aumento dei tempi di esecuzione per entrambi gli algoritmi, risultando in una interruzione prematura dei test a causa del raggiungimento di un timeout impostato preventivamente. Inoltre, rispetto ai casi precedenti si può osservare un inversione di tendenza rispetto alla densità delle istanze. Ancora una volta la motivazione di tale comportamento può essere osservata guardando al numero di round richiesti dall'esecuzione. Infatti in questo caso la crescita di questi ultimi diviene esponenziale (si noti la scala logaritmica per l'asse delle ascisse), causando un degrado significativo dei tempi d'esecuzione.

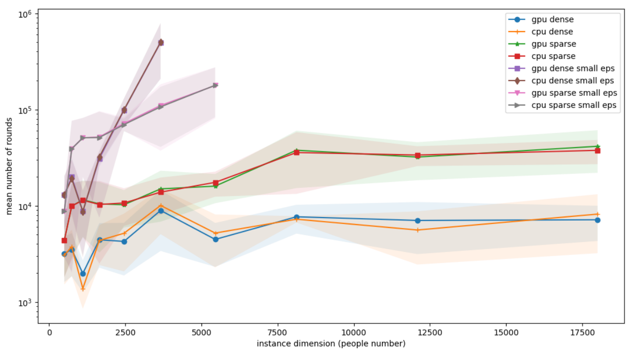

Figura 9: Numero di round eseguiti in media al variare della dimensione delle istanze per gli algoritmi seriale e parallelo non deterministico con diversi valori di ϵ

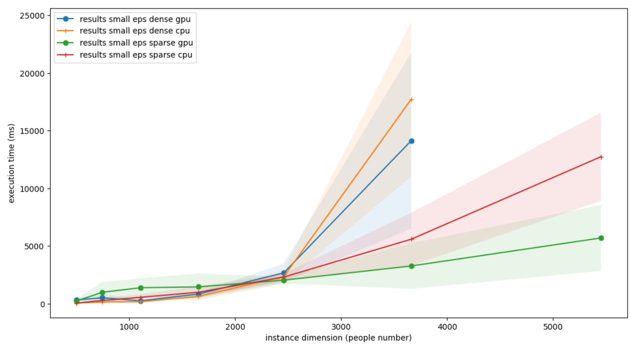

Figura 10: Confronto sui tempi di esecuzione delle varianti parallele non deterministiche con valori di ϵ sufficientemente piccolo per garantire l'ottimalità

4 Osservazioni finali

Il confronto con tra le diverse versioni ha permesso di osservare come l'implementazione parallela risulti essere in generale una scelta adeguata per l'*auction algorithm*, riuscendo a superare sempre le prestazioni della versione seriale, fatta eccezione per le istanze di dimensione minore. Tuttavia tale approccio presenta anche alcuni svantaggi come la maggior difficoltà implementativa e la necessita dell'impiego di un dispositivo di computazione ausiliario.

Riguardo alle diverse implementazioni proposte per l'algoritmo parallelo, le differenze pratiche sono risultate minime, facendo preferire le versioni non deterministiche per la maggiore semplicità del codice associata, unita alla minor probabilità di ricadere in casi "sfortunati".

Nonostante l'*auction algorithm* garantisca di raggiungere una soluzione ottimale nel caso di un ϵ sufficientemente piccolo, si è osservato come la riduzione di tale parametro causi un significativo aumento dei tempi di computazione a causa dell'elevato numero di iterazioni necessarie alla convergenza. Per tale ragione nei test si è preferito l'utilizzo di un valore unitario per ϵ che può rappresentare un buon compromesso qualora ci si possa accontentare si una soluzione sub-ottimale. In generale a seconda dello specifico scenario si potrebbe pensare di agire su tale parametro partendo da un valore anche più elevati, riducendolo fino al raggiungimento di un risultato soddisfacente.

L'approccio sviluppato presenta comunque alcune limitazioni ed inefficienze, come la serializzazione dell'esecuzione di round diversi e l'assenza di sovrapposizione nell'esecuzione dei kernel. Sebbene tali questioni siano in parte dovute alla struttura dell'algoritmo, potrebbe essere vantaggioso indagare l'utilizzo di strutture dati alternative che permettano di limitare o sovrapporre almeno parzialmente il lavoro svolto nella fase di offerta, osservando che i profitti riguardanti gli oggetti non modificati durante la fase di assegnamento rimangono costanti rispetto al round precedente.

Di seguito viene infine fornita una breve guida all'utilizzo dei principali eseguibili utilizzati.

A Miniguida all'utilizzo

Per compilare i sorgenti viene fornito un $Makefile$ che permette di compilare singolarmente o cumulativamente i file per l'esecuzione dell'*auction algorithm*. In particolare l'opzione di default permette di ottenere gli eseguibili `gpu_auction`, `gpu_auction_deterministic`, `gpu_auction_graph` e `cpu_auction`, corrispondenti con le tre versioni parallele e con quella seriale. Interfaccia per l'utilizzo di questi è la medesima, permettendone l'utilizzo in modo *standalone* secondo le seguenti modalità:

- fornendo all'eseguibile il nome di un file binario contenete m, n e una matrice intera $m \times n$, rappresentante i pesi del grafo, e un valore per ϵ si possono ottenere in output l'assegnamento trovato insieme ai prezzi finali di ciascun offerente e al costo totale della soluzione;

- utilizzando gli argomenti `-generate m n d eps [seed]` è possibile specificare gli omonimi parametri, dove `d` indica la densità degli archi, per generare in maniera pseudo casuale un istanza con pesi interi limitati a 10000, salvata in formato sia testuale che binario, su cui viene eseguito l'algoritmo ritornando i risultati analogamente al caso precedente.

Per generare delle istanze in blocco a partire da un file di configurazione è possibile richiamare l'eseguibile `test_case_generator` che salva i risultati in file binari posti nella cartella corrente o nella destinazione indicata come secondo parametro. In questo caso i file di configurazione devono presentare la seguente struttura:

- `SEED [LIMIT]` sulla prima riga per determinare il seed di generazione e opzionalmente un limite per la dimensione massima dei pesi da utilizzare (altrimenti viene scelto il valore massimo possibile);

- una serie di righe, una per istanza, con formato `m n density` per indicare i corrispondenti parametri di inizializzazione.

Per generare i casi di test ed eseguire gli algoritmi così da ottenere i risultati riportati nella sezione B discussi precedentemente, è possibile richiamare in sequenza i comandi `make generate_testcases` (utilizzerà approssimativamente 14 GB di spazio su disco) e `make` `run_tests` per avviare la computazione. I risultati, vengono salvati nella cartella corrente in dei file con formato *json*.

Per la compilazione degli eseguibili in formato *standalone* viene richiesta la presenza del compilatore cuda `nvcc` mentre per eseguire i test è necessario avere installato Python in versione 3 invocabile attraverso il comando `python3`.

B Tabelle dati

Tabella 2: Tempi di esecuzione medi e deviazione standard per i diversi algoritmi proposti

numero	tempi medi di esecuzione (ms)											
offerenti	dense gpu	dense cpu	sparse gpu	sparse cpu	det. dense gpu	det. sparse gpu	graph dense gpu	graph sparse gpu	small eps. dense gpu	small eps. dense cpu	small eps. sparse gpu	small eps. sparse cpu
500	85.00	20.65	117.37	31.48	92.94	128.68	73.74	101.06	332.91	67.15	235.50	53.56
	(37.54)	(5.34)	(58.07)	(6.26)	(42.71)	(67.40)	(32.33)	(51.33)	(170.34)	(25.50)	(137.29)	(15.53)
744	97.97	40.67	265.52	93.99	112.45	296.72	86.62	231.96	526.87	155.06	1001.40	283.08
	(41.19)	(9.58)	(172.16)	(40.66)	(50.99)	(200.90)	(36.46)	(154.01)	(284.19)	(55.82)	(896.04)	(204.67)
1108	64.73	51.04	320.12	169.61	52.81	332.83	57.80	286.57	256.91	185.28	1395.63	565.46
	(19.74)	(11.59)	(171.10)	(57.40)	(15.88)	(174.42)	(14.00)	(155.68)	(150.48)	(89.25)	(804.18)	(263.96)
1650	137.25	132.21	308.50	288.21	143.84	327.17	120.71	269.00	842.73	635.01	1467.23	1000.44
	(56.77)	(26.90)	(185.74)	(97.78)	(54.01)	(217.55)	(40.80)	(155.25)	(137.65)	(76.98)	(1161.82)	(528.70)
2458	148.29	254.13	335.39	571.08	182.95	367.07	124.06	291.47	2681.10	2367.75	2049.85	2317.74
	(65.22)	(60.48)	(114.21)	(78.00)	(87.58)	(123.70)	(42.32)	(89.15)	(796.02)	(561.92)	(286.46)	(280.92)
3660	296.05	618.13	498.15	1208.28	354.84	508.78	260.63	412.66	14135.33	17722.16	3278.19	5596.85
	(148.00)	(124.18)	(220.84)	(245.63)	(146.66)	(176.32)	(103.46)	(193.53)	(7620.05)	(6690.81)	(1973.56)	(2286.47)
5451	207.40	838.33	609.23	2306.07	240.65	693.91	172.92	496.16			5707.31	12747.94
	(66.80)	(177.88)	(153.59)	(250.25)	(91.84)	(153.86)	(47.32)	(122.28)			(2863.20)	(3816.37)
8117	362.77	1783.74	1375.81	5988.42	371.49	1358.39	322.60	1087.09				
	(78.52)	(78.09)	(676.95)	(1855.90)	(18.88)	(689.89)	(99.48)	(486.54)				
12087	451.84	2885.70	1400.44	9545.33	437.07	1547.90	368.84	1137.45				
	(119.83)	(297.15)	(424.43)	(738.69)	(106.00)	(251.25)	(48.86)	(336.49)				
18000	687.58	5541.21	2228.96	18053.06	781.46	2220.35	680.20	2021.77				
	(103.78)	(741.21)	(693.61)	(1392.17)	(196.09)	(405.54)	(155.95)	(568.53)				

Tabella 3: Numero di round medi e deviazione standard per i diversi algoritmi proposti

numero	Numero medio di round per la convergenza											
offerenti	dense gpu	dense cpu	sparse gpu	sparse cpu	det. dense gpu	det. sparse gpu	graph dense gpu	graph sparse gpu	small eps. dense gpu	small eps. dense cpu	small eps. sparse gpu	small eps. sparse cpu
500	3176.87	3071.33	4372.33	4364.33	3071.33	4364.33	3163.73	4372.33	12983.27	12974.00	8825.00	8773.67
	(1558.82)	(1552.49)	(2552.93)	(2557.94)	(1552.49)	(2557.94)	(1526.89)	(2552.93)	(7189.66)	(7424.21)	(5829.74)	(5866.24)
744	3507.33	3737.00	9935.13	9940.67	3737.00	9940.67	3571.80	9922.07	19938.93	19108.33	39266.60	39119.33
	(1662.54)	(1873.35)	(7039.40)	(7189.57)	(1873.35)	(7189.57)	(1682.33)	(7091.79)	(11491.31)	(11107.58)	(37265.98)	(37047.20)
1108	1979.93	1377.67	11520.53	11397.33	1377.67	11397.33	2027.13	11580.00	8743.67	8699.33	50844.73	50865.00
	(714.35)	(526.72)	(6794.83)	(6637.62)	(526.72)	(6637.62)	(593.52)	(6869.16)	(5668.91)	(5798.44)	(30931.16)	(30892.46)
1650	4425.67	4353.00	10452.33	10308.00	4353.00	10308.00	4487.33	10528.87	30853.53	32172.33	51902.67	51202.67
	(2154.68)	(1932.48)	(7188.58)	(7801.87)	(1932.48)	(7801.87)	(1763.11)	(6859.08)	(5291.65)	(5270.22)	(44364.14)	(43619.23)
2458	4256.67	5170.33	10241.33	10676.33	5170.33	10676.33	4168.73	10541.07	98341.80	99424.00	71565.07	69787.33
	(2373.14)	(3090.82)	(4299.81)	(4504.33)	(3090.82)	(4504.33)	(1843.62)	(4061.93)	(30987.04)	(31197.76)	(11296.83)	(10297.77)
3660	8935.07	10066.67	14965.73	13846.33	10066.67	13846.33	9234.07	14501.40	497157.07	502452.33	110249.27	106652.00
	(5543.74)	(4992.63)	(8134.88)	(5763.02)	(4992.63)	(5763.02)	(4502.62)	(8401.92)	(288563.89)	(292669.05)	(72795.18)	(65676.08)
5451	4465.53	5199.33	15995.80	17505.33	5199.33	17505.33	4374.47	15702.53			177905.27	178896.00
	(2148.99)	(2910.33)	(5355.98)	(5076.97)	(2910.33)	(5076.97)	(2003.38)	(5367.41)			(96852.25)	(95227.85)
8117	7676.47	7244.33	37770.93	35732.33	7244.33	35732.33	8457.73	36962.40				
	(2541.58)	(543.67)	(22505.28)	(22439.67)	(543.67)	(22439.67)	(4305.18)	(21007.62)				
12087	7030.67	5592.33	32082.73	33686.33	5592.33	33686.33	5680.53	30132.13				
	(3883.93)	(3142.12)	(13655.47)	(7832.03)	(3142.12)	(7832.03)	(1954.80)	(14260.18)				
18000	7145.13	8186.00	41501.73	37676.67	8186.00	37676.67	8258.53	42904.80				
	(2837.96)	(4972.83)	(19470.77)	(10612.75)	(4972.83)	(10612.75)	(4945.32)	(18815.52)				

Riferimenti bibliografici

Bertsekas, D. P. (1979). A distributed algorithm for the assignment problem. *Lab. for Information and Decision Systems Working Paper, MIT.*

Bertsekas, D. P. e Castanon, D. A. (1989). The auction algorithm for the transportation problem. *Annals of Operations Research*, 20(1):67–96.

Vasconcelos, C. N. e Rosenhahn, B. (2009). Bipartite graph matching computation on gpu. In *Energy Minimization Methods in Computer Vision and Pattern Recognition: 7th International Conference, EMMCVPR 2009, Bonn, Germany, August 24-27, 2009. Proceedings 7*, pages 42–55. Springer.