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 questo contesto, 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[1], 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[2].

1.2 L'algoritmo PETS

L'algoritmo Performance Effective Task Scheduling (PETS), presentato in "Low complexity performance effective task scheduling algorithm for heterogeneous computing environments"[2], 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[3] e CPOP[3], in particolare PETS evita di calcolare il rango di un processo attraverso una procedura ricorsiva come fanno invece le controparti sopra citate.

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[2] tuttavia non è immediata, cioè non è imbarazzantemente parallela, questo perché i processi hanno dipendenze fra loro ed inoltre la ricerca attraverso la DAG e l'ordinamento finale dei processi sulla base della priorità necessitano di un'attenzione particolare alla gestione della memoria per evitare race condition e inconsistenza dei dati. 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 eseguibili 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 dell'hardware.

2

Il framework OpenCL

OpenCL è uno standard open per lo sviluppo di programmi multi piattaforma eseguiti in parallelo che implementa un linguaggio derivato dal 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 detto *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. Tutti i work-item dello stesso work-group verranno eseguiti contemporaneamente 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 a cui accedono i kernel. 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 alto 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 "Low complexity performance effective task scheduling algorithm for heterogeneous computing environments"[2], è un'euristica che si propone di ottenere risultati migliori rispetto alle controparti già presenti in letteratura, vedi HEFT[3] e CPOP[3], 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 (Earliest Finish Time)

minore. Dove EFT è definita come segue:

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

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

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

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

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

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[2] 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 viene calcolata per ogni livello 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.

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.

Inoltre 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 computazionale $W_{\mathrm{i,j}}$ di ogni task v_{i} su ogni processore p_{j} è scelto randomicamente dall'intervallo $[W_{\mathrm{i}} \times (1 - \frac{\eta}{2}), W_{\mathrm{i}} \times (1 + \frac{\eta}{2})]$, dove W_{i}, costo medio della computazione del task i, è stato scelto casualmente da una distribuzione uniforme con estremi 0 e $2 \times$ Wdag, con Wdag costante del generatore che indica il costo medio di computazione dei task del grafo.

Infine β indica quanti archi uscenti avrà in media ogni nodo.

4.2 Struttura dati

Per mantenere in memoria le informazioni, riguardanti la rete, necessarie ad eseguire l'algoritmo si è usata una 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 particolare si è implementata una DAG in cui i nodi rappresentano i processi e, gli archi, le dipendenze fra essi. Gli archi sono pesati e il peso indica il tempo necessario al trasferimento dei dati da un task ad un altro.

L'elenco dei task è stato mantenuto in memoria attraverso un array monodimensionale che associa ad ogni task (attraverso un identificativo numerico) il suo costo

medio di esecuzione sui processori.

Per ragioni di ottimizzazione invece che una singola matrice di adiacenza, sono stato implementate tre matrici rettangolari, una che per ogni task tenesse traccia dei suoi predecessori, una speculare che tenesse traccia dei suoi successori e, infine, una che tenesse traccia dei pesi degli archi della prima matrice.

In ogni matrice rettangolare gli indici delle colonne rappresentano i task mentre le righe gli eventuali archi. Il numero di colonne è quindi fisso, mentre il numero di righe dipende dal massimo numero di successori e predecessori riscontrati all'interno della struttura dati. Inoltre, nelle prime due matrici all'interno delle singole celle vengono memorizzati gli identificativi del nodo all'altro estremo dell'arco, per questo è stato necessario mantenere una terza matrice con i pesi delle connessioni.

Questa scelta è dovuta al fatto che non è possibile implementare una lista di adiacenza sul device OpenCL in quanto non è possibile allocare memoria dinamicamente all'interno dei kernel né è possibile passare un'eventuale lista direttamente al device in quanto un puntatore ad un'area di memoria host non ha alcun senso per il device, a meno di non eseguire un'operazione di mapping della memoria host su device che però diminuirebbe drasticamente le prestazioni.

4.3 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 le prime tre fasi è stato implementato un kernel OpenCL da eseguire in parallelo su GPU.

4.3.1 Ricerca degli entrypoint

In questa fase si ricercano i task che non hanno nessun predecessore, questi verranno assegnati al livello 0 e da loro partirà una ricerca in ampiezza che calcolerà le metriche dei singoli task.

La ricerca di questi task è particolarmente efficiente in quanto basta controllare che la prima riga sia nulla, in questo caso infatti il task non avrà predecessori che altrimenti avrebbero riempito la colonna a partire dalla prima riga.

```
kernel void entry_discover(const int n_nodes, global edge_t* restrict edges, volatile
    global int* n_entries, global int* entrypoints)
{
  int current_node_index = get_global_id(0);

  int matrixToArrayIndex = matrix_to_array_indexes(0, current_node_index, n_nodes);

  if (edges[matrixToArrayIndex] <= -1)
  entrypoints[current_node_index] = 1;
}
```

Listing 4.1: Find entrypoints kernel

4.3.2 Calcolo dei livelli e dei ranghi

Per ogni task si analizzano i suoi predecessori per calcolare la metrica del nodo a partire dal parent che fornisce il rank più alto. Il grado è calcolato a partire da 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.

$$rank(v_i) = round(ACC(v_i) + DTC(v_i) + RPT(v_i)) \tag{4.1}$$

Infine si aggiungono tutti i figli alla coda, in modo da essere analizzati al prossimo ciclo del kernel, che continuerà ad essere eseguito fino a quando tutti i nodi non sono stati analizzati.

```
kernel void compute_metrics(global int* restrict nodes, global int* queue_, global int*
    next_queue_, global edge_t* restrict edges, global edge_t* restrict edges_reverse,
    global edge_t* restrict edges_weights, volatile global metrics_tt* metriche, const int
    max_adj_dept, const int max_adj_reverse_dept, const int n_nodes)
{
   int current_node_index = get_global_id(0);

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

   int ACC = nodes[current_node_index];
   for (int parentAdjIndex = 0; parentAdjIndex < max_adj_dept; j++) {
      matrixToArrayIndex = matrix_to_array_indexes(parentAdjIndex, current_node_index,
          n_nodes);
      int DTC = sum_all_in_column(edges_weights, node_index, n_nodes);
      int parent_index = edges[matrixToArrayIndex];
      if (parent_index >= 0){
         int RPT = metriche[parent_index].x;
         int weight_with_this_parent = DTC + RPT + ACC;
         int level_with_this_parent = metriche[parent_index].y + 1;
         metrics_with_this_parent = (int3)(weight_with_this_parent,
             level_with_this_parent, current_node_index);
         if (metrics_with_this_parent > metriche[current_node_index])
            metriche[current_node_index] = metrics_with_this_parent;
      }
   }

   for (int j = 0; j < max_adj_reverse_dept; j++) {
      int matrixToArrayIndex = matrix_to_array_indexes(j, i, n_nodes);
      int child_index = edges_reverse[matrixToArrayIndex];
      if (child_index >= 0)
         atomic_inc(&next_queue_[child_index]);
   }
}
```

Listing 4.2: Compute metrics kernel

4.3.3 Ordinamento dei task

Una volta calcolato il grado di ogni task è possibile ordinare quest'ultimi in modo che la priorità sia data ai task di livello più basso e, all'interno di ogni livello, al processo con grado maggiore. In caso di parità il task con ACC minore riceve una priorità più alta.

Per l'ordinamento è stato implementato un kernel che applicasse l'algoritmo parallelo Bitonic mergesort. L'implementazione è stata adeguata al codice a partire da quella presente in https://github.com/Gram21/GPUSorting

In questo caso il kernel viene lanciato diverse volte dall'host, ogni volta con parametri *stride* e *blocksize* diversi.

```
for (unsigned int blocksize = limit; blocksize <= metrics_len; blocksize <<= 1) {
    for (unsigned int stride = blocksize / 2; stride > 0; stride >>= 1) {
        if (stride >= limit) {
            sort_task_evts_end = run_bitonic_sort_kernel(metrics_array_len, stride,
                blocksize);
        }
    }
}
```

Listing 4.3: Bitonic mergesort host code

```
__kernel void bitonic_mergesort(const global int3* inArray, global int3* outArray, const
    uint size, const uint blocksize, const uint stride)
{
    uint gid = get_global_id(0);
    uint clampedGID = gid & (size / 2 - 1);

    uint index = 2 * clampedGID - (clampedGID & (stride - 1));
    char dir = (clampedGID & (blocksize / 2)) == 0;

    int3 left = data[index];
    int3 right = data[index + stride];

    sort(&left, &right, dir);

    data[index] = left;
    data[index + stride] = right;
}
```

Listing 4.4: Bitonic mergesort kernel for metrics array, source: `https://github.com/Gram21/GPUSorting`

```
inline void sort(int3 *a, int3 *b, int dir) {
    if (gt(*a, *b) == dir) swap(a, b);
}
inline int gt (int3 V1, int3 V2){
    return compare(V1, V2) > 0;
}
int compare (int3 V1, int3 V2){
    if(V1.y == V2.y){
        return (V1.x < V2.x) ? 1 : -1;
    }
```

```
    return (V1.y > V2.y) ? 1 : -1;
}
```

Listing 4.5: Sort utilities

4.3.4 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:

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

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

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

$$EST(v_{\text{i}}, p_{\text{j}}) = max(avail[j], max(AFT(v_{\text{t}}) + C_{\text{t,i}} : V_{\text{t}} \in pred(v_{\text{i}}))) \tag{4.2}$$

$$EFT(v_{\text{i}}, p_{\text{j}}) = W_{\text{i,j}} + EST(v_{\text{i}}, p_{\text{j}}) \tag{4.3}$$

A causa del fatto che il calcolo di queste metriche non dipende solo dai predecessori ma anche del prossimo slot libero nei processori, non è possibile parallelizzare l'esecuzione di questa fase dell'algoritmo, quindi viene di seguito riportato il codice C++ dell'implementazione seriale.

```
void ScheduleTasksOnProcessors(Graph<graphT> DAG, n_nodes)
{
    cl_int3 * task_processor_assignment = new cl_int3[n_nodes];
    int* processorsNextSlotStart = new int[DAG->number_of_processors];
    for (int i = 0; i < metrics_len; i++)
    {
        int current_node = ordered_metrics[i].z;
        int predecessor_with_max_aft, max_aft_of_predecessors,
            processor_for_max_aft_predecessor, weight_for_max_aft_predecessor;

        for (int j = 0; j < DAG->max_parents_for_nodes; j++)
        {
            int currentParent = DAG->edges[matrix_to_array_indexes(j, current_node,
                DAG->len)];
            if (currentParent > -1) {
                int edge_weight_with_parent = DAG->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, remaining_transfer_cost;
        for (int j = 0; j < DAG->max_parents_for_nodes; j++)
        {
            int currentParent = DAG->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 +
                        DAG->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 = DAG->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 4.6: Algoritmo di selezione del processore

4.4 Ottimizzazioni

Nel corso dello sviluppo sono stati tentati diversi approcci d'ottimizzazione, inizialmente la DAG era stata implementata sfruttando una classica matrice di adiacenza, ma i tempi necessari alla scoperta dei successori e predecessori di ogni nodo erano troppo elevati, si è allora tentato di ridurre i tempi sfruttando la matrice trasposta in modo da sfruttare al meglio la località dei dati e quindi gli eventuali cache hit, ma anche questo approccio non ha migliorato le prestazioni come si ci aspettava. Si è allora passati alle matrici rettangolari e, inoltre, si è implementata una versione vettorizzata del kernel *compute_metrics* che analizzasse n-uple di nodi invece che singoli elementi. Queste due modifiche hanno reso i tempi di runtime dell'implementazione parallela confrontabili con quelli presenti nel paper[2] che, tuttavia, analizza solo dataset di piccola entità. Per dataset di molti nodi invece, come vedremo nel prossimo capitolo, si nota un vantaggio considerevole della GPU rispetto alla CPU in termini di prestazioni.

```
kernel bool compute_metrics_vectorized8(const global int* restrict nodes, global int8*
    restrict queue_, const global edge_t* restrict edges, const global edge_t* restrict
    edges_reverse, const global edge_t* restrict edges_weights, global int3* metriche, const
    int max_adj_dept, const int max_adj_reverse_dept, const int n_nodes)
{
  int work_item = get_global_id(0);

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

  #pragma unroll
  for (int i = 0; i < 8; i++) {
    int node_index = work_item * 8 + i;
    int ACC = nodes[node_index];
    int DTC = sum_all_in_column(edges_weights, node_index, n_nodes);

    for (int parent = 0; parent < max_adj_dept; j++) {
      int matrixToArrayIndex = matrix_to_array_indexes(parent, node_index, n_nodes);
      int parentIndex = edges[matrixToArrayIndex];

      if (parentIndex >= 0) {
        int RPT = metriche[parent_index].x;
        int weight_with_this_parent = DTC + RPT + ACC;
        int level_with_this_parent = metriche[parentIndex].y + 1;
        int3 metrics_with_this_parent = (int3)(weight_with_this_parent,
            level_with_this_parent , node_index);
        if (metrics_with_this_parent > metriche[node_index])
          metriche[node_index] = metrics_with_this_parent;
      }
    }

    for (int child = 0; child < max_adj_reverse_dept; j++) {
      int node_index = work_item * 8 + i;
      int matrixToArrayIndex = matrix_to_array_indexes(child, node_index, n_nodes);

      int childIndex = edges_reverse[matrixToArrayIndex];
      if (childIndex >= 0) {
        int globalIndexStart = (int)(floor(j / 8.0));
        global int* next_nodes = &(queue_[globalIndexStart]);
        atomic_dec(&(next_nodes[childIndex - globalIndexStart * 8]));
        something_changed = true;
      }
    }
  }
  return something_changed;
}
```

Listing 4.7: Vectorized compute metrics kernel

4.4.1 Costo computazionale

La complessità di questa implementazione è $O(n)$ per la ricerca degli entrypoint, $O(n \cdot max(max\_adj\_dept, max\_adj\_reverse\_dept))$ per il calcolo delle metriche e $O(n\ log\ n)$ per l'ordinamento, dove n è il numero di nodi mentre max_adj_dept e $max\_adj\_reverse\_dept$ sono rispettivamente il massimo numero di successori che un nodo può avere e il massimo numero di predecessori. In definitiva, l'algoritmo ha complessità $O(n^2)$ nel caso di una rete in cui ogni task di un livello è connesso a tutti i task del livello successivo, e $O(n \cdot log\ n)$ nel caso medio in cui $max(max\_adj\_dept, max\_adj\_reverse\_dept) \ll n$.

5
Risultati

Riportiamo ora i tempi medi di runtime ottenuti eseguendo dei test con la versione Rectangular su un dataset di 4096 elementi eseguito per 15 volte su ognuno dei seguenti dispositivi:

Si riportano di seguito i tempi di runtime ottenuti eseguendo dei test su dataset di diverse dimensioni e caratteristiche generati attraverso il generatore descritto al capitolo 4.

Si sono effettuate 20 esecuzioni dell'algoritmo per ogni dataset, sia su GPU NVIDIA GeForce GTX 1650 che su CPU Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz.

5.0.1 Confronto tra prestazioni attese e prestazioni effettive

Tenendo conto dei dati teorici riguardanti la memory bandwidth delle varie piattaforme possiamo provare a determinare se questo algoritmo riesce effettivamente a sfruttare tutti i device allo stesso modo oppure se qualcuno di questi performa meno di quanto dovrebbe.

In particolare le memorie dei Device su cui sono stati eseguiti i test hanno banda passante teorica pari a:

1. Dedicata: 121.1 GB/s

2. CPU: 15.0 GB/s (Dovuta al bus PCIe 3.0)

da cui ne deriva una ratio pari a circa 16/3 tra Dedicata e CPU.

I valori del test ottenuti con la versione *Rectangular* vettorizzata mostrano invece una ratio di circa 20/9 calcolata facendo una stima dei byte letti e scritti in media dai kernel e dividendo questa cifra per il tempo di runtime.

Ne deduciamo che la GPU non è effettivamente sfruttata a pieno e che probabilmente le cause di questa mancanza sono da ricercare nella frammentazione della coda dei task da analizzare e nei trasferimenti di memoria con l'host necessari a terminare l'iterazione del kernel.

Il primo di questi problemi è dovuto al fatto che la coda, nella sua implementazione attuale, contiene un valore non nullo nelle posizioni dell'array che corrispondono ai nodi da analizzare. Per ovviare a questo problema si potrebbe pensare di usare una coda effettiva invece di un array, tuttavia questo non è possibile perché i workitem dovrebbero sincronizzarsi per decidere a quale indirizzo poter inserire il nuovo nodo da aggiungere alla coda.

Anche il secondo problema come discusso in precedenza non è risolvibile a causa dell'impossibilità di sincronizzare workitem di workgroup diversi.

5.0.2 Andamento delle prestazioni

Si riportano i risultati dei test eseguiti su una GPU NVIDIA GTX 1650 con dataset di varia grandezza. Per ogni dataset sono state eseguite 15 iterazioni con la versione *Rectangular* vettorizzata dei kernel e successivamente sono stati calcolati i tempi medi di runtime:

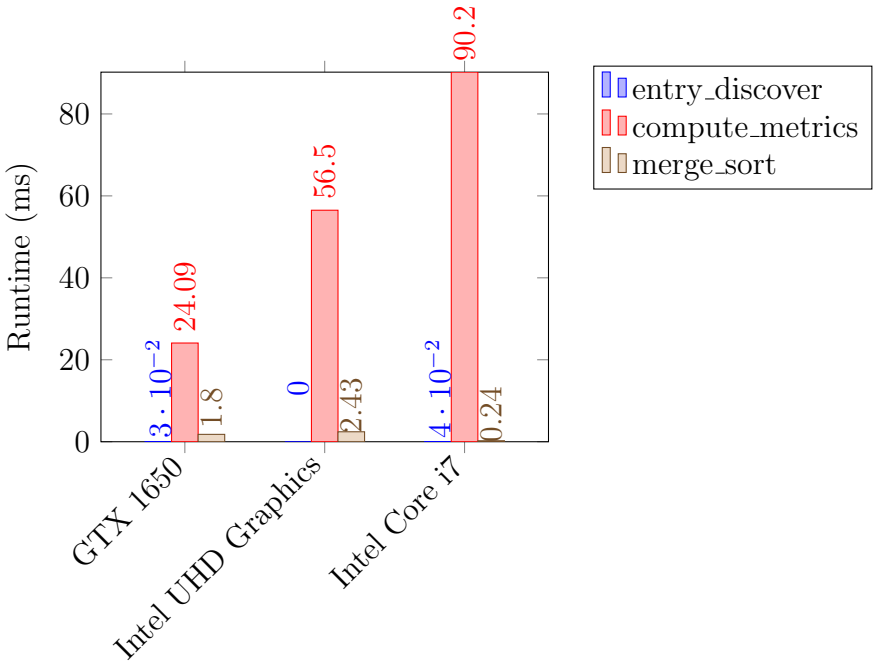

Figura 5.1: Tempi di runtime della versione *Rectangular* vettorizzata

Conclusioni

Inserire qui le conclusioni alla tesi.

Bibliografia

[1] J.D. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384–393. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(75)80008-0. URL: https://www.sciencedirect.com/science/article/pii/S0022000075800080.

[2] E Ilavarasan e Perumal Thambidurai. "Low complexity performance effective task scheduling algorithm for heterogeneous computing environments". In: *Journal of Computer sciences* 3.2 (2007), pp. 94–103.

[3] H. Topcuoglu, S. Hariri e Min-You Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), pp. 260–274. DOI: 10.1109/71.993206.