Scheduling dei processi su sistemi distribuiti tramite GPU

Gabriele Messina (X81000831)

14/09/2021

1 Idea generale

L'idea è implementare un algoritmo di scheduling che si occupi di gestire un ipotetico sistema distribuito centralizzato, in cui è una GPU a decidere quale task assegnare a quale unità di elaborazione.

La GPU potrebbe essere la soluzione ottimale visto l'ampio carico di lavoro dello scheduler di questi sistemi, immaginando, ad esempio, che si trovi all'interno di un sistema che riceve dall'esterno migliaia di richieste al minuto. Identifichiamo con il termine "task" tutte le richieste provenienti dall'esterno. Lo scheduler deve gestire questi task trovando un'unità di elaborazione che sia in grado di evadere la richiesta, e deve farlo in modo che il risultato sia quanto più vicino possibile all'algoritmo ottimale (ancora da eleggere). I task potrebbero avere delle informazioni riguardanti la priorità, il timestamp relativo alla creazione della richiesta, e un qualche identificativo del servizio a cui stanno cercando di accedere (ad esempio il metodo del web service che vorrebbero interrogare). Ogni work item, cioè ogni scheduler dovrebbe ritornare semplicemente una coppia ¡idTask, idUnitaElaborazione¿ scelta accuratamente in base alla priorità del task, al tempo di attesa già trascorso, al carico di lavoro di ogni unità di elaborazione e al tempo medio di completamento del servizio a cui il task chiede di accedere. (Quest'ultima potrebbe essere calcolata man mano dall'algoritmo o potremmo assumere, visto che si tratta di informazioni che non hanno a che fare con l'esterno, che lo scheduler sia già in possesso delle informazioni necessarie a stimare quanto un servizio sia dispendioso, mediamente, in termini di tempo e di risorse). L'algoritmo potrebbe avere complessità maggiore per provare ad avvicinarsi all'algoritmo ottimale in termini di correttezza delle scelte effettuate, oppure potrebbe prendere in considerazione più fattori ad esempio la possibilità di prelazione o le dipendenze fra task, tuttavia questo renderebbe l'implementazione molto più complessa e, inoltre, inficerebbe sulle prestazioni rendendo possibilmente l'algoritmo inutile. È infatti indispensabile che l'algoritmo sia quanto più veloce possibile in modo da star dietro all'elevato numero di richieste ricevute, anche se questo significa, a volte, fare delle scelte più naïve. Ho trovato questo articolo molto utile che spiega sommariamente quali siano le criticità dello scheduling sui sistemi distribuiti e quali sono gli algoritmi più

usati in ambito enterprise: https://levelup.gitconnected.com/scheduling-tasks-in-distributed-system-61de988c32b5 Sembra però che tutti questi algoritmi facciano uso di DAG (Directed Acyclic Graphs), non so se questo può essere un problema per il codice GPGPU.

2 Note man mano che acquisto conoscenza

jcssp si concentra su sistemi eterogenei che possono avere capacità di calcolo diverse e tassi di trasferimento diversi ecc, nel mio caso il problema è trascurabile e si potrebbe assumere che tutti i processori siano uguali e viaggino sullo stesso canale. I vari workitem avranno a disposizione una DAG con i processi da schedulare e le loro dipendenze ed eventualmente qualche matrice di adiacenza con le informazioni relative ai task. Ogni workitem dovrà quindi prendere in carico una task e scegliere una cpu a cui assegnarla secondo un algoritmo parallelizzabile. jcssp dice di trovare il rank di ogni nodo(task), risalendo dal task ricorsivamente verso la entry, questo causerebbe molti problemi nella nostra implementazione parallela quindi spero di trovare un'altra soluzione (o forse no visto che altrimenti l'implementazione sarebbe imbarazzantemente parallela).

La dag sarebbe però statica nel senso che i dati al suo interno non cambiano e quindi l'unico tipo di ottimizzazione sarebbe legato all'ordine di accesso in lettura, per il resto il lavoro del programma sarebbe quello di riempire le code dei processi relativi ad ogni processore, questo è l'unico problema di concorrenza riscontrabile in questo contesto. Rimangono le operazioni sulla DAG, soprattutto l'attraversamento.

2.1 Step del programma

1. Passo i dati della DAG (vettore con i node e matrice adiacenza) al kernel, questi dati contengono le informazioni relative ai singoli nodi ma nulla circa l'interoperabilità fra le task,
2. A questo punto il kernel calcola le metriche in parallelo relative ai nodi e alle dipendenze.
3. Dopodiché il kernel decide a quale coda e quindi a quale processore assegnare la task.
4. Verifica della correttezza e metriche di throughput lato CPU?

2.2 Note su jcssp

Nel nostro caso:

1. Average Computation Cost (ACC) ha valore, magari randomico.
   (a) "The ACC of a task is the average computation cost on all the m processors and it is computed by using Eqn.(5)" nel mio caso non c'è bisogno di fare una media perché i processori sono tutti uguali?

2. Data Transfer Cost (DTC) è uguale per tutti i task.

3. Rank of Predecessor Task (RPT) "The RPT of a task vi is the highest rank of all its immediate predecessor tasks and it computed using Eqn.(7)"

4. Rank is computed for each task vi based on its ACC, DTC and RPT values. We have used the maximum rank of predecessor tasks of task vi as one of the parameter to calculate the rank of the task vi and the rank computation is given in Eqn. (8). rank(vi) =roundACC(vi)+DTC(vi)+RPT(vi) (8)

5. Priority is assigned to all the tasks at each level l, based on its rank value. At each level, the task with highest rank value receives the highest priority followed by task with next highest rank value and so on. Tie, if any, is broken using ACC value. The task with minimum ACC value receives higher priority.

jcssp gestisce anche le metriche dei processori, ad esempio tiene conto del tempo previsto di completamento del task e di quello effettivo, infatti EFT e AFT (Estimated/actual finish time) sono metriche che vengono usate solo per paragone con gli altri algoritmi, ma non vengono effettivamente usati per lo scheduling.

Implemento l'algoritmo 4 descritto in Breadth First Search In OpenCL Thesis Rick Watertor. A questo si potrebbe aggiungere la gestione di una vera e propria coda invece di far leggere ad ogni WI il nodo con il suo stesso indice. La coda potrebbe essere semplicemente in int2 in cui in prima posizione c'è l'indice del nodo e in seconda il puntatore al next della coda.

3 Idee per prove da fare con gli algoritmi per verificare quale da le prestazioni migliori

3.1 Kernel entry search

invece che fa andare questo kernel per ogni nodo, potrei farlo andare per ogni cella della matrice di adiacenza in questo modo massimizzo le prestazioni. Ogni work item dovrebbe incrementare il contatore per il nodo relativo. In questo modo però ci sarebbero contemporaneamente n_nodes workitem che tentano di incrementare il contatore del nodo, il che potrebbe portare ad un rallentamento notevole!

3.2 MemoryBuffer vari

Invece che distruggerli dopo l'esecuzione del kernel, passarli al kernel successivo.

3.3 Ricerca delle foglie piuttosto che delle root come entries

Il generatore di task che ho trovato genera programmi, intesi come serie di task, con una sola root e una sola leaf, tutti gli altri task convergono in qualche modo a queste due. Ecco perché è indifferente partire dalle foglie o dalle root. Tuttavia credo di poter eliminare un paio di livelli del grafo in modo da poter generare una DAG con leaf multiple.

4 Implementazione

4.1 SDK

Per l'implementazione in parallelo si è deciso di usare OpenCL per la possibilità di eseguire il codice su più piattaforme.

4.2 Struttura DAG

In memoria la DAG è composta da una matrice di adiacenza in cui ogni edge (i,j) indica che i è un padre di j.

Si mantiene inoltre un array di nodes in cui l'indice corrisponde all'id del task e il valore della cella indica il peso del task.

4.3 Prima implementazione

1. si cercano le entries, cioè tutti i task "root" che quindi non hanno nessun parent.
2. a partire dalle entry, si calcolano le metriche di questi task e successivamente si inseriscono le task dipendenti o "figlie" in coda in modo da ripetere ripetere questo punto fino a quando tutte le metriche sono state calcolate.
3. a partire dalle metriche i task vengono ordinati in modo da decidere l'ordine di esecuzione degli stessi.

4.3.1 Difficoltà

Alcuni task hanno più di una dipendenza, per cui è stato necessario mettere in coda i task solo dopo che tutti i parent avessero calcolato la propria metrica. Inoltre la coda andava azzerata ad ogni ciclo.

4.4 Seconda implementazione

Si è deciso di provare a vettorizzare la coda dei task in modo da gestirne più di uno contemporaneamente.

1. si cercano le entries, cioè tutti i task "root" che quindi non hanno nessun parent.

2. a partire dalle entry, si calcolano le metriche di questi task e successivamente si inseriscono le task dipendenti o "figlie" in coda in modo da ripetere ripetere questo punto fino a quando tutte le metriche sono state calcolate.

3. a partire dalle metriche i task vengono ordinati in modo da decidere l'ordine di esecuzione degli stessi.

4.4.1 Difficoltà

In aggiunta alle difficoltà precedenti è stato necessario gestire anche la vettorizzazione.

4.5 Terza implementazione

Si è provato ad usare una matrice di adiacenza trasposta per verificare se questo poteva incidere positivamente sulle prestazioni.

4.6 Terza implementazione

Si è provato ad usare una matrice di adiacenza rettangolare in cui ogni colonna i-esima indica i parent del task i-esimo.