Dipartimento di Matematica ed Informatica

Relazione progetto GPGPU

Scheduling su sistemi distribuiti sfruttando le capacità di parallelizzazione delle GPU

13/02/2022

Studente:
Gabriele Messina (X81000831)

Università degli Studi di Catania

Indice

1 Idea generale **2**

2 Evoluzione del progetto **2**
2.1 Brute force . . . 3
2.2 Versione Standard . . . 3
2.3 Versione Vec4 . . . 3
2.4 Versione Rectangular . . . 4

3 Prima implementazione funzionante **4**
3.1 Fase 1 - Ricerca degli entrypoint . . . 4
3.2 Fase 2 - Calcolo delle metriche . . . 5
3.3 Fase 3 - Ordinamento dei task . . . 6
3.4 Considerazioni . . . 6
3.5 Variazioni . . . 7

4 Implementazione efficiente **8**
4.1 Fase 1 - Ricerca degli entrypoint . . . 8
4.2 Fase 2 - Calcolo delle metriche . . . 8
4.2.1 Fase 2 - Riduzione . . . 9
4.3 Fase 3 - Ordinamento dei task . . . 9
4.4 Considerazioni . . . 10
4.5 Variazioni . . . 10

5 Analisi delle prestazioni su diversi Device **12**
5.1 Versione *Rectangular* . . . 13
5.2 Versione *Rectangular* vettorizzata . . . 14
5.3 Versione *Rectangular with reduction* vettorizzata . . . 15
5.4 Confronto tra prestazioni attese e prestazioni effettive . . . 15
5.5 Andamento delle prestazioni . . . 16

6 Considerazioni finali **17**

Acronimi **18**

Riferimenti bibliografici **18**

1 Idea generale

Si è cercato di implementare un algoritmo di scheduling che venisse eseguito su GPU in modo da sfruttarne le capacità di parallelizzazione.

Per cominciare, dopo aver letto varia letteratura in merito si è deciso di prendere come articolo di riferimento *Low complexity performance effective task scheduling algorithm for heterogeneous computing environments*[1] che utilizza un Direct Acyclic Graph (DAG) per mantenere le informazioni sulle dipendenze fra i task e la mole di dati comunicati da un task ad un altro.

Tuttavia, visto che il nostro intento non è trovare un algoritmo ottimale, ma verificare la fattibilità dello scheduling tramite GPU, si è deciso di semplificare le specifiche dell'algoritmo *Performance Effective Task Scheduling (PETS)* teorizzato nell'articolo[1].

In particolare trascureremo la quantità di dati trasferiti da un task ad un altro e assumeremo che il canale di trasmissione abbia banda infinita, e che i processori abbiano tutti la stessa potenza di calcolo.

2 Evoluzione del progetto

Avendo quindi deciso d'implementare l'algoritmo PETS[1] si è cercato di suddividerlo in 3 fasi distinte in modo da semplificare lo sviluppo del codice. Le 3 fasi sono:

1. Ricerca degli entrypoint
2. Calcolo delle metriche
3. Ordinamento dei task

Per tutte le fasi sono stati implementati dei kernel OpenCL in modo da poter essere eseguiti su GPU, kernel denominati come segue:

1. *entry_discover*
2. *compute_metrics*
3. *merge_sort*

I dettagli dei vari kernel verranno mostrati più avanti nella relazione, lo scopo di questa sezione è invece quello di dar credito a tentativi di implementazione che, seppur fallimentari, hanno comunque aggiunto tasselli importanti per quello che poi sarebbe stato il risultato finale.

In particolare, mentre l'implementazione del primo e del terzo kernel è stata semplice e senza problemi degni di nota, l'implementazione del secondo kernel ha richiesto più tempo e attenzione, per questo si è deciso di descrivere i dettagli dei vari tentativi di seguito.

2.1 Brute force

Si è cominciato creando un kernel che, a partire dagli entrypoint trovati durante la fase 1, ne calcolasse la metrica e ne aggiungesse in coda i figli. Dopodiché l'Host riprendendo il controllo dell'esecuzione avrebbe controllato la presenza di eventuali cambiamenti nella coda rispetto all'esecuzione precedente, in caso affermativo il kernel sarebbe stato rieseguito con la nuova coda al posto degli entrypoint, altrimenti l'esecuzione sarebbe terminata e si sarebbe passati alla fase successiva.

Tuttavia, così facendo, ogni nodo che avesse avuto più di un genitore, avrebbe calcolato la propria metrica e avrebbe aggiunto i propri figli alla coda più e più volte cosa che di conseguenza avrebbe scatenato molte più esecuzioni del kernel rispetto a quanto necessario.

2.2 Versione Standard

Nel secondo tentativo, questo problema è stato risolto mantenendo in un array temporaneo il numero di genitori di ogni nodo, in questo modo è stato possibile calcolare la metrica di un task solo dopo che quest'ultimo fosse stato messo in coda da tutti i genitori. Questa implementazione è quella che verrà presentata nella prossima sezione con il nome di *standard* e, se pur funzionante, non è soddisfacente in termini di prestazioni forse a causa dei continui trasferimenti di dati tra Host e Device necessari a verificare se terminare o meno l'esecuzione ciclica del kernel. Si è provato quindi ad ottimizzare in vari modi l'algoritmo partendo da questa base funzionante.

Per prima cosa si è provato a prelevare e analizzare più task dalla coda cercando di far svolgere più lavoro ai singoli workitem, tuttavia nessun miglioramento è stato notato, allora si è provato a modificare la coda in modo da farla diventare una coda di `int4` invece della precedente coda di `int`, ma anche in questo caso non si è notato nessun miglioramento delle prestazioni. Si è cercato allora di mitigare il problema del continuo passaggio di dati tra Device e Host implementando una versione vettorizzata che fosse in grado di ciclare autonomamente fin quando tutte le metriche non fossero state calcolate.

2.3 Versione Vec4

Il primo test di un kernel indipendente dall'Host non ha avuto successo in quanto si veniva a creare una mutua attesa tra i workitem che aspettavano i risultati provenienti da workgroup diversi dal proprio e, viceversa, i workitem non ancora partiti aspettavano che i workgroup in esecuzione terminassero il loro lavoro per liberare il pool di workitem che è possibile eseguire contemporaneamente sulla GPU.

Il secondo tentativo, presentato nella prossima sezione con il nome di *vec4*, è stato più fortunato e, anche se non si è riusciti nell'intento di rimuovere completamente la dipendenza dall'Host, si sono almeno evitati i problemi di concorrenza e i relativi deadlock.

Infatti ogni workgroup può adesso sincronizzare i propri workitem terminando l'esecuzione quando non si riscontrano cambiamenti alla coda, tuttavia non ha modo di sincronizzarsi o controllare il lavoro di altri workgroup in quanto non c'è alcuna certezza sul fatto che workgroup diversi vengano eseguiti contemporaneamente, e anzi, soprattutto per grandi quantità di dati e quindi di workgroup si ha la certezza che la contemporaneità non ci potrà mai essere.

Continua quindi, anche se in maniera minore rispetto alle implementazioni passate, ad essere necessario un controllo da parte dell'Host che rimanda in esecuzione il kernel se la coda non è ancora completamente vuota, cioè se ci sono nodi che devono ancora essere processati.

Tuttavia, nonostante gran parte del lavoro venga svolto direttamente sul Device, le prestazioni sono risultate essere leggermente inferiori rispetto alla versione *standard*. Questi risultati hanno demolito l'assunzione iniziale secondo la quale i tempi d'esecuzione molto lunghi del kernel *standard* fossero dovuti ai continui trasferimenti dati tra Host e Device, e ci hanno portato a riflettere su eventuali ulteriori cambiamenti da poter apportare alla nostra implementazione.

2.4 Versione Rectangular

A questo punto si è deciso di cambiare approccio, visto che né la vettorizzazione né la diminuzione dei trasferimenti di dati tra Host e Device hanno migliorato le prestazioni, abbiamo modificato la struttura responsabile del DAG passando da una matrice di adiacenza quadrata ad una rettangolare, in cui la colonna i-esima contiene le informazioni relative ai genitori del task i-esimo.

I dettagli relativi a quest'ultima implementazione verranno descritti nel dettaglio fra poco ma possiamo già anticipare che questa modifica ha comportato un miglioramento notevole delle prestazioni.

3 Prima implementazione funzionante

Per la prima implementazione si è usato un array che contenesse il peso dei singoli task e una matrice di adiacenza per rappresentare gli archi e cioè le dipendenze tra i nodi.

L'algoritmo è diviso in tre fasi, ricerca degli entrypoint, calcolo delle metriche e ordinamento dei task. Per tutte le fasi sono stati implementati dei kernel OpenCL in modo da poter essere eseguiti su GPU.

3.1 Fase 1 - Ricerca degli entrypoint

Si scandisce il grafo in cerca dei nodi che non hanno nessun genitore, e cioè dei task che non hanno nessuna dipendenza e di cui si può calcolare immediatamente la metrica.

```
kernel void entry_discover(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;

    for (int j = 0; j < n_nodes; j++) {
        int matrixToArrayIndex = matrix_to_array_indexes(j, current_node_index, n_nodes);
        if (edges[matrixToArrayIndex] > 0) {
            return;
        }
    }
    entries[current_node_index] = 1;
}
```

Listing 1: Find entrypoints kernel

3.2 Fase 2 - Calcolo delle metriche

Si aggiungono gli entrypoint ad una coda e per ogni elemento della coda si calcola la metrica. La metrica di un task è una coppia in cui il primo elemento è dato dal peso del nodo che indica la mole di lavoro e quindi di tempo che il task impiega prima di terminare, mentre il secondo elemento è il livello, cioè la profondità massima a cui si trova nel grafo (se quindi un task t_1 ha due genitori $\{p_1, p_2\}$, il suo livello sarà dato da $max\_level(p_1, p_2) + 1$). Inoltre per ogni nodo in coda, dopo averne calcolato la metrica, si aggiungono in coda i suoi task dipendenti e la procedura si ripete fino a quando la coda non si svuota.

```
kernel void compute_metrics(global int * restrict nodes, global int *queue_, global int
    *next_queue_, const int n_nodes, global edge_t* restrict edges, volatile global
    int2 *metriche /*<RANK,LIVELLO>*/)
{
    int current_node_index = get_global_id(0);
     [...] //omissis of various security checks
    for(int j = 0; j<n_nodes; j++){
        int parent = j;
        int matrixToArrayIndex = matrix_to_array_indexes(parent, current_node_index,
            n_nodes);
        int edge_weight = edges[matrixToArrayIndex];
        if(edge_weight > 0){
            int weight_with_this_parent = edge_weight + metriche[parent].x +
                nodes[current_node_index];
            int level_with_this_parent = metriche[parent].y+1;
            int2 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 = j;
        matrixToArrayIndex = matrix_to_array_indexes(current_node_index, child, n_nodes);
        int adiacent = edges[matrixToArrayIndex];
        if(adiacent > 0){
            atomic_inc(&next_queue_[child]);
        }
    }
}
```

Listing 2: Compute metrics kernel

Questo kernel tuttavia deve essere eseguito dall'host più di una volta, in particolare verrà invocato fin quando la *next_queue* risultante non sarà identica a quella del ciclo precedente, cosa che indicherà che il kernel non ha fatto cambiamenti nell'ultimo ciclo e ha quindi terminato.

3.3 Fase 3 - Ordinamento dei task

Una volta calcolate le metriche di tutti i task, l'insieme dei task viene ordinato nel seguente modo:

1. Si ordinano in base al livello in modo crescente
2. I task di pari livello vengono ordinati in base al proprio peso in modo decrescente.

```
__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 3: MergeSort kernel for metrics couple array, source: `https://github.com/Gram21/GPUSorting`

3.4 Considerazioni

Dato n numero di nodi, la complessità di questa prima implementazione è $O(n^2)$ per la ricerca degli entrypoint, $O(n^2)$ per il calcolo delle metriche e $O(n\ log\ n)$ per l'ordinamento.

In definitiva, l'algoritmo ha complessità $O(n^2)$ e, come vedremo in dettaglio fra poco, il maggior peso in termini di runtime si riscontra nella Fase 2, per cui ci focalizzeremo maggiormente su quest'ultima fase d'ora in poi.

3.5 Variazioni

Per provare ad abbassare i tempi di runtime troppo alti dei kernel con questa implementazione si sono provati vari approcci:

- Per prima cosa abbiamo provato a lavorare con una matrice di adiacenza trasposta sperando di migliorare i tempi di accesso della GPU alla memoria, con scarso successo.
- In seguito abbiamo provato ad ottimizzare il codice in modo che venisse eseguito il più possibile in GPU perché fino a questo momento l'host prendeva il controllo ad ogni ciclo rimandando in input al kernel la coda che il kernel stesso aveva popolato. Anche in questo caso i risultati sperati non sono sopraggiunti.
- Come ultimo tentativo si è deciso di vettorizzare l'algoritmo nella speranza che facendo fare più lavoro ai singoli work item i tempi di runtime diminuissero, nella realtà però si è ottenuto il risultato opposto.

Riportiamo di seguito i risultati dei test eseguiti con l'algoritmo originale e le sue variazioni su un dataset di 4096 task su GPU NVIDIA GTX 1650 per 15 esecuzioni di cui successivamente è stata calcolata la media dei tempi di runtime.

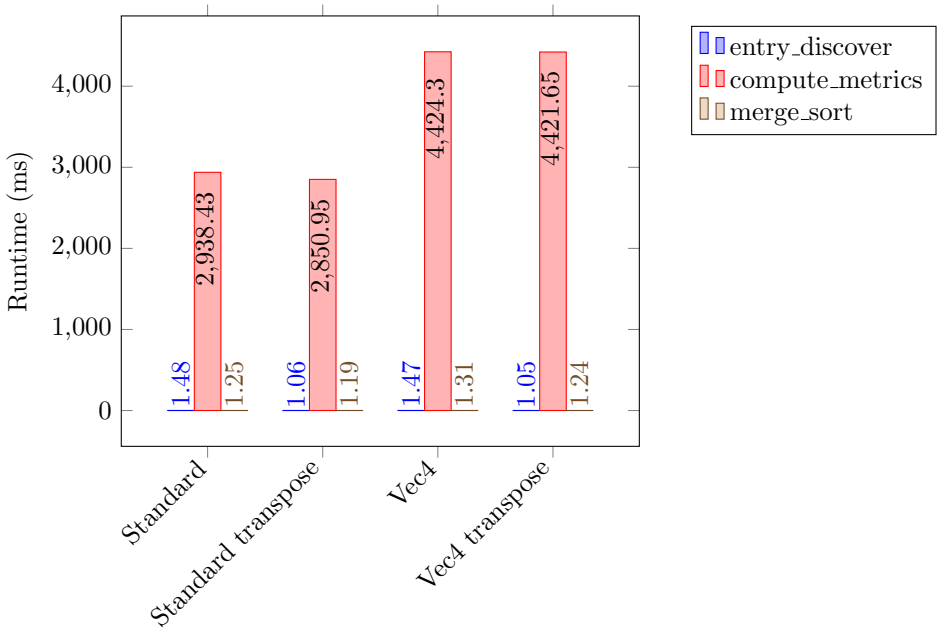


Figura 1: Tempi di runtime della prima implementazione

4 Implementazione efficiente

Per la seconda implementazione si è modificata la matrice di adiacenza in una matrice rettangolare in cui la colonna *i-esima* contiene i riferimenti ai figli del nodo *i-esimo* e, per diminuire ulteriormente i tempi di ricerca all'interno della matrice rettangolare, si è creata anche una matrice speculare che nella colonna *i-esima* contiene i riferimenti ai genitori del nodo *i-esimo*.

Inoltre la condizione che rimanda in esecuzione il kernel se la coda non è vuota viene calcolata direttamente su GPU applicando una riduzione alla coda dei task.

Il resto dell'algoritmo è rimasto identico ad eccezione dei cambiamenti necessari e dovuti alla nuova struttura della matrice di adiacenza.

4.1 Fase 1 - Ricerca degli entrypoint

Grazie alla nuova struttura per verificare che un nodo sia un entrypoint è sufficiente controllare che la prima posizione della colonna relativa al nodo in esame abbia almeno un elemento, cosa che si traduce nel controllare che il primo elemento della colonna esista.

```
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: Find entrypoints kernel II

4.2 Fase 2 - Calcolo delle metriche

Per quanto riguarda il calcolo delle metriche, la differenza rispetto alla prima implementazione risiede solo nel diverso accesso alla matrice di adiacenza.

```
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 5: Compute metrics kernel II

Anche in questo caso il kernel deve essere eseguito dall'host più volte fin quando non si produrrà una *next_queue* vuota ma il controllo avviene tramite riduzione su GPU attraverso il seguente kernel.

4.2.1 Fase 2 - Riduzione

```
kernel void reduce_queue(global int* restrict output, const global int2* restrict input,
    local int* restrict lmem, int npairs)
{
    const int global_index = get_global_id(0);
    int2 pair = global_index < npairs ? input[global_index] : (int2)(0, 0);
    const int local_index = get_local_id(0);
    bool value = (pair.x > 0) || (pair.y > 0);
    lmem[local_index] = value;

    for (int stride = get_local_size(0) / 2; stride > 0; stride /= 2) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if (local_index < stride) {
            value |= lmem[local_index + stride] > 0;
            lmem[local_index] = value;
        }
    }

    if (local_index == 0) output[get_group_id(0)] = value;
}
```

Listing 6: Reduce queue kernel

4.3 Fase 3 - Ordinamento dei task

In questa fase non è stato modificato nulla in quanto il kernel lavora su un array di metriche che non ha alcuna correlazione con la matrice di adiacenza e che quindi non è influenzato dalle modifiche a quest'ultima,

4.4 Considerazioni

La complessità di questa implementazione è $O(n)$ per la ricerca degli entrypoint, $O(n \cdot max\_adj\_dept)$ per il calcolo delle metriche e $O(n\ log\ n)$. In definitiva, l'algoritmo ha complessità $O(n \cdot max\_adj\_dept)$, dove n è il numero di nodi e max_adj_dept è il numero massimo di figli che un nodo può avere.

4.5 Variazioni

Anche in questo caso si è provato a vettorizzare il kernel per abbassare i tempi di runtime che, comunque, rispetto alla prima implementazione sono già più bassi di qualche ordine di grandezza.

Sarebbe stato inutile inoltre verificare eventuali variazioni con la matrice trasposta, in quanto già nell'implementazione base sfruttiamo due matrici di adiacenza, una che tiene traccia dei figli di ogni nodo ed una speculare che tiene traccia dei genitori.

Riportiamo di seguito i tempi di runtime dei kernel con la nuova implementazione e, successivamente, il confronto tra il *compute_metrics* più basso della versione *Standard* e quello della versione corrente compresa la sua variante vettorizzata.

Anche in questo caso per i test si è usato un dataset di 4096 task in esecuzione su una GPU NVIDIA GTX 1650 per 15 esecuzioni di cui successivamente è stata calcolata la media dei tempi di runtime.

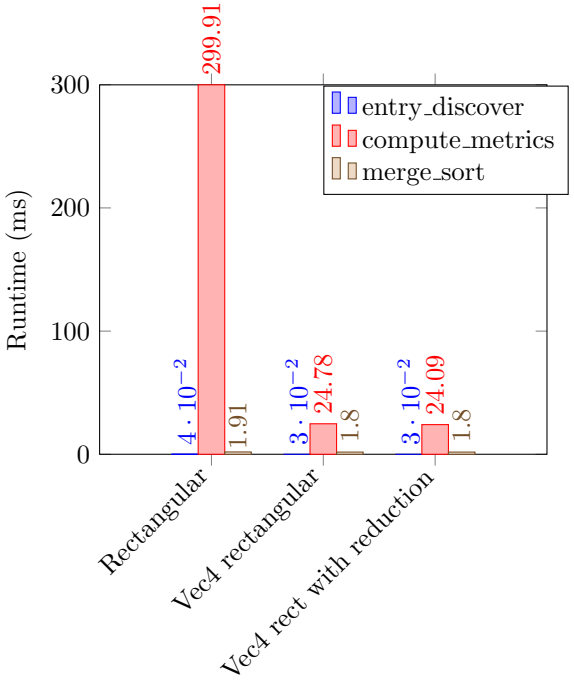


Figura 2: Tempi di runtime della versione *Rectangular*

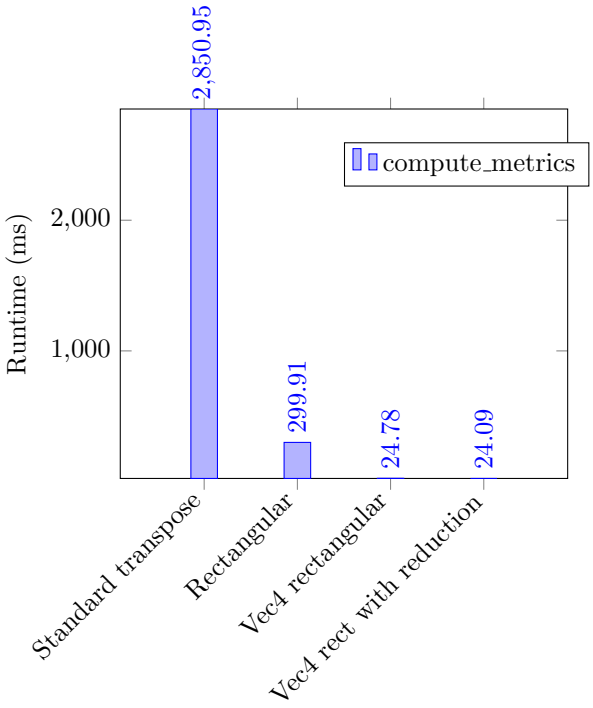


Figura 3: Tempi di runtime della *Standard* e della *Rectangular* a confronto

Come si può vedere quindi, i tempi sono notevolmente migliorati semplicemente cambiando la modalità di memorizzazione e accesso ai dati. Inoltre, visto l'ulteriore miglioramento ottenuto grazie alla versione vettorizzata del kernel si è implementato anche un kernel vettorizzato che sfruttasse una coda di int8 invece del precedente int4, tuttavia in questo caso non si sono notati miglioramenti degni di nota.

Miglioramenti più netti si notano invece mettendo a confronto le versioni *Vec4 rectangular*, *Vec4 rect with reduction* e *Vec8 rect with reduction* sui grandi dataset, mostriamo di seguito i risultati ottenuti usando un dataset di 262144 task in esecuzione su una GPU NVIDIA GTX 1650 per 15 esecuzioni di cui successivamente è stata calcolata la media dei tempi di runtime.

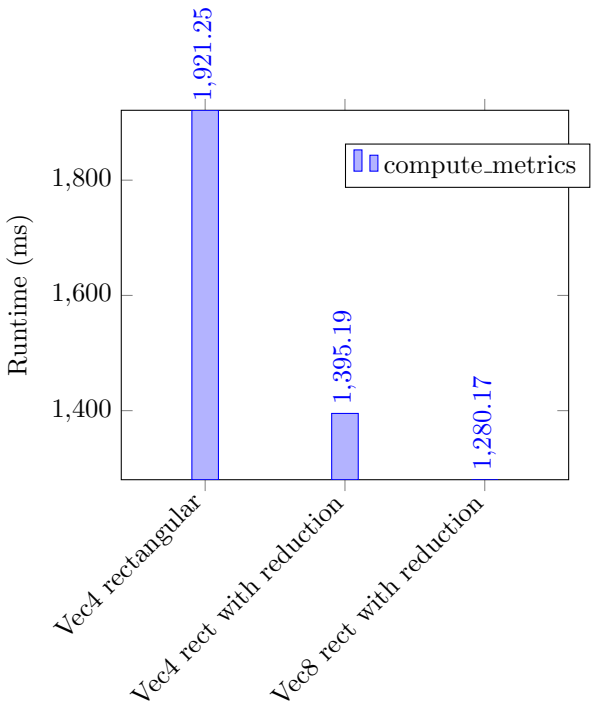


Figura 4: Tempi di runtime della *Vec4 rectangular*, della *Vec4 rect with reduction* e della *Vec7 rect with reduction* a confronto su un dataset di 262144 task

5 Analisi delle prestazioni su diversi Device

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:

1. NVIDIA CUDA GeForce GTX 1650 with Max-Q Design
2. Intel(R) UHD Graphics
3. Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz

5.1 Versione *Rectangular*

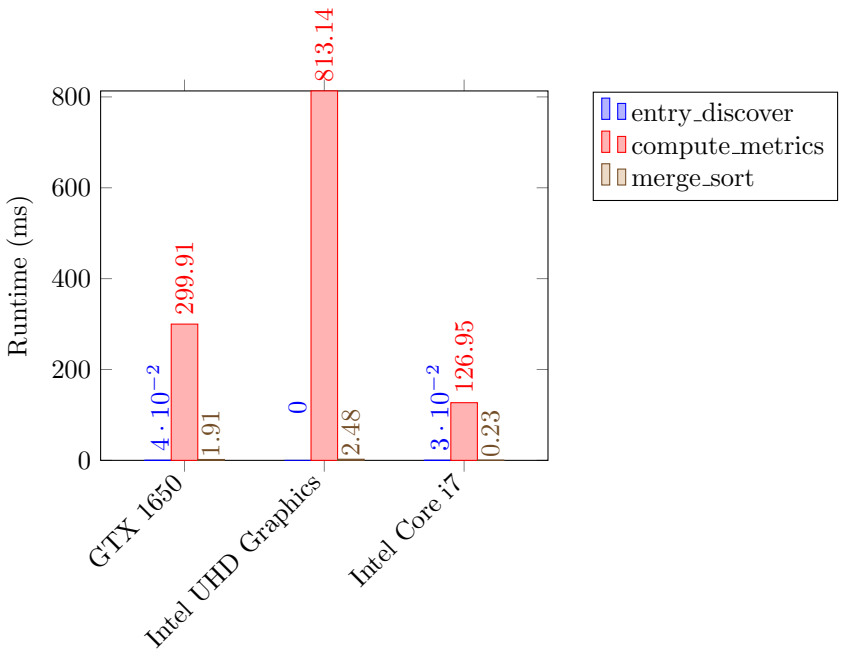


Figura 5: Tempi di runtime della versione *Rectangular*

5.2 Versione *Rectangular* vettorizzata

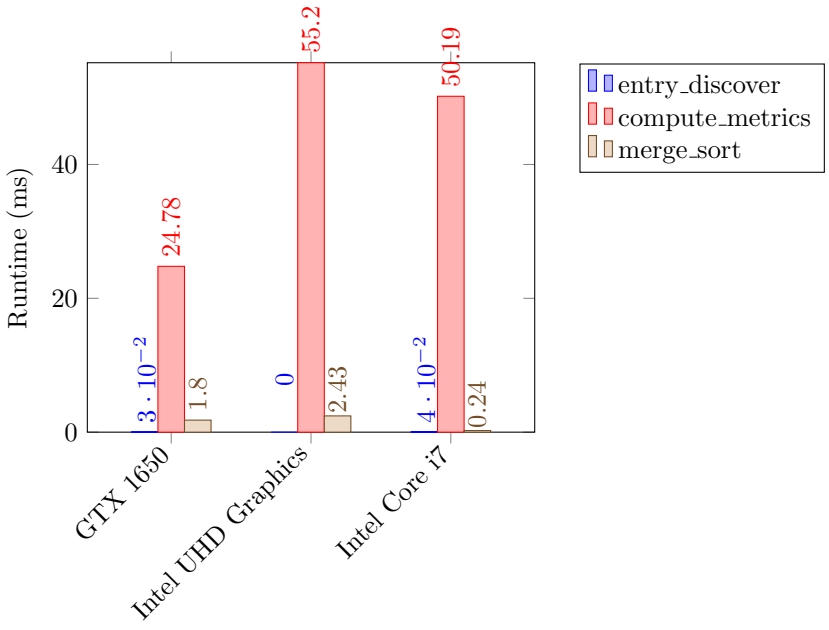


Figura 6: Tempi di runtime della versione *Rectangular* vettorizzata

5.3 Versione *Rectangular with reduction* vettorizzata

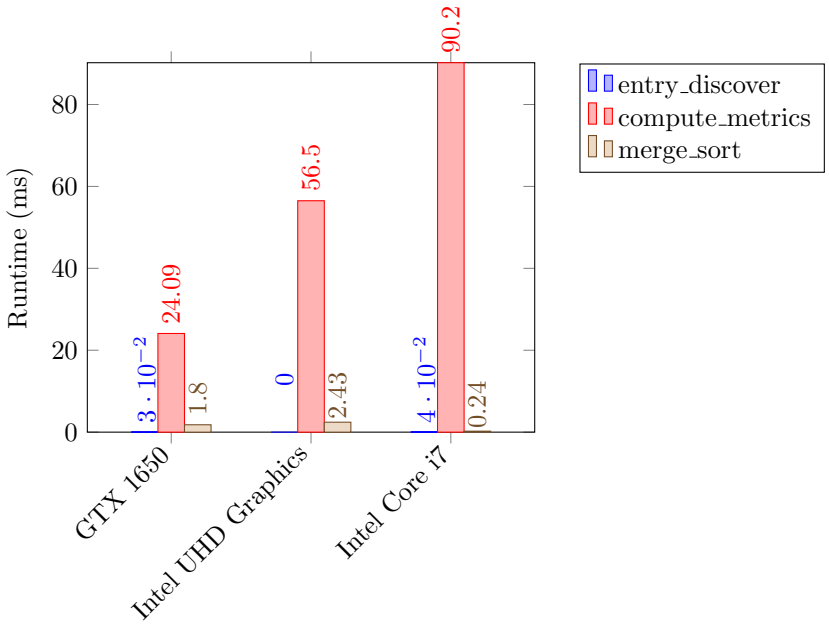


Figura 7: Tempi di runtime della versione *Rectangular* vettorizzata

5.4 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. Integrata: 15.0 GB/s (Dovuta al bus PCIe 3.0)
3. 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 Integrata(o 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. In sostanza la Dedicata performa meglio della Integrata(e della CPU) solo del doppio mentre i valori attesi vorrebbero la scheda Nvidia circa 5 volte più prestante della controparte. Inoltre effettuando una modifica al codice, sfruttando

cioè la funzione clEnqueueMapBuffer fornita da OpenCL invece della precedente clEnqueueWriteBuffer, le prestazioni della Intel UHD incrementano ulteriormente portando la ratio a circa 3/2, cioè l'Integrata, potendo evitare i trasferimenti di dati con l'host visto che lavorano sulla stessa memoria, diventa solo di una volta e mezza più lenta della GPU integrata che ovviamente non può beneficiare di questa modifica. La CPU invece sembra, stranamente, peggiorare i propri tempi di runtime in seguito a questa modifica.

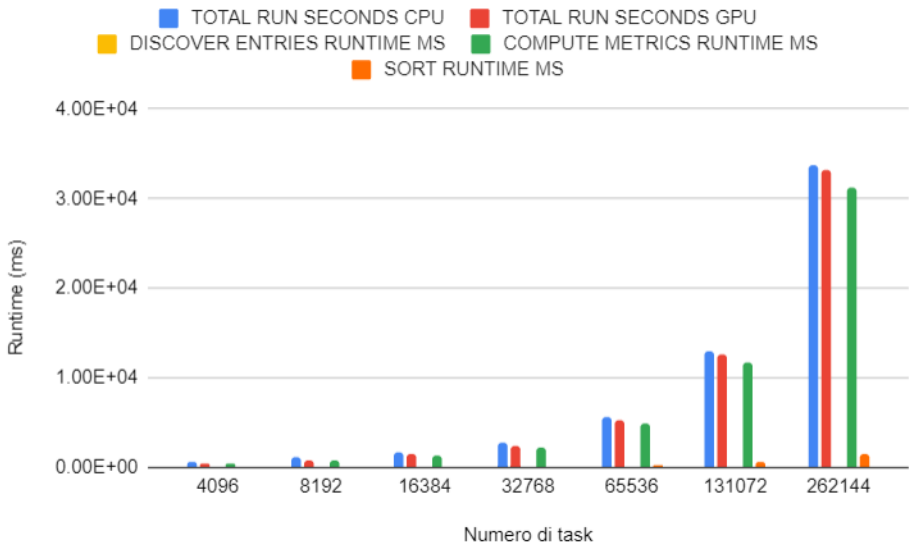
Ne deduciamo che la GPU non è effettivamente sfruttata a pieno e che probabilmente le cause di questa mancanza sono da ricercare nella frammentazione della coda dei task da analizzare e nei trasferimenti di memoria con l'host necessari a terminare l'iterazione del kernel.

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.

5.5 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:



L'andamento delle prestazioni all'aumentare del numero di task è in linea con l'analisi asintotica presentata precedentemente.

6 Considerazioni finali

In definitiva, eseguire algoritmi di scheduling sulla GPU sembra una strada percorribile.

Anche se in questo contesto si è dato per scontato che tutti i task fossero noti all'instante 0, 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. Certo è da considerare anche l'overhead che si avrebbe schedulando dei processi tramite GPU che poi andrebbero eseguiti su CPU, tuttavia nel contesto di grossi applicativi o sistemi distribuiti questa può essere una valida alternativa.

Inoltre l'algoritmo implementato in questo esperimento 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 kernel apposito per interfacciarsi con la scheda video.

Acronimi

DAG Direct Acyclic Graph 2, 4

PETS Performance Effective Task Scheduling 2

Riferimenti bibliografici

[1] 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.