Parallelizzazione dell'algoritmo di belief propagation tramite CUDA.

Alessandro Gottoli - VR352595
Fabio Pettenuzzo - VR362044

3 marzo 2014

Indice

1 Introduzione

In questa relazione andremo ad illustrare come abbiamo parallelizzato l'algoritmo di belief propagation su GPGPU. Per ragioni di brevità ci concentreremo solo sulle fasi che abbiamo parallelizzato, delegando una più approfondita descrizione dell'algoritmo ad un'apposita relazione.

Per il momento è sufficiente sapere che si tratta di un algoritmo *message passing* che lavora su *junction tree*, cioè una rappresentazione ad albero delle *reti bayesiane*, e ci permette di calcolare i marginali di ogni cricca non compromettendo il calcolo della probabilità congiunta dell'intera rete e in caso di nuova evidenza di propagarla facilmente.

Ogni nodo del junction tree è una cricca massimale delle variabili della rete. Gli archi sono chiamati separatori e nella nostra implementazione vengono individuati tramite l'algoritmo di *kruskal* calcolando l'albero di copertura massimo. Il peso di ogni arco è dato dal numero di variabili che hanno in comune le cricche che collega e l'algoritmo cerca l'albero di peso massimo per rispettare la proprietà di *running intersection*.

Dopo questa fase di trasformazione l'algoritmo associa la tabella di probabilità condizionale di ogni variabile della rete ad una cricca che contiene tutte le variabili che compaiono nella tabella. Moltiplicando tra loro le tabelle associate alla stessa cricca si ottiene una tabella che identifica la tabella dei potenziali per le variabili della cricca e ogni loro configurazione. Queste tabelle vengono identificate come ψ. Anche i separatori hanno la loro tabella formata da tutte le configurazioni che possono assumere le variabili in esso contenuto, e inizialmente vengono inizializzate a 1. Queste tabelle le identificheremo come ϕ. I messaggi che vengono passati tra due nodi sono proprio le informazioni necessarie per aggiornare queste tabelle. Le due diverse tabelle da aggiornare compongono le due fasi che abbiamo deciso di parallelizzare.

Supponiamo di aggiornare la tabella di ψ_2 partendo dalla tabella del suo vicino ψ_1. La notazione che useremo sarà '*' per identificare la nuova tabella aggiornata. L'aggiornamento della tabella del separatore ϕ_{1-2} viene chiamata *marginalizzazione* e si ottiene facendo la somma di tutti i valori della tabella ψ_1 che hanno la stessa configurazione per le variabili del separatore.

$$\phi^*_{1-2} = \sum_{vars_{\psi_1}/vars_{\phi_{1-2}}} \psi_1$$

L'aggiornamento della tabella ψ_2 viene chiamata *scattering* e si ottiene moltiplicando la stessa ψ_2 per il rapporto tra la tabella del ϕ^*_{1-2} appena calcolata e quella vecchia ϕ_{1-2}.

$$\psi^*_2 = \psi_2 \frac{\phi^*_{1-2}}{\phi_{1-2}}$$

Queste operazioni si prestano abbastanza bene ad ad essere parallelizzate, in quanto il prodotto del numero di stati che possono assumere le variabili all'interno della stessa cricca può raggiungere dimensioni elevate. Seguendo l'articolo [1] abbiamo deciso di parallelizzare sulla dimensione del separatore, dato che ogni elemento della cricca è associato ad una sola configurazione del separatore.

Dato che gli elementi non sono ordinati per favorire queste operazioni si è resa necessaria una fase di preprocessing, che viene applicata in entrambe le versioni

(sequenziale e parallela). Questa fase consiste nel creare in ogni separatore delle tabelle degli indici associate alle tabelle dei potenziali delle cricche collegate dal separatore stesso. Le tabelle degli indici sono state costruite in modo da sfruttare la *coalescenza della memoria*, ossia permettendo a thread aventi id adiacenti di accedere ad indirizzi di memoria adiacenti. Pertanto gli indici dei valori che vengono sommati sono posizionati sempre ad una distanza pari alla dimensione del separatore.

In fase di inizializzazione viene scelta la scheda avente maggior capacità computazionale.
Si noti che solo le schede dotate di capacità computazionale maggiore o uguale a `2.0` (cioè a partire dalla generazione Fermi) supportano efficientemente il tipo di dato `double`, garantendo prestazioni che sono solo la metà rispetto a quelle ottenibili con i valori in singola precisione.

Prima di lanciare ogni kernel viene invocata la funzione `cudaDeviceSynchronize()`, in modo da evitare conflitti con eventuali altri utilizzatori della macchina.
Inoltre a seguito di ogni funzione CUDA viene invocata la funzione `cudaGetLastError()` per rilevare eventuali errori. Queste funzioni possono comportare un leggero aggravio in termini prestazionali, ma forniscono più garanzie circa la correttezza del codice.

Di seguito analizzeremo le scelte progettuali effettuate supponendo di invocare l'aggiornamento della tabella ψ_2 a partire dai dati della tabella ψ_1 collegata dal separatore con tabella ϕ_{1-2}.

2 Marginalizzazione

L'input della procedura è rappresentato da:

- ψ_1: la matrice che rappresenta la tabella dei potenziali. È composta da valori `double` ed ha dimensione $n \cdot m$, dove n è la dimensione del separatore, cioè la dimensione dell'output che otterremo sommando i valori oppurtuni e che sarà ϕ^*_{1-2}. Essa è rappresentata come un array di elementi.

- tabellaIndici di ψ_1: una matrice di valori `unsigned int`, anch'essa di dimensione $n \cdot m$ e rappresentata come un array di elementi. Questa tabella si rende necessaria per non dover ricercare in ψ_1 gli elementi sa sommare tra loro, e quindi contiene in ogni cella l'indice della tabella dei potenziali del valore di interesse.

Un generico elemento in posizione (i, j), dove $i \in [0...m]$ e $j \in [0...n]$, è pertanto identificato da:
$\psi_1[$ tabellaIndici$[\ i \cdot n + j\ ]\ ]$
Fanno eccezione gli indici aventi come valore `SIZE_MAX`. Questi ultimi vengono utilizzati per indicare che in quella posizione non esiste alcun dato da elaborare. Il loro utilizzo sarà più chiaro quando illustreremo le procedure per calcolare la riduzione su GPGPU, per il momento è sufficiente sapere che, per motivi legati all'efficienza, entrambe le tabelle in input e l'array di output hanno una dimensione che è potenza di 2. Pertanto, nel caso in cui i dati effettivi non siano una potenza di 2, le tabelle saranno dimensionate secondo la potenza di 2 successiva e il valore `SIZE_MAX` servirà per contraddistinguere i dati in eccesso, che non verranno presi in esame durante i calcoli.

L'output della procedura è un array $result$ di dimensione n contenente in ogni cella $j \in [0...n]$:

$$result[j] = \sum_{i=0}^{m} \psi_1[tabellaIndici[i \cdot n + j]] \tag{1}$$

Quest'operazione può essere vista come un caso particolare di un'operazione di riduzione dove il risultato ottenuto non è un unico valore contenente la somma di tutti i valori dell'array di partenza, ma un array contenente la somma di alcuni valori, secondo lo schema precedentemente descritto.

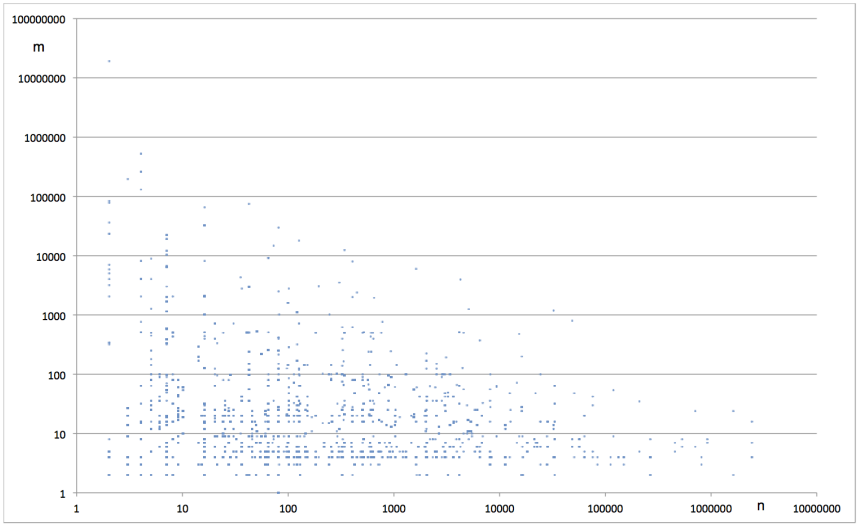

Figura 1: L'asse delle ascisse rappresenta i valori di n, mentre l'asse delle ordinate rappresenta i valori di m. Entrambi i valori sono su scala logaritmica e relativi ai benchmark su cui sono stati effettuati i test.

Il metodo è stato implementato da zero in quanto la gestione degli indici non permetteva l'utilizzo di librerie esterne. Purtroppo le dimensioni dei dati (riportate in Figura 1 e 2) sono estremamente variabili:

- vi sono matrici che presentano una delle due dimensioni molto più grande rispetto all'altra
- vi sono matrici che presentano un buon bilanciamento tra le due dimensioni
- infine molte matrici presentano dimensioni che rendono sconveniente la parallelizzazione basata su GPGPU.

Si rende pertanto necessario uno sviluppo di più approcci, e una precisa definizione dei casi di utilizzo. Di seguito descriveremo gli approcci seguiti, delineando successivamente in che contesto sono risultati più vantaggiosi.

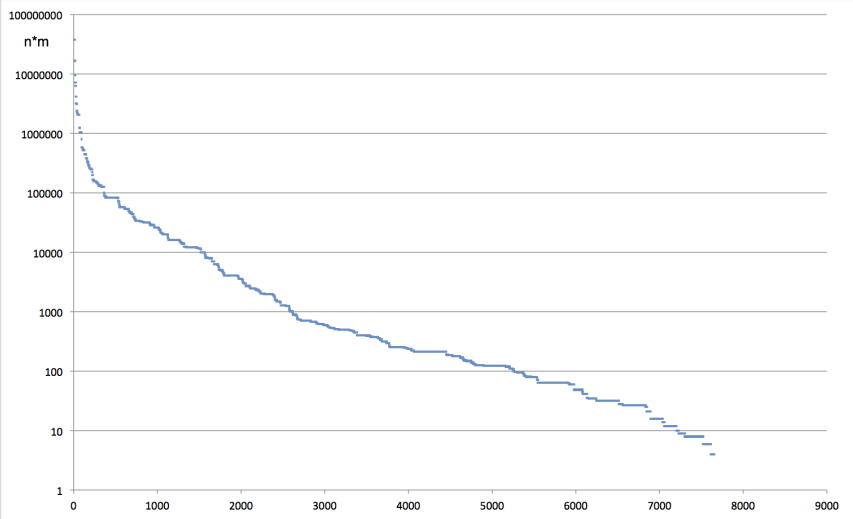

Figura 2: L'asse delle ordinate rappresenta, per ogni matrice dei benchmark su cui sono stati effettuati i test, la sua dimensione totale secondo una scala logaritmica.

2.1 Procedura che parallelizza su m

Il primo approccio vuole andare a coprire i casi in cui n sia un numero piccolo mentre m sia un numero grande. Questo caso è simile ad una riduzione classica, rappresentata dal caso particolare in cui n sia 1.

L'elaborazione si compone di due fasi, gestite da due kernel appositi:

2.1.1 Fase 1

In una prima fase, schematizzata brevemente in Figura 3, ogni blocco di thread (di dimensione `blockDim`) legge dalla *memoria globale* un numero doppio di valori rispetto alla sua dimensione. Infatti al suo interno ogni thread legge due valori, posti a distanza `blockDim`, e ne scrive la somma in una cella di un array allocato nella *memoria condivisa*. Il tutto avviene utilizzando la tabellaIndici come sopra descritto.

Così facendo viene effettuata una prima fase di riduzione durante la lettura dalla *memoria globale*, ottimizzando l'utilizzo della *memoria condivisa*. Questa risorsa è preziosa in quanto la riduzione ha una ridotta intensità aritmetica (1 flop per ogni elemento caricato) e le sue prestazioni sono pertanto limitate dalla larghezza di banda disponibile. Si noti che ogni valore `double` occupa 8 Byte, quindi ogni multiprocessore può gestire nella *memoria condivisa* al massimo 6144 valori. Inoltre, se non venisse fatta quest'operazione, metà delle threads presenti in ogni blocco verrebbero utilizzate solo per fare una lettura in *memoria globale*.

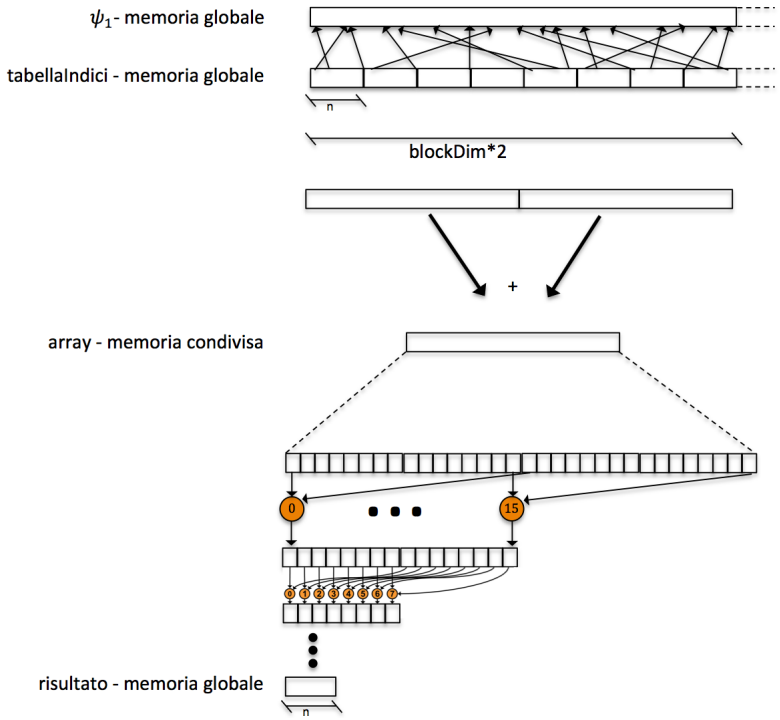

Figura 3: Schema relativo alla marginalizzazione, parallelizzata in funzione di m (fase 1)

Una volta caricate le somme parziali nella *memoria condivisa* il kernel entra in un ciclo il quale, tramite uno shift a destra, dimezza ad ogni iterazione gli elementi da elaborare e le corrispondenti threads. Ad ogni iterazione ogni thread legge un valore presente nella prima metà degli elementi (corrispondente al proprio `threadId`) e un valore nella seconda metà, sfruttando quindi la *coalescenza della memoria*. Così facendo non si ha *divergenza*, in quanto tutte le thread attive in un blocco hanno id consecutivi e vengono evitati i conflitti tra gli accessi ai banchi di *memoria condivisa*. Quando il numero di elementi da elaborare è pari a n i valori vengono scritti in *memoria globale*. Per evitare sovrapposizioni tra i risultati scritti da blocchi diversi la scrittura avviene utilizzando un offset dipendente dall'id del blocco in oggetto.

2.1.2 Fase 2

La seconda fase si occupa di elaborare i risultati parziali ottenuti dai blocchi nella prima fase, restituendo il risultato finale. Seguendo un meccanismo analogo a quello descritto per la prima fase, ogni blocco di thread (di dimensione `blockDim`) legge dalla *memoria globale* un numero doppio di valori rispetto alla sua dimensione. Ad ogni iterazione ogni thread legge un valore presente nella prima metà degli elementi (corrispondente al proprio `threadId`) e un valore nella seconda metà. In questo modo viene sfruttata la *coalescenza della memoria* in quanto thread aventi id adiacenti accedono ad indirizzi di memoria adiacenti. Inoltre così facendo non si ha *divergenza*, in quanto tutte le thread attive in un

blocco hanno id consecutivi e vengono evitati i conflitti tra gli accessi ai banchi di *memoria condivisa*.

Ogni thread legge quindi due valori, posti a distanza `blockDim`, e ne scrive la somma in una cella di un array allocato nella *memoria condivisa*. In questo caso non è necessario appoggiarsi alla tabellaIndici, in quanto i risultati scritti dalla prima fase sono già ordinati correttamente. Analogamente a quanto descritto nella fase 1, una volta caricate le somme parziali nella *memoria condivisa* il kernel entra in un ciclo il quale dimezza ad ogni iterazione gli elementi da elaborare e le corrispondenti threads, fino ad ottenere n elementi risultanti.

Tuttavia, poiché un singolo kernel potrebbe non essere sufficiente per ottenere il risultato finale, questa fase viene iterata fino a lanciare un kernel contenente un solo blocco di thread ed ottenere quindi un array di lunghezza n. Il lancio di più kernel in sequenza serve anche come punto di sincronizzazione globale in quanto, per poter procedere nel calcolo, occorre avere la garanzia che tutti i blocchi dell'iterazione precedente abbiano terminato l'esecuzione. Il numero di iterazioni nei casi in esame si mantiene estremamente contenuto ed è in buona sostanza logaritmico nella dimensione dell'input, non comportando dunque un overhead eccessivo.

2.1.3 Loop unrolling

In entrambe le fasi abbiamo scelto di implementare il loop unrolling relativo al ciclo che esegue la riduzione all'interno del kernel. In particolare quanto è attiva un'unica warp, le istruzioni vengono eseguite in modalità SIMD sincrona, procedendo in lockstep. Non è quindi necessario invocare una sincronizzazione esplicita tramite la funzione `__syncthreads()`.

Tuttavia il compilatore potrebbe ordinare le istruzioni di accesso alla *memoria condivisa* inducendo un comportamento non corretto. Ad esempio potrebbe utilizzare i registri (aventi come scope una singola thread) come buffer per ottimizzare le istruzioni store. In questo caso la *memoria condivisa* è utilizzata come veicolo di comunicazione tra le threads all'interno di un warp, rendendo quindi necessario una redichiarazione del puntatore come `volatile`. In questo modo il compilatore esegue il flushing dei registri ad ogni scrittura, e preleva i dati direttamente dalla memoria senza utilizzare i registri come cache.

Come funzione alternativa abbiamo valutato l'utilizzo di `__threadfence()`, tuttavia abbiamo verificato che, utilizzando questa funzione, veniva aggiunto un overhead inutile se, come nel nostro caso, vi è un unico warp attivo.

Per poter applicare quest'ottimizzazione efficientemente necessitiamo di ridurre al minimo il numero di branch decisi a runtime. Abbiamo dunque utilizzato i template per parametrizzare la dimensione del blocco di thread. Nel codice host uno switch invoca il kernel opportuno in funzione della dimensione scelta a runtime, mentre nel kernel un analogo switch permette di eseguire le corrispondenti istruzioni.

Si noti che questa ottimizzazione fa risparmiare lavoro in tutte le warp, non solo nell'ultima in quanto senza unrolling tutte le warp avrebbero eseguito ogni iterazione del ciclo for, anche se poi non avrebbero partecipato all'aggiornamento dei risultati.

2.1.4 Istruzioni shuffle

Un'ulteriore ottimizzazione che abbiamo valutato (ma non implementato) riguarda l'utilizzo delle istruzioni shuffle per permettere uno scambio di valori efficiente all'interno di una warp. Queste istruzioni sono molto utilizzate per eseguire la riduzione di un array nel senso "classico". Tuttavia nel nostro caso abbiamo degli scambi di valori all'interno di una warp solo quando n è minore di 32 e solo nelle ultime iterazioni del ciclo interno al kernel. Pertanto abbiamo preferito concentrare i nostri sforzi su altre ottimizzazioni.

2.2 Procedura che parallelizza su n

Il secondo approccio vuole andare a coprire i casi in cui m sia un numero piccolo mentre n sia un numero grande.
Anche in questo caso l'elaborazione si compone di due fasi, gestite da due kernel appositi:

- In una prima fase ogni blocco di thread (di dimensione `blockDim`) legge dalla *memoria globale* un numero di valori doppio rispetto alla sua dimensione. Infatti al suo interno ogni thread legge un valore nella prima metà degli elementi (in funzione del proprio id e del blocco d'appartenenza) e un valore a distanza $n \cdot m/2$, e ne scrive la somma in *memoria globale*. Il tutto avviene utilizzando la tabellaIndici come descritto in precedenza.

- La seconda fase si occupa di elaborare i risultati parziali ottenuti dai blocchi nella prima fase, restituendo il risultato finale. Il procedimento eseguito è analogo alla prima fase, con la differenza che in questo caso non è necessario appoggiarsi alla tabellaIndici, in quanto i risultati scritti dalla prima fase sono già ordinati correttamente.

Poiché un singolo kernel potrebbe non essere sufficiente per ottenere il risultato finale, questa fase viene iterata, lanciando ad ogni iterazione un numero inferiore di blocchi, fino a lanciare un kernel contenente un solo blocco di thread ed ottenere quindi un array di lunghezza n.

Il lancio di più kernel in sequenza serve anche come punto di sincronizzazione globale in quanto, per poter procedere nel calcolo, occorre avere la garanzia che tutti i blocchi dell'iterazione precedente abbiano terminato l'esecuzione. Il numero di iterazioni nei casi in esame si mantiene estremamente contenuto ed è in buona sostanza logaritmico nella dimensione dell'input, non comportando dunque un overhead eccessivo.

In una prima implementazione avevamo realizzato un unico kernel in cui la sincronizzazione tra blocchi era emulata utilizzando le istruzioni atomiche per creare una barriera. Questa soluzione evita l'overhead derivato dalla creazione di nuovi kernel, ma si è dimostrata inadeguata nel caso in cui il numero di blocchi lanciati fosse superiore al numero di blocchi che potevano essere effettivamente schedulati nella GPU. Non potendo garantire la stabilità della soluzione nel caso in cui m fosse eccessivamente grande, abbiamo preferito implementare la soluzione descritta nel paragrafo precedente.

2.2.1 Dimensionamento dei bloccchi

In entrambi gli approcci il numero di threads per blocco viene calcolato prima di ogni lancio di kernel.

- Se $n \cdot m$, che rappresenta il numero totale di elementi da elaborare, è maggiore o uguale a maxThreads·2 (in seguito spiegheremo come viene inizializzato maxThreads) il numero di thread allocate è pari a maxThreads.
- Altrimenti sarà la potenza di 2 successiva a $(n + 1)/2$.

Il valore maxThreads ottimale è stato trovato tramite ripetuti test empirici. Abbiamo implementato una procedura che esegue 100 iterazioni consecutive del kernel, per aumentare la precisione di rilevamento, e ne calcola il tempo medio. La procedura è stata poi eseguita all'interno di 2 cicli annidati. Il primo itera su tutte le dimensioni (potenze di 2) delle tabelle in input presenti nei benchmark di riferimento. Il secondo itera su tutte le configurazioni di maxThreads ammesse. Infine per ogni configurazione di input è stata registrato il valore di maxThreads che garantiva prestazioni migliori. Il valore di maxThreads che quasi sempre portava a prestazioni migliori è 512, pertanto abbiamo scelto di lasciare fisso questo parametro.

2.2.2 Scelta tra i due approcci

La parallelizzazione su m è molto efficiente, ma si rivela inapplicabile quando n è più grande della dimensione del blocco e le sue prestazioni peggiorano man mano che ci si avvicina a questo limite. Al contrario, l'approccio che parallelizza su n è utilizzabile, in linea teorica, per elaborare tutti gli elementi. Tuttavia, svolgendo test analoghi a quelli effettuati per determinare il miglior bilanciamento tra threads e numero di blocchi, abbiamo constatato che le prestazioni peggiorano notevolmente in presenza di m grande. Dopo aver fatto numerose prove abbiamo deciso di applicare il primo approccio solo quando n è minore di 512, e il secondo approccio nei restanti casi.

2.2.3 Utilizzo della memoria costante

Un'ulteriore ottimizzazione che abbiamo valutato (ma non implementato) per velocizzare la marginalizzazione è quella di utilizzare la *memoria costante* per memorizzare la tabellaIndici e la matrice ψ_1. Questa memoria, grazie alla cache, permette di reperire i dati molto più velocemente.

Tuttavia essendo limitata a 64KB non è possibile memorizzarvi all'interno più di 5120 elementi (si noti che ogni elemento occupa 8 Byte per il valore e 4 Byte per l'indice). Considerando che per le tabelle di ridotte dimensioni la GPU non porta ad un vantaggio sostanziale, i casi in cui tale soluzione sarebbe applicabile sono un numero relativamente esiguo. Per questo abbiamo preferito concentrarci su altre ottimizzazioni che, a nostro giudizio, possono portare migliori benefici prestazionali.

3 Scattering

L'input della procedura è rappresentato da:

- ψ_2: la matrice che rappresenta la tabella dei potenziali. È composta da valori `double`, ha dimensione $n \cdot m$ ed è rappresentata come un array di elementi.

- tabellaIndici ψ_2: una matrice di valori `unsigned int`, anch'essa di dimensione $n \cdot m$ e rappresentata come un array di elementi. Analogamente a quanto descritto per la marginalizzazione, contiene in ogni cella l'indice della matrice psi del valore di interesse.

  Un generico elemento in posizione (i, j), dove $i \in [0...m]$ e $j \in [0...n]$, è pertanto identificato da:

  $\psi_2[$ tabellaIndici$[$ $i \cdot n + j$ $]$ $]$
  Anche in questo caso per elaborare i dati più efficientemente le tabelle sono dimensionate secondo potenze di 2, e viene utilizzata la costante `SIZE_MAX` nella tabellaIndici per contraddistinguere i dati in eccesso, che non verranno presi in esame durante i calcoli.

- ϕ^*: è un array di valori `double`, di dimensione n

- ϕ: è un array di valori `double`, di dimensione n

L'output della procedura è ψ_2, la matrice ricevuta in input aggiornata come di seguito:

$$\psi_2[tabellaIndici[i \cdot n + j]] = \psi_2[tabellaIndici[i \cdot n + j]] \cdot \phi^{'}[j] \tag{2}$$

dove $i \in [0...m]$, $j \in [0...n]$ e

$$\phi^{'}[j] = \phi^*[j]/\phi[j] \tag{3}$$

L'operazione può quindi essere divisa in due fasi:

3.0.4 Fase 1

La prima fase esegue la divisione elemento per elemento degli array ϕ e ϕ^*, aggiornando il valore direttamente in ϕ. Sebbene siano presenti molte librerie che eseguono efficientemente simili operazioni per quanto riguarda la somma, non sono state trovate varianti per la divisione. Abbiamo pertanto implementato il kernel, allocando per ogni blocco un numero di threads pari a:

- maxThreads nel caso in cui n (dimensione dell'array) fosse superiore a maxThreads;

- la potenza di 2 successiva ad n altrimenti.

Analogamente a quanto descritto nella procedura per il calcolo della marginalizzazione, il valore maxThreads ottimale è stato trovato tramite ripetuti test empirici. In questo caso il valore di maxThreads che quasi sempre portava a prestazioni migliori è 256, pertanto abbiamo scelto di lasciare fisso questo parametro.

3.0.5 Fase 2

Durante la seconda fase ogni elemento appartenente alla j-esima riga della matrice psi (recuperato attraverso la tabellaIndici) viene moltiplicato per il corrispondente j-esimo elemento dell'array restituito dalla prima fase. La procedura suddivide i blocchi e le threads in modo da avere una threads per ogni elemento della matrice, sfruttando la stessa funzione utilizzata per la prima fase dello scattering. Anche in questo caso il valore di maxThreads che quasi sempre portava a prestazioni migliori è 256.

4 Normalizzazione

L'input della procedura è rappresentato da:
$\psi_2{}^*$: matrice di valori `double` di dimensione $n \cdot m$ ed è rappresentata come un array di elementi.

Output:
$\psi_2{}^*$ normalizzata, ossia in modo che la somma di tutti i suoi elementi sia 1.

La procedura si può scomporre in due fasi:

- nella prima fase viene eseguita la somma di tutti gli elementi della matrice.
- nel caso in cui la somma totale sia diversa da 1, la seconda fase si occupa di moltiplicare ogni elemento della matrice per un valore costante precedentemente calcolato.

Essendo operazioni molto utilizzate esistono diverse librerie che le implementano. Abbiamo quindi effettuato dei test prestazionali utilizzando le librerie Thrust e cuBLAS. Dai test le cuBLAS si sono rivelate più veloci rispetto alle Thrust, probabilmente a causa del fatto che queste ultime non assumono che ogni riga abbia la stessa lunghezza, e di conseguenza non sfruttino appieno questa regolarità.

4.0.6 Soglia di parallelizzazione

Analizzando la figura 2 si nota come un gran numero di tabelle presenti una dimensione troppo piccola per poter ottere uno speedup positivo dall'elaborazione su GPGPU. Abbiamo pertanto tentato di stabilire una soglia al di sotto della quale elaborare i dati solo su CPU.

Il controllo viene effettuato sulla dimensione della tabella dei potenziali, prima di eseguire ogni fase di marginalizzazione e di scattering (si noti che a seconda di come procedono le elaborazioni potrebbe risultare conveniente eseguire la marginalizzazione su GPGPU ma non lo scattering, o viceversa).

Mediante misurazioni empiriche abbiamo identificato come 5000 il numero minimo di elementi per cominciare un'elaborazione basata su GPGPU. Si noti tuttavia che tale valore costituisce solo un'approssimazione in quanto in presenza di piccoli input i tempi misurati risultano essere molto bassi, ed è dunque difficile ottenere misure precise. Abbiamo inoltre verificato che, dato l'elevato overhead, una volta trasferiti i dati su scheda video è conveniente elaborarli fino alla fine, anche quando questi diventano piccoli.

5 Test e conclusioni

I test di performance sono stati effettuati su una macchina con cpu Intel Core i7-3770 e gpu NVidia GeForce GTX 680. Le caratteristiche più approfondite si trovano in Tabella 1.

Come benchmark sono state utilizzate le reti presenti nella pagina web `http://bndg.cs.aau.dk/html/bayesian_networks.html` (allegate al codice nella directory RetiEsempio). Per poterle utilizzare è stato costruito un parser che costruisce la rete bayesiana e tutta la procedura di individuazione cricche e costruzione del junction tree per poter applicare l'algoritmo.

Sono state fatte diverse misurazioni per analizzare l'impatto della nostra implementazione parallela, riassunte nella tabella 2. La tabella è suddivisa in cinque parti:

- Nella prima parte vengono riportate le caratteristiche delle reti in esame.
- La seconda parte riporta i tempi totali di esecuzione dell'algoritmo, considerando quindi l'overhead dovuto ai trasferimenti in memoria e al lancio di nuovi kernel.
- La terza parte riporta i tempi relativi all'esecuzione delle fasi di marginalizzazione e scattering su CPU.
- La quarta parte riporta i tempi relativi all'esecuzione delle fasi di marginalizzazione e scattering su GPGPU, senza però contare l'overhead dovuto ai trasferimenti in memoria.
- La quarta parte riporta i tempi relativi all'esecuzione delle fasi di marginalizzazione e scattering su GPGPU, comprensivi dell'overhead dovuto ai trasferimenti in memoria.

Come ci si aspettava si hanno risultati migliori al crescere delle dimensioni medie del separatore, tuttavia il tempo trascorso per il trasferimento dei dati tra host e device ci permette di raggiungere uno speedup positivo solo con i benchmark più impegnativi. In particolare dai test notiamo come l'esecuzione su GPGPU sulle reti Munin1, Barley, Link e Water porta rispettivamente ad uno speedup di 1.796, 1.351, 1.316 e 1.086.

In seguito abbiamo analizzato le prestazioni delle singole fasi (illustrate dalle Figure 11, 12, 13, 14, 15, 16, 17) per capire quanta parallelizzazione è possibile ottenere senza considerare l'overhead dovuto ai trasferimenti in memoria. Nella fase di marginalizzazione il nostro algoritmo si comporta bene in tutte le situazioni, portando lo speedup da un minimo di 2.018 ad un massimo di 12.6. Lo scattering invece registra prestazioni migliori, ottenendo valori compresi tra 6 e 16.6.

Si noti inoltre come l'esecuzione su GPGPU comprenda comunque l'esecuzione di alcuni calcoli su CPU quando la dimensione della tabella risulta essere inferiore alla soglia di parallelizzazione descritta nella Sezione 4.0.6.

Le cause dei modesti risultati ottenuti sono a nostro avviso varie:

- come riportato in Figura 2 vi sono molti casi in cui la dimensione dei dati è talmente esigua da rendere difficile se non impossibile ottenere uno speedup positivo.

- come riportato in Figura 1 anche in presenza di tabelle di grandi dimensioni, vi sono molti casi in cui è complicato stabilire quale approccio utilizzare per parallelizzare le operazioni, in quanto il rapporto tra le dimensioni è estremamente variabile.

- inoltre tutte le funzioni implementate sono caratterizzate da una ridotta intensità aritmetica (generalmente 1 flop per ogni elemento caricato). Come conseguenza le prestazioni sono fortemente limitate dai trasferimenti in memoria. Una possibile soluzione a questo problema è rappresentato dall'utilizzo degli streams, messi a disposizione dall'architettura kepler, ma questa soluzione avrebbe richiesto di ripensare globalmente l'algoritmo per gestire in maniera profondamente diversa le elaborazioni.

Tuttavia confrontandoci con l'articolo [1] possiamo notare come i valori che non tengono conto dell'overhead dovuto ai trasferimenti in memoria siano in linea con i loro risultati. In particolare nei benchmark Water e Munin4 abbiamo ottenuto risultati migliori; in Mildew e Diabetes abbiamo ottenuto risultati peggiori, mentre con il benchmark Barley i risultati sono complessivamente equivalenti. A tal proposito è opportuno specificare che:

- nell'articolo [1] non sono state prese in considerazione le reti Link e Munin1, che con la nostra implementazione garantiscone un notevole speedup.

- le specifiche tecniche delle macchine, riportate in Tabella 1 sono notevolmente differenti sia per quanto riguarda la CPU che per quanto riguarda la GPU, rendendo molto difficile un confronto.

- se confrontiamo le caratteristiche dei junction tree, esse sono leggermente diverse e ciò e dovuto al metodo (non specificato nell'articolo) utilizzato per la trasformazione della rete bayesiana in junction tree.

- nell'articolo [1] non è chiaro come siano state misurate le prestazoni e in particolare come sia stato considerato l'overhead dovuto ai trasferimenti in memoria.

Riferimenti bibliografici

[1] Lu Zheng, Ole J Mengshoel e Jike Chong. «Belief propagation by message passing in junction trees: Computing each message faster using gpu parallelization». In: *Proc. of the 27th Conference on Uncertainty in Artificial Intelligence (UAI-11)*. 2011.

GPU NVIDIA GeForce GTX 680	
# of Processing Cores	1536
Shared Memory	48K per block
Global Memory	4 GB
Memory Bandwidth	192 GB/sec peak

CPU Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz	
# of Cores	4
Processor Clock	3.4GHz
Cache	8 MB
Memory	16 GB

Tabella 1: Caratteristiche hardware della macchina su cui abbiamo testato la nostra implementazione.

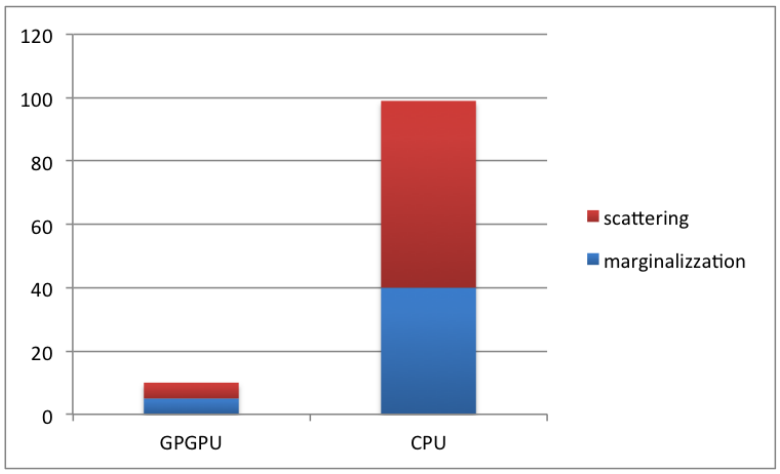

Figura 4: Rete Mildew - tempi misurati in ms.
Confronto tra CPU e GPGPU (senza considerare i trasferimenti in memoria)

Dataset	**Mildew**	**Diabetes**	**Barley**	**Munin4**	**Water**	**Munin1**	**Link**
# JT nodes	29	337	36	877	21	162	586
Max CPT size	1249280	84480	7257600	448000	589824	38400000	2097152
Min CPT size	336	495	216	4	9	4	4
Avg CPT size	117257	29157	476133	10102	144205	516887	40928
Max SPT size	62464	5280	907200	56000	147456	2400000	262144
Min SPT size	72	16	7	2	3	2	3
Avg SPT size	3950	1698	38237	1376	28527	58691	4418
Rilevamenti tempo esecuzione Belief Propagation [ms]:							
BP on CPU	101	275	935	366	152	5976	1122
BP on GPU	132	780	692	688	140	3329	853
Speedup	0,765	0,353	1,351	0,532	1,086	1,796	1,316
Rilevamenti fasi dettagliate [ms]:							
Sequenziale:							
marg. on CPU	40	98	378	157	63	2554	492
scatt. on CPU	59	180	511	236	83	3425	692
CUDA (senza trasferimenti in memoria)							
marg. on GPU	5	45	30	36	23	447	120
scatt. on GPU	5	30	36	28	5	295	48
Speedup marg.	8,000	2,018	12,600	4,361	2,730	5,195	4,100
Speedup scatt.	11,800	6,000	14,194	8,429	16,600	10,860	14,417
CUDA (con trasferimenti in memoria)							
marg. on GPU	39	303	279	214	47	1458	331
scatt. on GPU	83	441	402	394	99	1852	534
Speedup marg.	1,026	0,323	1,355	0,734	1,340	1,593	1,487
Speedup scatt.	0.711	0,408	1,271	0,599	0,838	1,730	1,296

Tabella 2: Statistiche del junction tree (JT) relativi alle reti del dataset, performance della belief propagation (BP) in relazione alle dimensioni delle tabelle dei potenziali delle cricche (CPT) e del separatore (SPT).

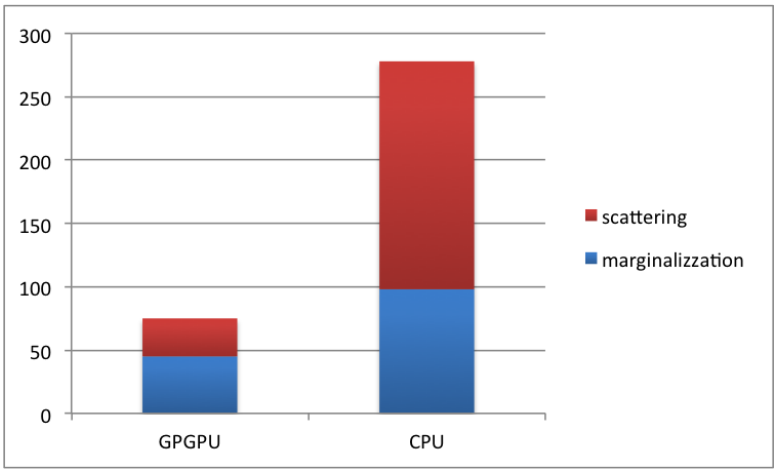

Figura 5: Rete Diabetes - tempi misurati in ms.
Confronto tra CPU e GPGPU (senza considerare i trasferimenti in memoria)

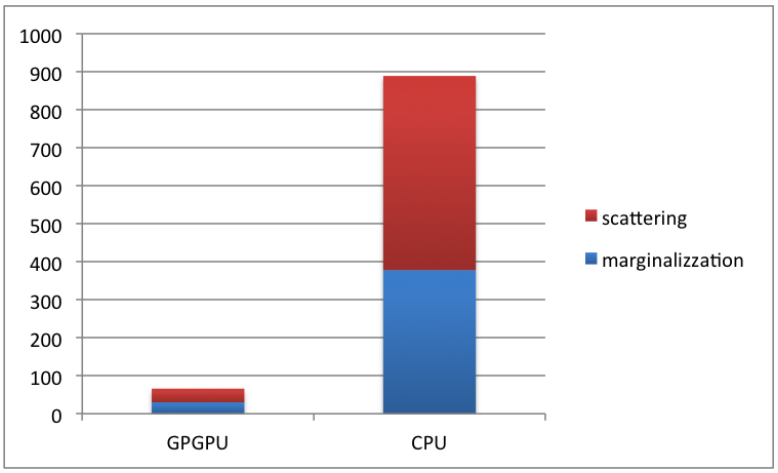

Figura 6: Rete Barley - tempi misurati in ms.
Confronto tra CPU e GPGPU (senza considerare i trasferimenti in memoria)

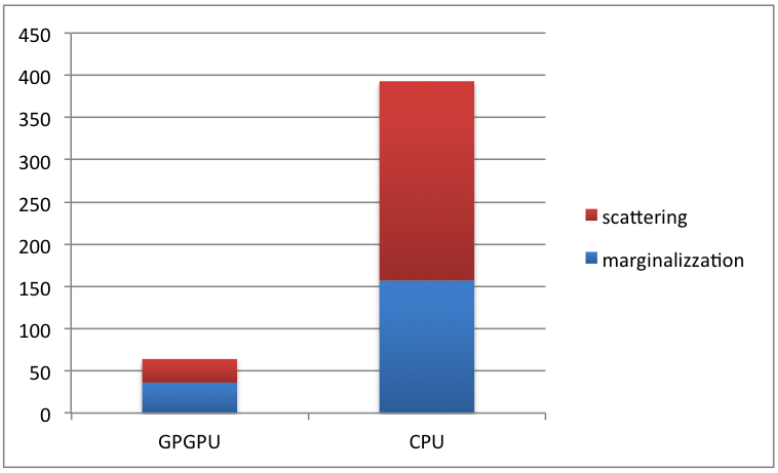

Figura 7: Rete Munin4 - tempi misurati in ms.
Confronto tra CPU e GPGPU (senza considerare i trasferimenti in memoria)

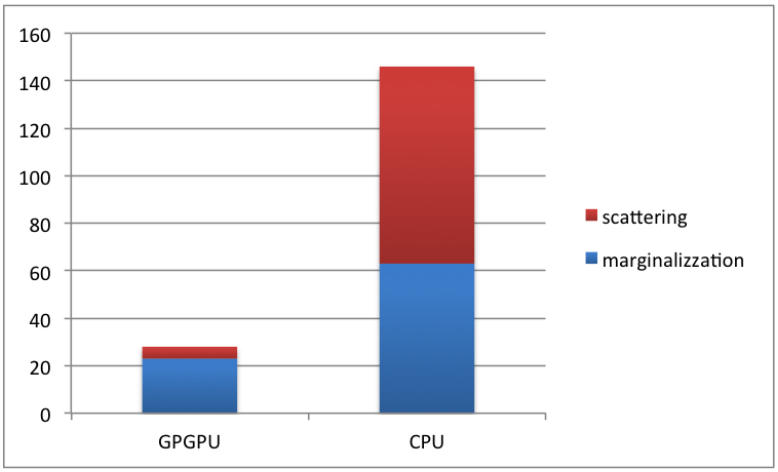

Figura 8: Rete Water - tempi misurati in ms.
Confronto tra CPU e GPGPU (senza considerare i trasferimenti in memoria)

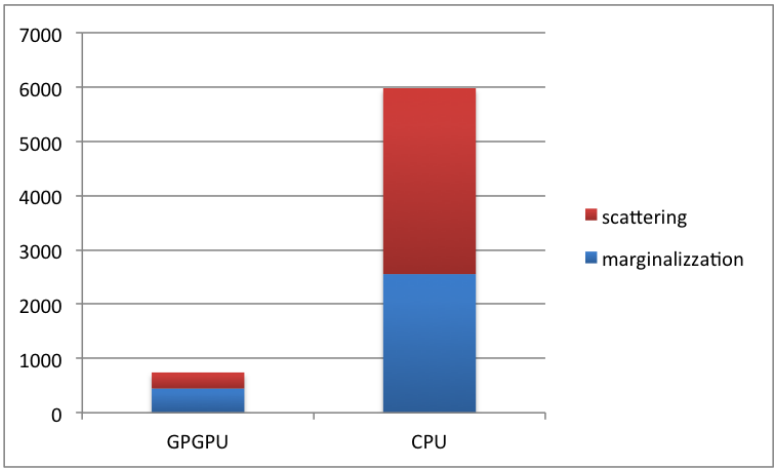

Figura 9: Rete Munin1 - tempi misurati in ms.
Confronto tra CPU e GPGPU (senza considerare i trasferimenti in memoria)

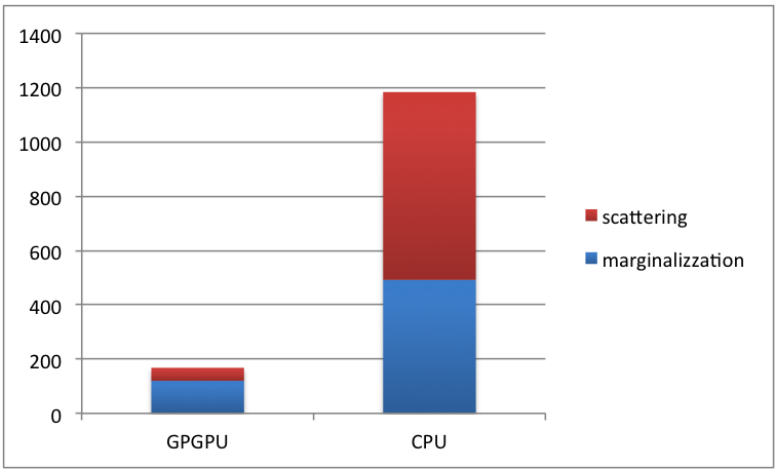

Figura 10: Rete Link - tempi misurati in ms.
Confronto tra CPU e GPGPU (senza considerare i trasferimenti in memoria)

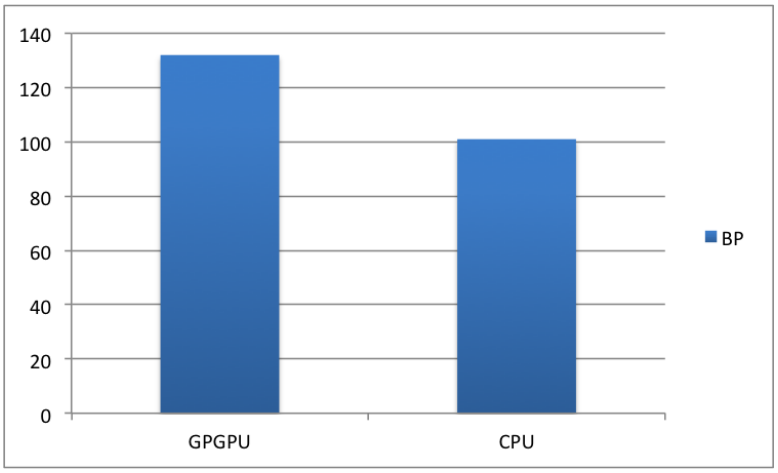

Figura 11: Rete Mildew - tempi misurati in ms.
Confronto tra CPU e GPGPU (tempo totale, inclusi i trasferimenti in memoria)

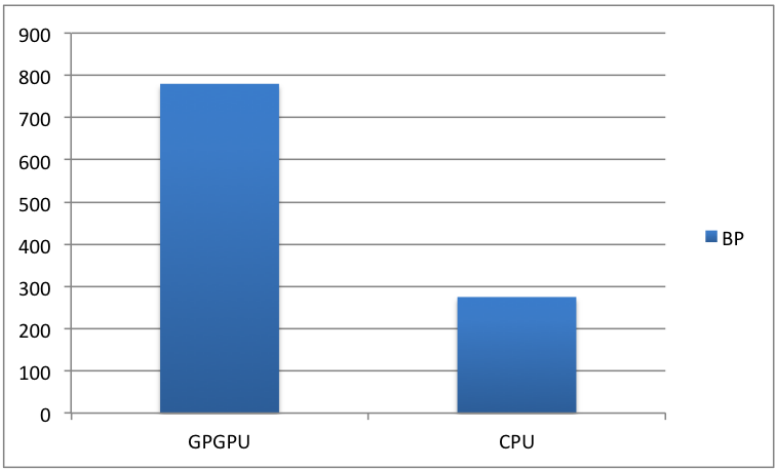

Figura 12: Rete Diabetes - tempi misurati in ms.
Confronto tra CPU e GPGPU (tempo totale, inclusi i trasferimenti in memoria)

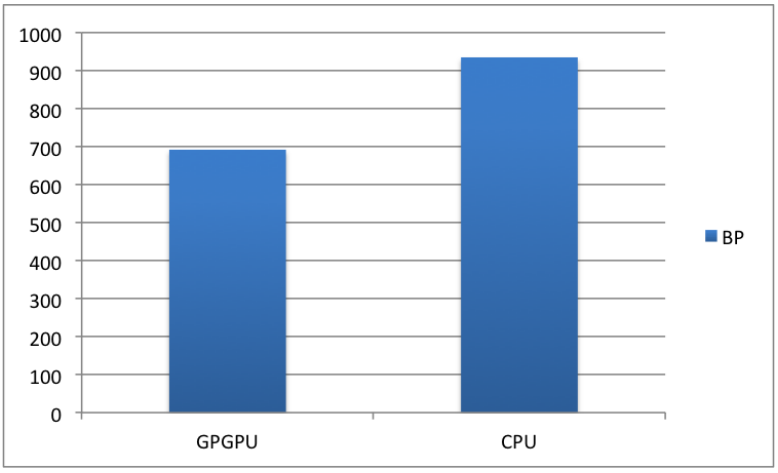

Figura 13: Rete Barley - tempi misurati in ms.
Confronto tra CPU e GPGPU (tempo totale, inclusi i trasferimenti in memoria)

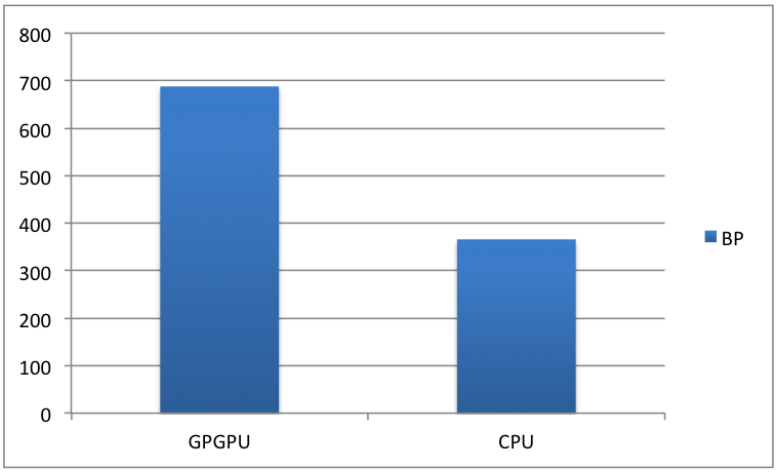

Figura 14: Rete Munin4 - tempi misurati in ms.
Confronto tra CPU e GPGPU (tempo totale, inclusi i trasferimenti in memoria)

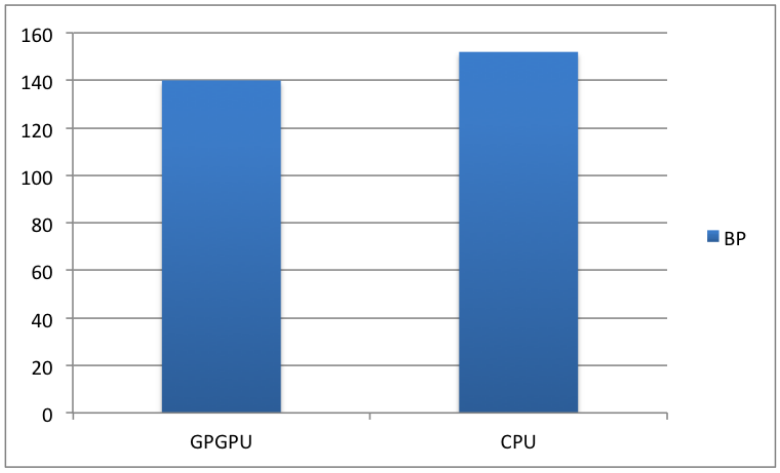

Figura 15: Rete Water - tempi misurati in ms.
Confronto tra CPU e GPGPU (tempo totale, inclusi i trasferimenti in memoria)

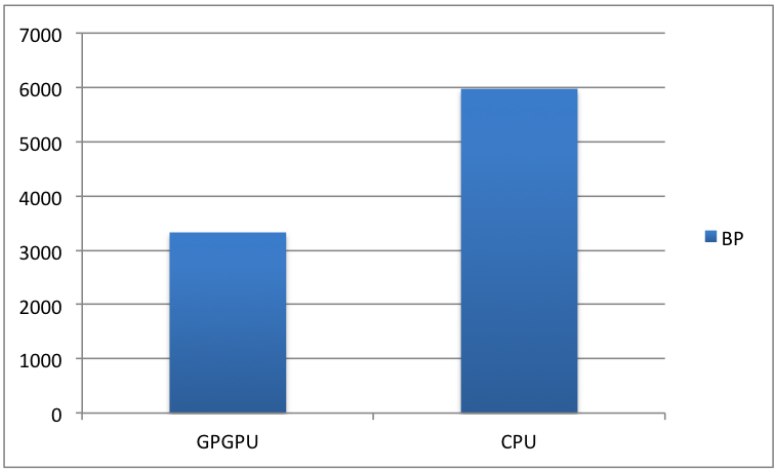

Figura 16: Rete Munin1 - tempi misurati in ms.
Confronto tra CPU e GPGPU (tempo totale, inclusi i trasferimenti in memoria)

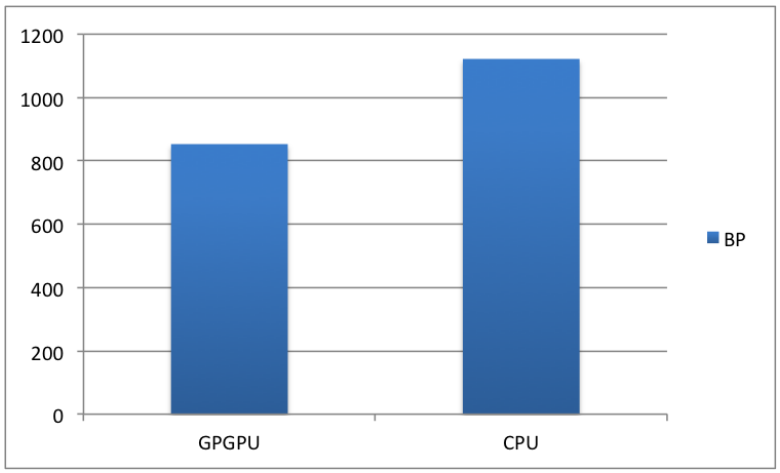

Figura 17: Rete Link - tempi misurati in ms.
Confronto tra CPU e GPGPU (tempo totale, inclusi i trasferimenti in memoria)