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

Implementazione parallela su GPU dell'algoritmo di task scheduling PETS

Gabriele Messina

Relatore
Prof. Giuseppe Bilotta

Anno Accademico 2021/22

Abstract

Computer systems are becoming more complex everyday and the resources required by the processes increase accordingly. Is it possible to optimize the scheduling of processes to cope with this constant increase?
In these pages, we will demonstrate that it is possible to parallelize task scheduling, exploiting the computational capabilities of GPUs to improve optimal processor selection times. In particular, this approach will guarantee optimal performance for distributed systems where the workload is high or on servers that provide cloud computing services.

⋆⋆⋆

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, sfruttando le capacità computazionali delle GPU, 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.

Indice dei contenuti

1

Introduzione e 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, le informazioni sulla rete 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. Nel nostro 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 ottimale, 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 computazionale dello scheduling su sistemi eterogenei rispetto ad altre euristiche presenti in letteratura come ad esempio HEFT[3] e CPOP[3], in particolare PETS si propone di non 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 Unit).

L'esecuzione parallela di un algoritmo consiste nel dividere il problema in sottoproblemi 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 ed inoltre la ricerca attraverso il grafo 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[4] è un framework per lo sviluppo di programmi multi piattaforma eseguibili in parallelo. Il tool fornisce allo sviluppatore un'interfaccia comune, indipendente dalla piattaforma su cui l'applicazione verrà eseguita, attraverso la quale è possibile, tra le altre cose, accedere alla memoria GPU e manipolare tipi vettoriali. Sviluppare in OpenCL permette quindi di avere del codice multi piattaforma, al costo però di rinunciare a eventuali ottimizzazioni su piattaforme specifiche, ottimizzazioni per le quali sarebbe necessario sfruttare le interfacce native dell'hardware.

2

Il framework OpenCL

OpenCL[4] 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 (*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.

Questo approccio è comunemente detto *separate source* proprio perché mantiene separato il codice host da quello device, permettendo di gestire il primo attraverso qualsiasi linguaggio purché questo sia supportato da OpenCL o dalla community.

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 secondo il paradigma di programmazione *stream processing*. I dati di input dei kernel vengono quindi gestiti come un flusso di informazioni su cui, per ogni elemento, è necessario eseguire una sequenza di operazioni.

La separazione e l'implementazione fisica dei compute unit dipendono dal device scelto, di conseguenza OpenCL offre delle API comuni che permettono di eseguire le operazioni presenti sulla maggior parte dell'hardware. Questo impedisce di ottimizzare il codice a basso livello ma permette un risparmio di tempo notevole in fase di sviluppo se si vuole eseguire il codice su più device.

In particolare, OpenCL definisce un global work size e un local work size.

Il global work size indica il numero di istanze del kernel da eseguire, ogni istanza è detta work-item.

Il local work size determina 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 sia la memoria host, che viene gestita nel modo classico attraverso il linguaggio C o C++, sia 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.

2.3 Alternative ad OpenCL

Una possibile alternativa a OpenCL è CUDA[5]. CUDA è una piattaforma per lo sviluppo e l'esecuzione parallela di software ideata e mantenuta da NVIDIA. A differenza di OpenCL, CUDA ha un approccio *single source*, i kernel vengono quindi implementati insieme al codice host attraverso delle notazioni aggiunte da NVIDIA ad alcuni dei linguaggi più diffusi. Questo permette a CUDA di gestire più facilmente il trasferimento di dati tra host e device anche per via del fatto che CUDA fornisce delle API di basso livello per l'accesso alle funzionalità native dell'hardware proprietario NVIDIA.

Tuttavia lo sviluppo con CUDA ha degli aspetti negativi, fra tutti l'impossibilità di eseguire il codice su hardware non NVIDIA ma anche il fatto che sia un software proprietario e non uno standard open come OpenCL, caratteristica che potrebbe portare al mancato supporto, ad esempio, di alcuni sistemi operativi.

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:

$$rank(v_{\mathrm{i}}) = round(ACC(v_{\mathrm{i}}) + DTC(v_{\mathrm{i}}) + RPT(v_{\mathrm{i}})) \tag{3.1}$$

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 definito come segue:

$avail[j]$:= prossimo istante in cui il processore j è libero.

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

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

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

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

4

L'implementazione in OpenCL

4.1 Generatore dei task

Al fine di poter comparare i risultati ottenuti in [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.

Per i test, di cui parleremo nel Capitolo 5, sono stati generati 100 dataset ognuno generato scegliendo casualmente ogni parametro tra i range di valori di seguito riportati:

- $v = \{2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}\}$
- $\beta = \{1, 2, 3, 4, 5\}$
- $\alpha = \{0.5, 1.0, 2.0\}$
- $CCR = \{0.1, 0.5, 1.0, 5.0, 10.0\}$
- $\eta = \{0.1, 0.5, 1.0\}$

4.2 Strutture 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 è stata 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 state 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 implementativa è stata fatta perché OpenCL non permette di implementare una lista di adiacenza sul device in quanto non è possibile allocare memoria dinamicamente all'interno dei kernel, né è possibile passare direttamente il puntatore ad un'eventuale lista in quanto il puntatore ad un'area di memoria host non avrebbe alcun senso per il device, a meno di 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, sono state individuate 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, mentre l'ultimo step viene eseguito serialmente.

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 il task non avrà predecessori che altrimenti avrebbero riempito la colonna a partire dalla prima riga.

```
1  kernel void entry_discover(const int n_nodes, global edge_t* restrict edges,
       global int* entrypoints)
2  {
3    int current_node_index = get_global_id(0);
4
5    int matrixToArrayIndex = matrix_to_array_indexes(0, current_node_index,
         n_nodes);
6
7    if (edges[matrixToArrayIndex] <= -1)
8    entrypoints[current_node_index] = 1;
9  }
```

Codice 4.1: entry_discover kernel

4.3.2 Calcolo dei livelli e dei ranghi

Per ogni task si analizzano i suoi predecessori in modo da 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.

Da cui:

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

Dopo aver calcolato il grado del nodo, si aggiungono tutti i suoi figli alla coda, in modo da essere analizzati al prossimo ciclo del kernel se tutti i predecessori sono già stati analizzati. Questo è possibile perché la coda è stata inizializzata, per ogni nodo, con il numero di predecessori. In questo modo ogni predecessore decrementa il valore in coda di ogni suo figlio subito dopo essere stato analizzato. Ad ogni ciclo vengono quindi estratti dalla coda solo i nodi con valore nullo, cioè i nodi per cui il grado di tutti i parent è già stato calcolato. La procedura termina quando tutti i nodi della rete sono stati analizzati.

```
1   void ComputeMetrics::run_kernel(Graph<int>* DAG)
2   {
3     metrics_t* metrics = new metrics_t[metrics_len];
4     for (int i = 0; i < metrics_len; i++) metrics[i] = metrics_t{
          DAG->nodes[i],0,i};
5
6     OCLBufferManager::SetMetrics(metrics);
7
8     bool moreToProcess = false;
9     do{
10      run_compute_metrics_kernel();
11      moreToProcess = reduce(BufferManager.GetQueue());
12    } while (moreToProcess);
13  }
```

Codice 4.2: Codice host per il kernel compute_metrics

```
1  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)
2  {
3    int current_node_index = get_global_id(0);
4
5    [...] //omissis of various security checks
6
7    /* Calcolo del rank a partire dai predecessori */
8    int ACC = nodes[current_node_index];
9    for (int parentAdjIndex = 0; parentAdjIndex < max_adj_dept; j++) {
10     matrixToArrayIndex = matrix_to_array_indexes(parentAdjIndex,
           current_node_index, n_nodes);
11     int DTC = sum_all_in_column(edges_weights, node_index, n_nodes);
12     int parent_index = edges[matrixToArrayIndex];
13     if (parent_index >= 0){
14       int RPT = metriche[parent_index].x;
15       int weight_with_this_parent = DTC + RPT + ACC;
16       int level_with_this_parent = metriche[parent_index].y + 1;
17       metrics_with_this_parent = (int3)(weight_with_this_parent,
             level_with_this_parent, current_node_index);
18       if (metrics_with_this_parent > metriche[current_node_index])
19         metriche[current_node_index] = metrics_with_this_parent;
20     }
21   }
22
23   /* Aggiunta dei successori alla coda */
24   for (int j = 0; j < max_adj_reverse_dept; j++) {
25     int matrixToArrayIndex = matrix_to_array_indexes(j, i, n_nodes);
26     int child_index = edges_reverse[matrixToArrayIndex];
27     if (child_index >= 0)
28       atomic_dec(&next_queue_[child_index]);
29   }
30 }
```

Codice 4.3: 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 [6].

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);
    }
  }
}
```

Codice 4.4: Bitonic mergesort codice host

```
__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;
}
```

Codice 4.5: bitonic_mergesort kernel, 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){s
  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;
}
```

Codice 4.6: Funzioni necessarie al kernel bitonic_mergesort

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 (Earliest Finish Time)

minore. Dove EFT è definita come segue:

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

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

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

Da cui:

$$EST(v_i, p_j) = max(avail[j], max(AFT(v_t) + C_{t,i} : V_t \in pred(v_i))) \tag{4.2}$$

$$EFT(v_i, p_j) = W_{i,j} + EST(v_i, p_j) \tag{4.3}$$

A causa del fatto che il calcolo di queste metriche non dipende solo dai predecessori ma anche dal prossimo slot libero nei processori, si è deciso di non implementare una versione parallela dell'ultima fase dell'algoritmo. Riportiamo quindi il codice C++ dell'implementazione seriale.

```
1   void ScheduleTasksOnProcessors(Graph<graphT> DAG, int n_nodes)
2   {
3     cl_int3* task_processor_assignment = new cl_int3[n_nodes];
4     int* processorsNextSlotStart = new int[DAG->number_of_processors];
5     for (int i = 0; i < metrics_len; i++)
6     {
7       int current_node = ordered_metrics[i].z;
8       int predecessor_with_max_aft, max_aft_of_predecessors,
            processor_for_max_aft_predecessor, weight_for_max_aft_predecessor;
9
10      find_predecessor_with_max_eft(
11        DAG, current_node, predecessor_with_max_aft, max_aft_of_predecessors,
12        processor_for_max_aft_predecessor, weight_for_max_aft_predecessor
13      );
14
15      int cost_of_predecessors_in_different_processors;
16      handle_transfert_cost_beetwen_tasks(
17        DAG, current_node, predecessor_with_max_aft,
18        cost_of_predecessors_in_different_processors
19      );
20
21      assign_task_to_processor(
22        DAG, current_node, max_aft_of_predecessors,
23        processor_for_max_aft_predecessor, weight_for_max_aft_predecessor,
24        cost_of_predecessors_in_different_processors
25      );
26
27      processorsNextSlotStart[task_processor_assignment[current_node].x] =
            task_processor_assignment[current_node].z;
28    }
29  }
```

Codice 4.7: Algoritmo di selezione del processore

```
1  inline void find_predecessor_with_max_eft(Graph<graphT> DAG, int current_node, int
       predecessor_with_max_aft, int cost_of_predecessors_in_different_processors) {
2    for (int j = 0; j < DAG->max_parents_for_nodes; j++)
3    {
4      int currentParent = DAG->edges[matrix_to_array_indexes(j, current_node,
           DAG->len)];
5      if (currentParent > -1) {
6        int edge_weight_with_parent = DAG->predecessors[matrix_to_array_indexes(j,
             current_node, DAG->len)];
7        int parentEFT = task_processor_assignment[currentParent].z +
             edge_weight_with_parent;
8        if (parentEFT > max_aft_of_predecessors) {
9          max_aft_of_predecessors = parentEFT;
10         predecessor_with_max_aft = currentParent;
11         processor_for_max_aft_predecessor =
               task_processor_assignment[currentParent].x;
12         weight_for_max_aft_predecessor = edge_weight_with_parent;
13       }
14     }
15   }
16 }
```

Codice 4.8: Funzione di calcolo del predecessore con EFT massimo

```
1  inline void handle_transfert_cost_beetwen_tasks(Graph<graphT> DAG, int
       current_node, int predecessor_with_max_aft, int max_aft_of_predecessors, int
       processor_for_max_aft_predecessor, int weight_for_max_aft_predecessor) {
2    for (int j = 0; j < DAG->max_parents_for_nodes; j++)
3    {
4      int currentParent = DAG->edges[matrix_to_array_indexes(j, current_node,
           DAG->len)];
5      if (currentParent > -1 && currentParent != predecessor_with_max_aft) {
6        cost_of_predecessors_in_different_processors = max(
7        cost_of_predecessors_in_different_processors,
8        task_processor_assignment[currentParent].z +
             DAG->predecessors[matrix_to_array_indexes(j, current_node, DAG->len)]
9        );
10     }
11   }
12 }
```

Codice 4.9: Funzione di gestione del costo di trasferimento dati tra task

```
1  inline void assign_task_to_processor(Graph<graphT> DAG, int current_node, int
       max_aft_of_predecessors, int processor_for_max_aft_predecessor, int
       weight_for_max_aft_predecessor, int
       cost_of_predecessors_in_different_processors) {
2    int eft_min = INT_MAX;
3    int remaining_transfer_cost;
4    for (int processor = 0; processor < DAG->number_of_processors; processor++) {
5      int cost_of_predecessor_in_same_processor = 0;
6      int cost_on_processor = DAG->costs[matrix_to_array_indexes(current_node,
           processor, DAG->number_of_processors)];
7      if (processor_for_max_aft_predecessor == processor) {
8        cost_of_predecessor_in_same_processor = weight_for_max_aft_predecessor;
9      }
10     remaining_transfer_cost = max(max_aft_of_predecessors -
           cost_of_predecessor_in_same_processor,
           cost_of_predecessors_in_different_processors);
11
12     int est = max(processorsNextSlotStart[processor], remaining_transfer_cost);
13     int eft = est + cost_on_processor;
14     if (eft < eft_min) {
15       eft_min = eft;
16       task_processor_assignment[current_node] = cl_int3{ processor, est, eft };
17     }
18   }
19 }
```

Codice 4.10: Funzione che gestisce lo scheduling del task su un processore

NOTA: Il codice sorgente integrale può essere trovato al seguente indirizzo: `https://github.com/GabrieleMessina/SchedulingWithGPU`

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 quanto 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.

```
1  kernel void compute_metrics_vectorized8(const global int* restrict nodes, global
       int8* restrict queue_, const int n_nodes, const global edge_t* restrict
       edges, const global edge_t* restrict edges_reverse, const global edge_t*
       restrict edges_weights, global metrics_tt* metriche, const int max_adj_dept,
       const int max_adj_reverse_dept)
2  {
3    local int something_changed;
4    int something_changed_for_me;
5    int work_item_local_index = get_local_id(0);
6
7    [...] //omissis of various security checks
8
9    do {
10     if (work_item_local_index == 0)
11     something_changed = 0;
12
13     if (!work_item_outside_range)
14     something_changed_for_me = compute_metrics_vectorized8_internal(nodes,
           queue_, n_nodes, edges, edges_reverse, edges_weights, metriche,
           max_adj_dept, max_adj_reverse_dept);
15
16     if (something_changed_for_me)
17     something_changed = 1;
18
19     barrier(CLK_LOCAL_MEM_FENCE);
20   } while (something_changed > 0);
21 }
```

Codice 4.11: compute_metrics_vectorized8 kernel

```
1 bool compute_metrics_vectorized8_internal(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)
2 {
3   int work_item = get_global_id(0);
4   int something_changed = false;
5
6   [...] //omissis of various security checks
7
8   int8 nodes8 = queue_[work_item];
9   int8 indexes_to_analyze_vec = (nodes8 == 0);
10  global int* _nodes = &(queue_[work_item]);
11
12  if(nodes8.s0 == 0) atomic_dec(&(_nodes[0]));
13  if(nodes8.s1 == 0) atomic_dec(&(_nodes[1]));
14  [...]
15  if(nodes8.s7 == 0) atomic_dec(&(_nodes[7]));
16
17  int4 any_to_analyze_in_quartet = indexes_to_analyze_vec.s0123 +
        indexes_to_analyze_vec.s4567;
18  int2 any_to_analyze_in_couple = any_to_analyze_in_quartet.s01 +
        any_to_analyze_in_quartet.s23;
19  int any_to_analyze = any_to_analyze_in_couple.s0 + any_to_analyze_in_couple.s1;
20  if(any_to_analyze == 0)return something_changed;
21
22  int node_index;
23  if (indexes_to_analyze_vec.s0 == -1){
24    node_index = work_item * 8;
25    something_changed |= compute_metrics(work_item, node_index, nodes, queue_,
          n_nodes, edges, edges_reverse, edges_weights, metriche, max_adj_dept,
          max_adj_reverse_dept);
26  }
27  if (indexes_to_analyze_vec.s1 == -1){
28    node_index = work_item * 8+1;
29    something_changed |= compute_metrics(work_item, node_index, nodes, queue_,
          n_nodes, edges, edges_reverse, edges_weights, metriche, max_adj_dept,
          max_adj_reverse_dept);
30  }
31  [...]
32  if (indexes_to_analyze_vec.s7 == -1){
33    node_index = work_item * 8+7;
34    something_changed |= compute_metrics(work_item, node_index, nodes, queue_,
          n_nodes, edges, edges_reverse, edges_weights, metriche, max_adj_dept,
          max_adj_reverse_dept);
35  }
36
37  return something_changed;
38 }
```

Codice 4.12: compute_metrics_vectorized8 kernel internal function

```
1  bool compute_metrics(int workitem, int node_index, const global int* restrict
       nodes, global int8* restrict queue_, const int n_nodes, const global edge_t*
       restrict edges, const global edge_t* restrict edges_reverse, const global
       edge_t* restrict edges_weights, global metrics_tt* metriche, const int
       max_adj_dept, const int max_adj_reverse_dept){
2    bool something_changed = false;
3    int matrixToArrayIndex;
4    int DTC = 0;
5    for (int j = 0; j < max_adj_reverse_dept; j++) {
6      int matrixToArrayIndex = matrix_to_array_indexes(j, node_index, n_nodes);
7      if(edges_weights[matrixToArrayIndex] > -1){
8        DTC += edges_weights[matrixToArrayIndex];
9      }
10   }
11
12   int ACC = nodes[node_index];
13   metrics_tt metrics_with_this_parent;
14   metrics_tt best_metrics_for_parent = (metrics_tt)(DTC + ACC, 0, ACC);
15
16   /* Calcolo del rank a partire dai predecessori */
17   for (int j = 0; j < max_adj_dept; j++) {
18     int parent = j;
19     matrixToArrayIndex = matrix_to_array_indexes(parent, node_index, n_nodes);
20     int parentIndex = edges[matrixToArrayIndex];
21     metrics_tt parent_metric = metriche[parentIndex];
22     if (parentIndex >= 0) {
23       int weight_with_this_parent = DTC + parent_metric.x + ACC;
24       int level_with_this_parent = parent_metric.y + 1;
25       metrics_with_this_parent = (metrics_tt)(weight_with_this_parent,
             level_with_this_parent, node_index);
26       if (metrics_with_this_parent > best_metrics_for_parent)
27         best_metrics_for_parent = metrics_with_this_parent;
28     }
29   }
30   metriche[node_index] = best_metrics_for_parent;
31
32   /* Aggiunta dei successori alla coda */
33   for (int child = 0; child < max_adj_reverse_dept; j++) {
34     matrixToArrayIndex = matrix_to_array_indexes(child, node_index, n_nodes);
35     int childIndex = edges_reverse[matrixToArrayIndex];
36
37     if (childIndex >= 0) {
38       int globalIndexStart = child / 8;
39       int localIndex = childIndex - globalIndexStart * 8;
40       global int* four_next_nodes = &(queue_[globalIndexStart]);
41       atomic_dec(&(four_next_nodes[childIndex - globalIndexStart * 8]));
42       something_changed = true;
43     }
44   }
45   return something_changed;
46 }
```

Codice 4.13: compute_metrics_vectorized8 kernel internal function

4.5 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 e predecessori che un nodo può avere.

In definitiva, l'algoritmo ha complessità $O(n^2)$ nel caso in cui il grafo sia denso, e $O(n \cdot log\ n)$ nel caso in cui $max(max\_adj\_dept, max\_adj\_reverse\_dept) \ll n$.

5
Risultati

Riportiamo di seguito i tempi di runtime risultanti dall'esecuzione del programma su un pool di 100 dataset ottenuti grazie al generatore di cui al Capitolo 4. Per ogni dataset sono state eseguite 20 iterazioni in modo da evitare errori nell'analisi dovuti a momentanei picchi o crolli di prestazioni dell'hardware.

I kernel sono stati eseguiti sia su GPU che su CPU in modo da poter effettuare dei confronti, nello specifico:

- GPU: NVIDIA GeForce GTX 1650 with Max-Q Design
- CPU: Intel(R) Core(TM) i7-10710U @ 1.10GHz

Device	**GPU**	**CPU**
Frequenza di clock	1245 MHz	1100 MHz
Core	1024	6
Migliore dimensione work-group	32	128
Host	Intel(R) Core(TM) i7-10710U	Intel(R) Core(TM) i7-10710U

Tabella 5.1: Dettaglio sull'hardware usato per i test

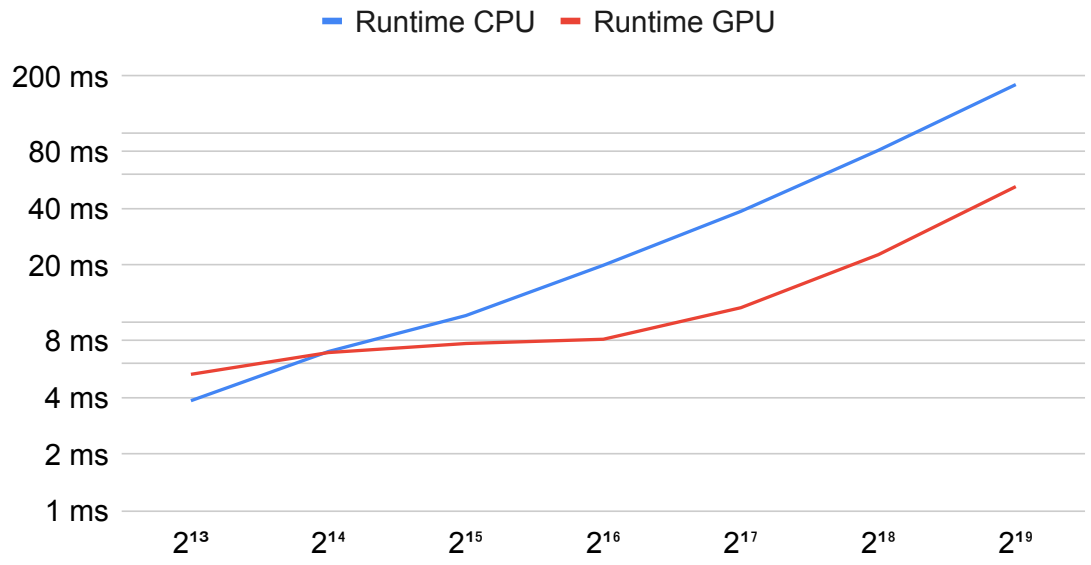

Figura 5.1: Tempi totali di runtime in relazione al numero di task del dataset. Grafico in scala logaritmica.

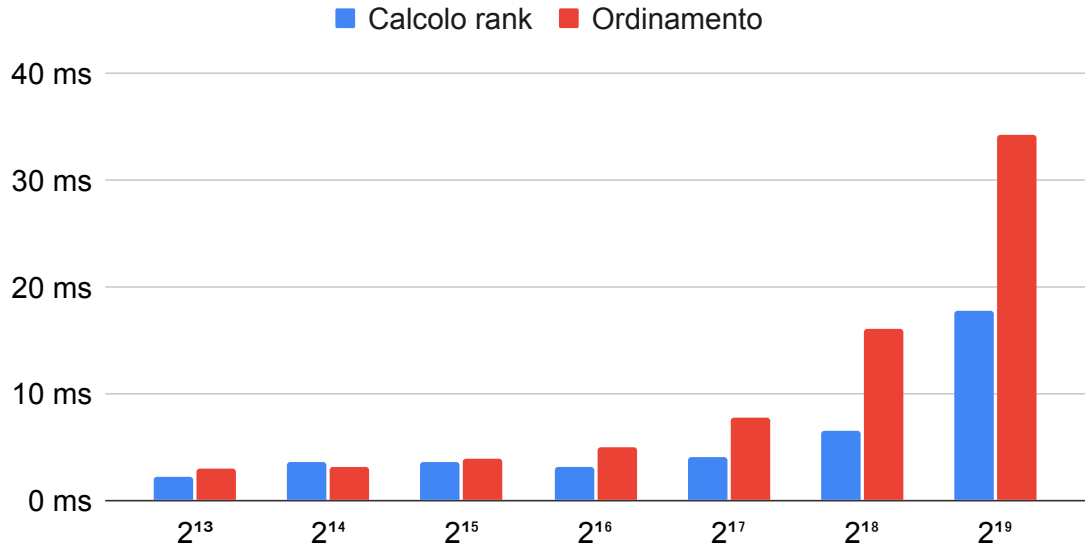

Figura 5.2: Dettaglio tempi di runtime per i kernel *compute_metrics* e *bitonic_mergesort* eseguiti su GPU, in relazione al numero di task del dataset.

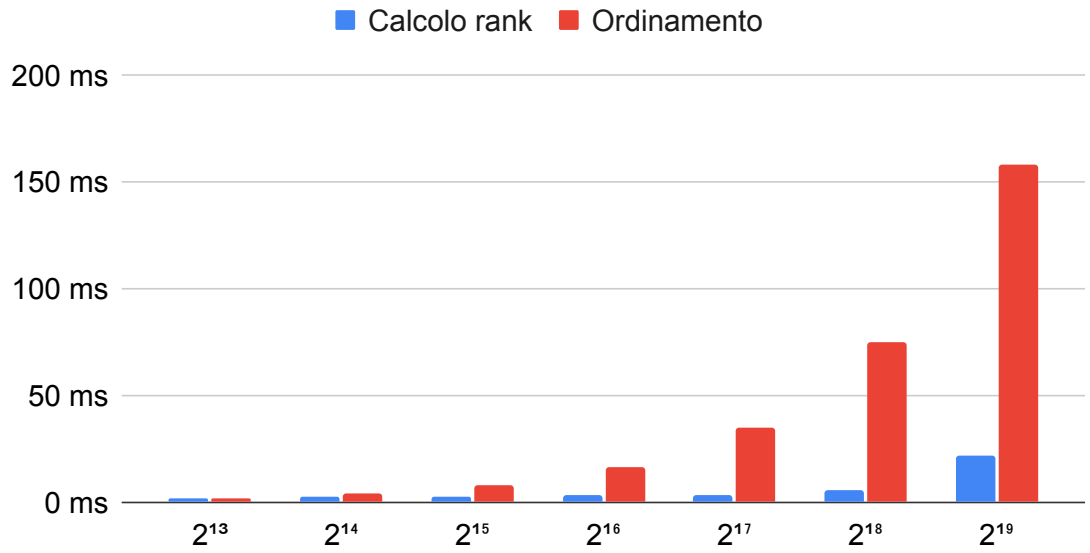

Figura 5.3: Dettaglio tempi di runtime per i kernel *compute_metrics* e *bitonic_mergesort* eseguiti su CPU, in relazione al numero di task del dataset.

Si noti che, in questi grafici, non sono stati riportanti i tempi riguardanti la prima e l'ultima fase dell'algoritmo, questo perché il primo kernel ha tempi d'esecuzione trascurabili rispetto agli altri mentre l'ultimo viene eseguito serialmente dalla stessa macchina host, fornendo quindi gli stessi risultati in termini di tempi d'esecuzione.

5.1 Confronto tra prestazioni attese e prestazioni effettive

Tenendo conto dei dati teorici riguardanti la memory bandwidth delle varie piattaforme possiamo determinare se questo algoritmo riesce effettivamente a sfruttare al meglio tutti i device oppure se uno 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:

- GPU: 121.1 GB/s
- CPU: 15.0 GB/s (Dovuta al bus PCIe 3.0)

da cui ne deriva una ratio pari a circa $8:1$ tra GPU e CPU.

I valori dei test ottenuti rivelano invece una ratio di circa 3.5 : 1 per grandi dataset 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 nella fase 2 e nello scambio di informazioni con l'host necessario a terminare l'iterazione dei kernel di calcolo della metrica e di ordinamento finale.

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 non è risolvibile a causa dell'impossibilità di sincronizzare workitem di workgroup diversi, per cui è necessario ritornare il controllo all'host in modo che possa verificare, attraverso una riduzione parallela, se è ancora necessario proseguire con un'altra esecuzione del kernel.

5.2 Andamento delle prestazioni

Come si evince dalla Figura 5.1, i tempi di runtime su GPU seguono un andamento lineare per piccoli dataset, questo perché il potenziale della GPU non viene sfruttato appieno e, di conseguenza, è più prestante la CPU. Man mano che la grandezza dei dateset aumenta, si nota un'inversione di tendenza che vede i tempi su GPU consolidarsi al di sotto di quelli CPU, sebbene entrambi abbiano un andamento quadratico in linea con l'analisi di complessità riportata nella sezione 4.5.

Per quanto riguarda i singoli kernel, notiamo come l'ordinamento richieda un tempo visibilmente più lungo degli altri kernel sia su GPU (per grandi dataset) che su CPU dove però il runtime cresce esponenzialmente al crescere del numero di task, mentre il kernel per il calcolo del rank risulta essere confrontabile tra i due device

fino ai dataset contenenti 2^{18} task, a partire da questi, infatti, i risultati sono migliori sulla GPU.

Conclusioni

In definitiva, eseguire algoritmi di scheduling sulla GPU sembra una strada percorribile. Le prestazioni che è in grado di fornire una GPU su grandi quantità di dati da processare restano superiori a quelle delle CPU, soprattutto se si ottimizza il codice per le caratteristiche proprie delle schede video come, ad esempio, la vettorizzazione. Inoltre, l'algoritmo implementato in queste pagine è stato notevolmente rallentato dal fatto che OpenCL non consente la sincronizzazione tra workgroup diversi, cosa che nell'eventualità di un'implementazione reale di scheduling su GPU potrebbe essere evitata lavorando ad un livello più basso di astrazione o, addirittura, sviluppando un sistema operativo con kernel in grado di sfruttare la scheda video a questo scopo.

In futuro, si potrebbe parallelizzare anche l'ultima fase del processo, implementando un algoritmo in grado di gestire eventuali conflitti, riscontrati durante l'esecuzione del kernel, dovuti ad associazioni task-processore che si rivelano solo successivamente non ottimali. Inoltre si potrebbe verificare se implementando l'algoritmo con CUDA si ottengono dei miglioramenti nelle prestazioni grazie alle API native.

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.

[4] *The OpenCL Specification*. URL: https://registry.khronos.org/OpenCL/specs/opencl-1.2.pdf.

[5] *CUDA Zone - Library of Resources*. URL: https://developer.nvidia.com/cuda-zone.

[6] *GPU Sorting Algorithms in OpenCL*. URL: https://github.com/Gram21/GPUSorting.

[7] *Codice sorgente integrale*. URL: https://github.com/GabrieleMessina/SchedulingWithGPU.