Università degli Studi di Catania
Dipartimento di Matematica e Informatica
Corso di Laurea in Informatica

Implementazione parallela su GPU dell'algoritmo di task scheduling PETS

Gabriele Messina

Relatore
Prof. Giuseppe Bilotta

Anno Accademico 2021/22

Indice dei contenuti

Abstract

Qui viene inserita l'introduzione in inglese.

Introduzione

I sistemi informatici diventano ogni giorno più complessi e le risorse richieste dai processi aumentano di conseguenza, è possibile ottimizzare lo scheduling dei processi in modo da far fronte a questo incremento costante?
In queste pagine dimostreremo che è possibile parallelizzare lo scheduling dei processi per ottenere un miglioramento nei tempi di scelta del processore ottimale, in particolare questo approccio garantirà ottime prestazioni a sistemi distribuiti in cui il carico di lavoro è elevato o su server che forniscono servizi di cloud computing.

1
Contesto

Un'insieme di risorse interconnesse fra loro attraverso una rete ad alta velocità è detto sistema informatico eterogeneo se i nodi della rete hanno caratteristiche e prestazioni diverse fra loro. Un sistema eterogeneo richiede uno scheduling efficiente dei processi per poter garantire alte prestazioni all'utente finale.

1.1 Task scheduling

Il problema del task scheduling consiste nell'assegnare ogni processo di un'applicazione al processore che garantisce le prestazioni migliori in quel momento cioè che, considerando il carico di lavoro su tutti i processori, il costo computazionale del processo e il costo del trasferimento di dati da eventuali processi genitori, porti a compimento il lavoro del processo nel minor tempo. La letteratura a riguardo è vasta e, soprattutto negli ultimi anni, si è concentrata anche sull'ottimizzazione dello scheduling in sistemi eterogenei dove si aggiunge un ulteriore livello di complessità dovuto al fatto che i processori hanno prestazioni diverse fra loro, caratteristica che influisce sul tempo di esecuzione di un processo e, quindi, sulla scelta del processore migliore a cui assegnarlo.

Quando le caratteristiche dei processi si conoscono a priori queste informazioni vengono rappresentate attraverso una struttura dati detta DAG o Direct Acyclic Graph, si tratta di un grafo orientato aciclico che permette di rappresentare le informazioni sulle dipendenze tra processi. Una dipendenza indica che un processo necessita di dati provenienti da un altro il quale deve necessariamente finire l'esecuzione prima che l'altro possa essere eseguito. In una DAG i nodi rappresentano i processi e, gli archi, le dipendenze fra essi. Gli archi sono pesati e il peso indica la quantità di dati da trasferire da un processo all'altro o il tempo necessario al trasferimento.

Il problema del task scheduling è NP-completo, quindi una soluzione ottimale può essere trovata solo dopo una ricerca esaustiva all'interno della struttura dati, ricerca che richiederebbe però tempi di scheduling elevati che i sistemi real-time non possono permettersi. In letteratura sono state quindi proposte molte euristiche che si propongono di trovare un buon compromesso tra tempo di scheduling e approssimazione della soluzione ottima, uno di questi è l'algoritmo PETS.

1.2 L'algoritmo PETS

L'algoritmo Performance Effective Task Scheduling (PETS), presentato in [], è un'euristica nata per diminuire il costo computazione dello scheduling su sistemi eterogenei rispetto ad altre euristiche presenti in letteratura come ad esempio HEFT e CPOP, in particolare PETS evita di calcolare il rango di un processo attraverso una procedura ricorsiva come fanno invece le controparti sopra citate.

L'algoritmo Performance Effective Task Scheduling (PETS), presentato in [], è un'euristica che si propone di ottenere risultati migliori rispetto alle controparti già presenti in letteratura, vedi HEFT e CPOP, in termini di costo computazionale e ottimizzazione generale delle risorse.

1.3 Programmazione parallela

Tuttavia, pur scegliendo un'euristica molto efficiente, la soluzione richiede comunque tempi computazionali elevati, si ci è chiesto quindi se fosse possibile ridurre i tempi di scheduling parallelizzando l'algoritmo e sfruttando le capacità computazionali delle GPU (Graphics Processing Units).

L'esecuzione parallela di un algoritmo consiste nel dividere il problema in problemi di entità minore e risolvere quest'ultimi contemporaneamente su diverse unità di elaborazione. In questo contesto le GPU giocano un ruolo fondamentale grazie alla loro capacità di processare grandi quantità di dati per ciclo di clock e alla gestione nativa di dati vettorizzati.

L'implementazione parallela dell'algoritmo PETS tuttavia non è immediata, cioè non è imbarazzantemente parallela, questo perché i processi hanno dipendenze fra loro, inoltre la ricerca attraverso la DAG e l'ordinamento finale dei processi sulla base della priorità hanno una natura ricorsiva. È stato quindi necessario rivedere l'algoritmo PETS per poter parallelizzare la sua esecuzione, in particolare, sono stati sviluppati dei kernel in linguaggio OpenCL che replicassero il comportamento originario in chiave parallela.

1.4 OpenCL

OpenCL è un framework che permette di sviluppare programmi che possono essere eseguiti in parallelo, in particolare fornisce allo sviluppatore un'interfaccia comune indipendente dalla piattaforma su cui l'applicazione verrà eseguita. Questo permette di avere del codice multi piattaforma, rinunciando però a eventuali ottimizzazioni su piattaforme specifiche, ottimizzazioni per le quali sarebbe necessario sfruttare le interfacce native.

2

Il framework OpenCL

OpenCL è uno standard open per lo sviluppo di programmi multi piattaforma eseguiti in parallelo che implementa un linguaggio simile al C con il quale è possibile scrivere delle funzioni, dette kernel, che potranno essere compilate a run-time ed eseguite in parallelo sull'hardware desiderato (device).

La compilazione a run-time è necessaria per garantire la portabilità del codice su device diversi e avviene sulla piattaforma, detta host, che inizia l'esecuzione del programma attraverso le API (Application Programming Interface) messe a disposizione da OpenCL stesso. In fase di compilazione il codice OpenCL viene quindi tradotto in istruzioni native del device di destinazione attraverso un compilatore, rilasciato dalla casa di produzione dell'hardware, in grado di generare codice performante sul device a partire dal kernel fornito.

Il fatto che OpenCL fornisca API comuni per tutte le piattaforme impedisce di ottimizzare il codice a basso livello, ma è un trade-off necessario per ottenere un'applicazione multi piattaforma.

2.1 Esecuzione di un programma OpenCL

OpenCL fa uso di una coda dei comandi che permette all'host di inviare kernel e relativi dati al device per essere eseguiti.

Dopo essere stati aggiunti alla coda, i kernel vengono assegnati alle varie compute unit del device, unità di elaborazione che eseguono lo stesso codice in parallelo.

La separazione e l'implementazione fisica dei compute unit dipendono dal device scelto e, anche in questo caso, OpenCL si limita a fornire delle API comuni.

In particolare, OpenCL, definisce un global work size e un local work size.
Il global work size definisce il numero di istanze del kernel da eseguire, ogni istanza è detta work-item. Il local work size definisce invece quanti work-item raggruppare in un work-group. Ogni work-item dello stesso work-group verrà eseguito sulla stessa compute unit.

2.2 Suddivisione della memoria in OpenCL

OpenCL ha bisogno di manipolare, oltre alla memoria host che viene gestita nel modo classico attraverso il linguaggio C o C++, anche la memoria device. Per fare questo fornisce un'astrazione con tre classi di memoria:

- Global memory
- Local memory
- Private memory

La global memory è accessibile a tutti i work-item di tutti i work-group ma ha un tempo di risposta più basso delle altre. La local memory è la memoria locale di un work-group ed è quindi accessibile solo a work-item dello stesso work-group. Infine la private memory è limitata al singolo work-item.

Data la natura parallela dell'esecuzione, l'accesso alla memoria deve essere fatto con attenzione per evitare problemi di race condition o di inconsistenza dei dati. OpenCL fornisce quindi delle istruzioni, dette *fence*, per la sincronizzazione dei work-item. Le *fence* impediscono al work-item di proseguire alle istruzioni successive se

prima tutti gli altri work-item del gruppo non sono arrivati alla stessa istruzione *fence*.

La sincronizzazione è, in via teorica, possibile anche tra work-item di diversi work-group, tuttavia OpenCL non garantisce che tutti i work-group siano in esecuzione nello stesso momento, e questo, in combinazione con le istruzioni *fence*, potrebbe portare a dei deadlock.

3

L'algoritmo PETS

L'algoritmo Performance Effective Task Scheduling (PETS), presentato in [], è un'euristica che si propone di ottenere risultati migliori rispetto alle controparti già presenti in letteratura, vedi HEFT e CPOP, in termini di costo computazionale e ottimizzazione generale delle risorse.
L'algoritmo è diviso in tre fasi:

- Ordinamento sulla base del livello
- Calcolo delle priorità
- Selezione del processore

3.1 Ordinamento sulla base del livello

In questa fase si attraversa la DAG attraverso una ricerca in ampiezza (BFS), partendo quindi dal task con tempo di inizio minore (o entry-task), con l'obiettivo di raggruppare nello stesso livello task che fra loro sono indipendenti e che quindi possono essere eseguiti in parallelo. In particolare, per ogni task, il livello ad esso associato è dato dal livello massimo dei suoi predecessori (nodi della DAG da cui il task dipende) incrementato di uno. I task che non hanno predecessori sono detti entry-task o entrypoint e sono assegnati al livello 0.

3.2 Calcolo della priorità

Per ogni task si procede quindi a calcolare la sua priorità o il suo grado in modo da determinare all'interno di ogni livello l'ordine ottimale di esecuzione dei task, l'ordine cioè che fornisce uno scheduling finale migliore. Per fare questo si definiscono tre proprietà:

- Average Computation Cost (ACC): costo medio di computazione del task sui processori.
- Data Transfer Cost (DTC): quantità di dati da trasferire dal task a tutti i suoi successori.
- Rank of Predecessor Task (RPT): il grado più alto fra i predecessori del task.

Il grado è calcolato attraverso la seguente equazione:

$$rank(v_i) = round(ACC(v_i) + DTC(v_i) + RPT(v_i)) \tag{3.1}$$

Quindi, per ogni livello, i task con grado più alto ricevono priorità maggiore.

3.3 Selezione del processore

A questo punto si scandiscono i task sulla base della loro priorità e, ogni task, viene associato al processore migliore, cioè quello che fornisce EFT minore. Dove EFT è

definita come segue:

$$avail[j] := \text{prossimo istante in cui il processore } j \text{ è libero}$$

$$C_{\mathrm{t,i}} := \text{tempo necessario al trasferimento dei dati tra i task } t \text{ e } i$$

$$W_{\mathrm{i,j}} := \text{costo computazionale del processo } i \text{ sul processore } j$$

$$EST(v_{\mathrm{i}}, p_{\mathrm{j}}) = max(avail[j], max(AFT(v_{\mathrm{t}}) + C_{\mathrm{t,i}} : V_{\mathrm{t}} \in pred(v_{\mathrm{i}}))) \tag{3.2}$$

$$EFT(v_{\mathrm{i}}, p_{\mathrm{j}}) = W_{\mathrm{i,j}} + EST(v_{\mathrm{i}}, p_{\mathrm{j}}) \tag{3.3}$$

3.4 Complessità computazionale

La prima fase dell'algoritmo ha complessità $O(e)$ se si implementa la DAG attraverso liste di adiacenza.
L'ordinamento dei task ha complessità $O(\log(v))$ se si implementa una coda di priorità attraverso un heap binario.
Infine per ogni task in coda si calcola il costo su ogni processore, quindi la complessità computazionale dell'algoritmo PETS è $O(e)(p + \log v)$. $O(e)(p * \log v)$?
Con v numero di task, e numero di archi e p numero di processori.

4

L'implementazione in OpenCL

4.1 Generatore dei task

Al fine di poter comparare i risultati ottenuti nel paper[jcssp] con i nostri, si è deciso di implementare un generatore di task con caratteristiche simili a quelle del paper, che fornisca in output una varietà di DAG su cui poter eseguire i test.
Il generatore dipende da diversi parametri di input che sono:

- Number of tasks in the graph (v)
- Out degree (β)
- Shape parameter of a graph (α)
- Communication to Computation Ratio (CCR)
- Range percentage of computation cost (η)

In particolare, l'altezza del grafo è generata randomicamente a partire da una distribuzione uniforme con valore medio pari a $\frac{\sqrt{v}}{\alpha}$, mentre l'ampiezza a partire da una distribuzione uniforme con valore medio pari a $\sqrt{v} \times \alpha$. In questo modo si può generare un grafo più o meno denso semplicemente modificando il parametro α.

Inoltre il parametro CCR indica quanto i task sono impattanti dal punto di vista computazionale rispetto alla mole di dati che analizzano, se CCR è molto basso il costo computazionale è più alto rispetto alla quantità di dati trasmessi ai task successori pertanto l'applicazione può essere considerata ad alta intensità di calcolo, viceversa se CCR è alto l'applicazione trasmette molti dati tra i task ma non è molto pesante dal punto di vista computazionale.

Infine il parametro η indica il grado di eterogeneità del sistema, cioè se ci sono differenze significative tra le prestazioni dei processori, se η è alto i costi dei task sui processori variano molto di processore in processore, viceversa se η basso tutti i processori completeranno lo stesso task in tempi uguali. A partire da questo, il costo medio della computazione W_{i} per ogni task è stato scelto casualmente da una distribuzione uniforme con estremi 0 e $2 \times$ Wdag, dove Wdag è una costante del generatore che indica il costo medio di computazione dei task del grafo, e il costo computazionale $W_{\mathrm{i,j}}$ di ogni task v_{i} su ogni processore p_{j} è scelto randomicamente dall'intervallo $[W_{\mathrm{i}} \times (1 - \frac{\eta}{2}), W_{\mathrm{i}} \times (1 + \frac{\eta}{2})]$.

4.2 Fasi dell'algoritmo

Al fine di implementare attraverso OpenCL una versione parallelizzata dell'algoritmo PETS, si è suddiviso l'algoritmo in 4 fasi:

1. Ricerca degli entrypoint
2. Calcolo dei livelli e dei ranghi
3. Ordinamento dei task
4. Selezione del processore

Per ogni fase è stato implementato un kernel OpenCL da eseguire in parallelo su GPU.

4.2.1 Ricerca degli entrypoint

```
kernel void entry_discover_rectangular(const int n_nodes, global edge_t* restrict edges,
    volatile global int* n_entries, global int* entries)
{
  int current_node_index = get_global_id(0);
  if (current_node_index >= n_nodes) return;

  if (edges[matrixToArrayIndex] <= -1)
  entries[i] = 1;
}
```

Listing 4.1: Find entrypoints kernel II

4.2.2 Calcolo dei livelli e dei ranghi

```
kernel void compute_metrics_rectangular(global int* restrict nodes, global int* queue_,
    global int* next_queue_, const int n_nodes, global edge_t* restrict edges, global
    edge_t* restrict edges_reverse, volatile global int2* metriche, const int
    max_adj_dept)
{
  int current_node_index = get_global_id(0);
  if(current_node_index >= n_nodes) return;

   [...] //omissis of various security checks

  for (int j = 0; j < max_adj_dept; j++) {
    int parentAdjIndex = j;
    matrixToArrayIndex = matrix_to_array_indexes(parentAdjIndex, current_node_index,
        n_nodes);
    int edge_weight = 1;
    int parent_index = edges[matrixToArrayIndex];
    if (parent_index >= 0){
      int weight_with_this_parent = edge_weight + metriche[parent_index].x +
          nodes[current_node_index];
      int level_with_this_parent = metriche[parent_index].y + 1;
      metrics_with_this_parent = (int2)(weight_with_this_parent,
          level_with_this_parent);
      if (gt(metrics_with_this_parent, metriche[current_node_index]))
      metriche[current_node_index] = metrics_with_this_parent;
    }
    int child_index = edges_reverse[matrixToArrayIndex];
    if (child_index >= 0)
    atomic_inc(&next_queue_[child_index]);
  }
}
```

Listing 4.2: Compute metrics kernel II

4.2.3 Ordinamento dei task

```
__kernel void merge_sort(const __global int2* inArray, __global int2* outArray, const uint
    stride, const uint size)
{
    const uint baseIndex = get_global_id(0) * stride;
    if ((baseIndex + stride) > size) return;
    const char dir = 1;
    uint middle = baseIndex + (stride >> 1);
    uint left = baseIndex;
    uint right = middle;
    bool selectLeft;

    for (uint i = baseIndex; i < (baseIndex + stride); i++) {
        selectLeft = (left < middle && (right == (baseIndex + stride) || lte(inArray[left],
            inArray[right]))) == dir;

        outArray[i] = (selectLeft) ? inArray[left] : inArray[right];

        left += selectLeft;
        right += 1 - selectLeft;
    }
}
```

Listing 4.3: MergeSort kernel for metrics couple array, source: `https://github.com/Gram21/GPUSorting`

4.2.4 Selezione del processore

```
void ScheduleTasksOnProcessors()
{
    for (int i = 0; i < metrics_len; i++)
    {
        int current_node = ordered_metrics[i].z;
        if (current_node >= n_nodes) continue;
        int predecessor_with_max_aft = -1;
        int max_aft_of_predecessors = -1;
        int processor_for_max_aft_predecessor = -1;
        int weight_for_max_aft_predecessor = 0;

        for (int j = 0; j < DAG->max_parents_for_nodes; j++)
        {
            int currentParent = edges[matrix_to_array_indexes(j, current_node, DAG->len)];
            if (currentParent > -1) {
                int edge_weight_with_parent = predecessors[matrix_to_array_indexes(j,
                    current_node, DAG->len)];
                int parentEFT = task_processor_assignment[currentParent].z +
                    edge_weight_with_parent;
                if (parentEFT > max_aft_of_predecessors) {
```

```
                max_aft_of_predecessors = parentEFT;
                predecessor_with_max_aft = currentParent;
                processor_for_max_aft_predecessor =
                    task_processor_assignment[currentParent].x;
                weight_for_max_aft_predecessor = edge_weight_with_parent;
            }
        }
    }

    int eft_min = INT_MAX;

    int  cost_of_predecessors_in_different_processors  = 0;
    int remaining_transfer_cost = 0;
    for (int j = 0; j < DAG->max_parents_for_nodes; j++)
    {
        int currentParent = edges[matrix_to_array_indexes(j, current_node, DAG->len)];
        if (currentParent > -1 && currentParent != predecessor_with_max_aft) {
            cost_of_predecessors_in_different_processors  = max(
            cost_of_predecessors_in_different_processors ,
            task_processor_assignment[currentParent].z +
                predecessors[matrix_to_array_indexes(j, current_node, DAG->len)]);
        }
    }

    for (int processor = 0; processor < DAG->number_of_processors; processor++) {
        int cost_of_predecessor_in_same_processor = 0;
        int cost_on_processor = costs[matrix_to_array_indexes(current_node, processor,
            DAG->number_of_processors)];
        if (processor_for_max_aft_predecessor == processor) {
            cost_of_predecessor_in_same_processor = weight_for_max_aft_predecessor;
        }
        remaining_transfer_cost = max(max_aft_of_predecessors -
            cost_of_predecessor_in_same_processor,
            cost_of_predecessors_in_different_processors );

        int est = max(processorsNextSlotStart[processor], remaining_transfer_cost);
        int eft = est + cost_on_processor;
        if (eft < eft_min) {
            eft_min = eft;
            task_processor_assignment[current_node] = cl_int3{ processor, est, eft };
        }
    }

    processorsNextSlotStart[task_processor_assignment[current_node].x] =
        task_processor_assignment[current_node].z;
  }
}
```

Listing 4.4: Compute metrics kernel II

4.3 Ottimizzazioni

5

Risultati

Conclusioni

Inserire qui le conclusioni alla tesi.

Riferimenti bibliografici

1. Simone Faro, Thierry Lecroq: The exact online string matching problem: A review of the most recent results. ACM Comput. Surv. 45(2): 13 (2013)

2. Domenico Cantone, Simone Faro, Emanuele Giaquinta: A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. Inf. Comput. 213: 3-12 (2012)