Dipartimento di Matematica ed Informatica

Relazione progetto General Purpose GPU

TRAVELING SALESMAN PROBLEM

23/09/2022

Studente:

Salvo Luca (Matricola 1000006173)

Indice

1 Idea generale

Il **problema del commesso viaggiatore** (chiamato anche **Traveling Salesman Problem** o **TSP**) pone la seguente domanda: "Dato un elenco di città e le distanze tra ogni coppia di città, qual è il percorso più breve possibile che visita ogni città esattamente una volta e ritorna alla città di origine?". Si tratta di un problema NP-hard di ottimizzazione combinatoria.

2 Evoluzione del progetto

2.1 Prima versione

La prima implementazione di questo progetto è separata in 5 fasi:

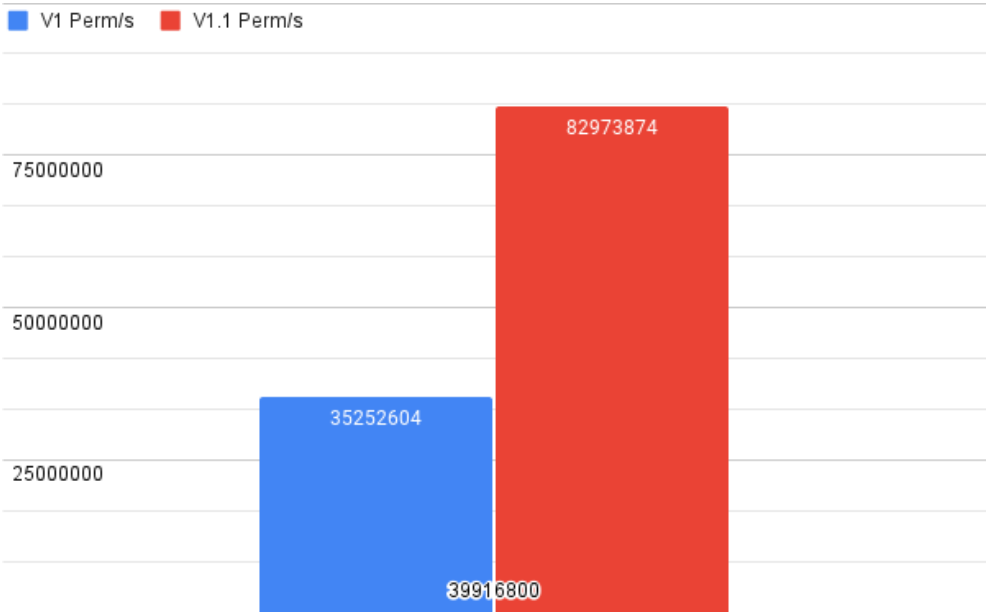
1. Inizializzazione di opencl
2. Creazione di un grafo generato casualmente a partire da una serie di parametri passati da console
3. Caricamento in memoria device della matrice di adiacenza del grafo sotto forma di array contiguo
4. La fase di elaborazione delle permutazioni, un ciclo che si occupa di:
    a. Generare il prossimo chunk di permutazioni
    b. Caricare in memoria device le permutazioni da elaborare
    c. Preparare un buffer in cui scrivere i costi di ogni permutazione
    d. Eseguire il kernel che elaborerà una permutazione per work-item scrivendo il costo della permutazione se identifica un loop o una costante INFINITY in caso contrario
    e. Lettura del buffer dei costi, ricerca del minimo nell'array eseguita su cpu
    f. Salvataggio statistiche (runtime, bandwidth…)
5. Calcolo statistiche della run riportando il risultato su un file csv per una successiva analisi a confronto

Questa versione del codice non segue le best practice e non cerca di utilizzare al meglio la potenzialità del codice parallelizzato ma ha fornito una base su cui poter espandere.

2.1.1 Migliorie lato host

Il metodo di lettura dei costi è stato sostituito con una mappatura dei dati. Avendo generato il buffer con il flag `CL_MEM_USE_HOST_PTR` verrà riutilizzata la stessa memoria pre-allocata e quindi verranno sincronizzati solamente i dati.

Sono state inoltre ridotte le dimensioni dell'array delle permutazioni cambiando il tipo da un `int` ad un `char`, così da risparmiare 3 byte per ogni vertice nell'array, così ottenendo un aumento delle prestazioni in lettura, non si è presentata alcuna limitazione eccessiva delle capacità del sistema dato che non è capace di gestire un numero maggiore di 20 vertici per motivazioni di tempistica.



Media di permutazioni al secondo su un campione di 20 esecuzioni dei due codici sullo stesso task di 12 vertici, per un totale di 39.916.800 permutazioni

Come si può vedere nel grafo sono bastate queste piccole modifiche e l'utilizzo di alcune best practice per ottenere il ~250% delle prestazioni iniziali

La creazione dei buffer in input e la scrittura dei dati in essi è stato modificato, sin dalla prima versione il buffer veniva ricreato dall'array delle permutazioni eseguendo una copia dei dati tramite il flag `CL_MEM_COPY_HOST_PTR`, sono state provate inoltre la possibilità di creare il buffer una sola volta e scrivere o mappare i dati.

Create Buffer with CL_MEM_COPY_HOST_PTR

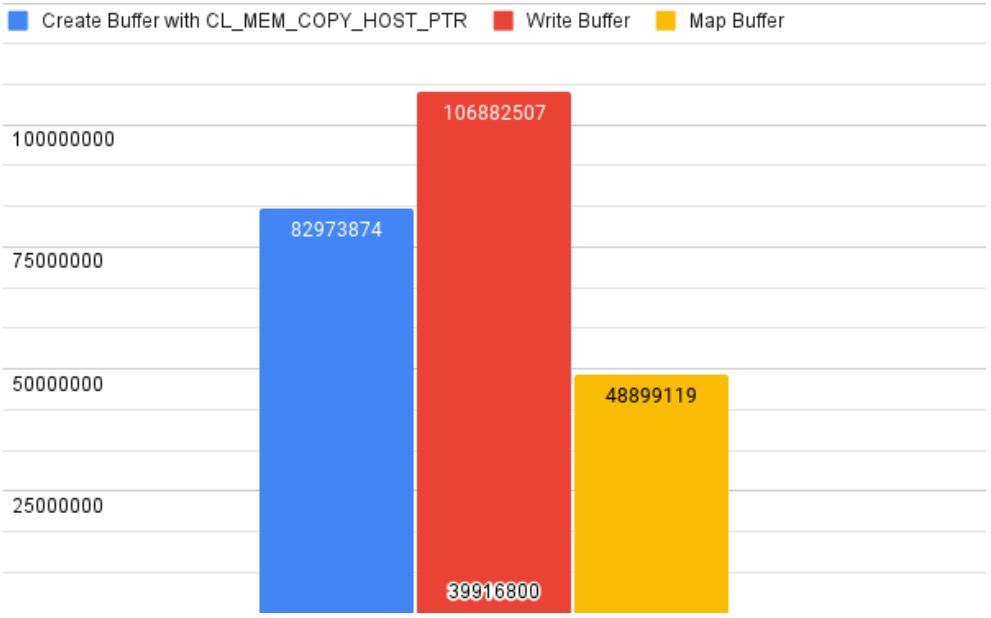
```
cl_mem d_permutations = clCreateBuffer(
    info.context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    current_number_of_permutations * (v - 1) * sizeof(int),
    permutations, &err);

```

Write Buffer

```
err = clEnqueueWriteBuffer(
    info.queue, d_permutations, CL_FALSE, 0,
    current_number_of_permutations * (v - 1) * sizeof(char),
    permutations, 0, NULL, &write_evt);
```

Map Buffer

```
clEnqueueMapBuffer(
    info.queue, d_permutations, CL_FALSE,
    CL_MAP_WRITE, 0, permutations_size,
    0, NULL, &write_evt, &err);
```



Media di permutazioni al secondo su un campione di 20 esecuzioni dei tre codici sullo stesso task di 12 vertici, per un totale di 39.916.800 permutazioni.
Si nota la differenza di prestazioni è alquanto notevole

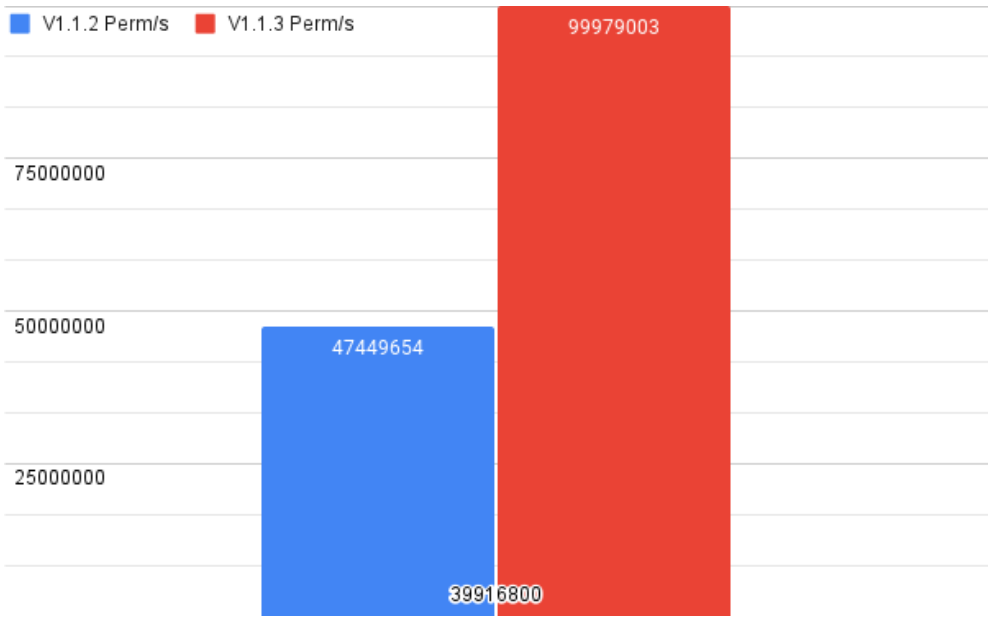
Dal grafico si evince che la scrittura tramite WriteBuffer, come si presuppone, ha prestazioni migliori rispetto alla ricreazione del buffer. Inoltre si nota che la mappatura dei dati non è una scelta corretta in caso di kernel ripetuto più volte sui dati data la necessità di rilasciare la mappatura prima di poter eseguire il kernel su quel buffer.

Sotto consiglio del professore ho cercato e testato possibili motivazioni per l'inefficienza del mapping, ed ho trovato dei fix:

- L'utilizzo del flag `CL_MAP_WRITE_INVALIDATE_REGION`, come consigliato dal professore, ha reso la chiamata `MapBuffer` immediata, le tempistiche restituiscono un risultato di 0 ms, lasciando tutto il lavoro alla chiamata `UnmapMemObject` che si occupa solamente di passare i dati così riducendo i tempi.
- Inoltre si può notare che la mappatura era eseguita su l'intera lunghezza dell'array delle permutazioni anche quando non necessario, quindi si sono ridotte le dimensioni al minimo necessario tramite il calcolo con la variabile `current_number_of_permutations`

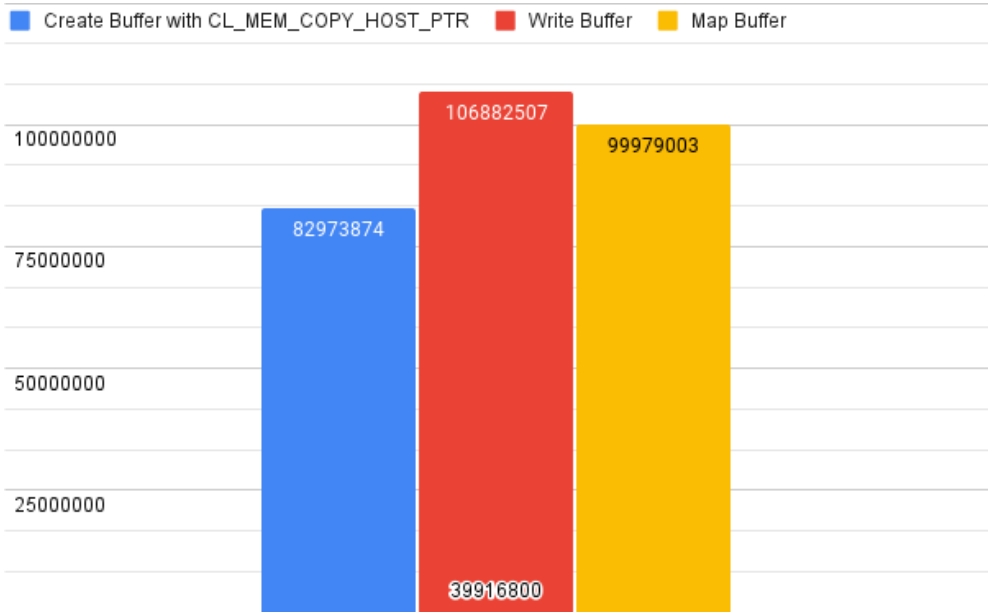
Map Buffer with CL_MAP_WRITE_INVALIDATE_REGION

```
clEnqueueMapBuffer(
    info.queue, d_permutations, CL_FALSE, CL_MAP_WRITE_INVALIDATE_REGION,
    0, current_number_of_permutations * (v - 1) * sizeof(char),
    0, NULL, &map_input_evt, &err);
```



Media di permutazioni al secondo su un campione di 20 esecuzioni dei due codici sullo stesso task di 12 vertici, per un totale di 39.916.800 permutazioni.

I risultati sono stati molto evidenti, portando il mapping quasi al livello del write

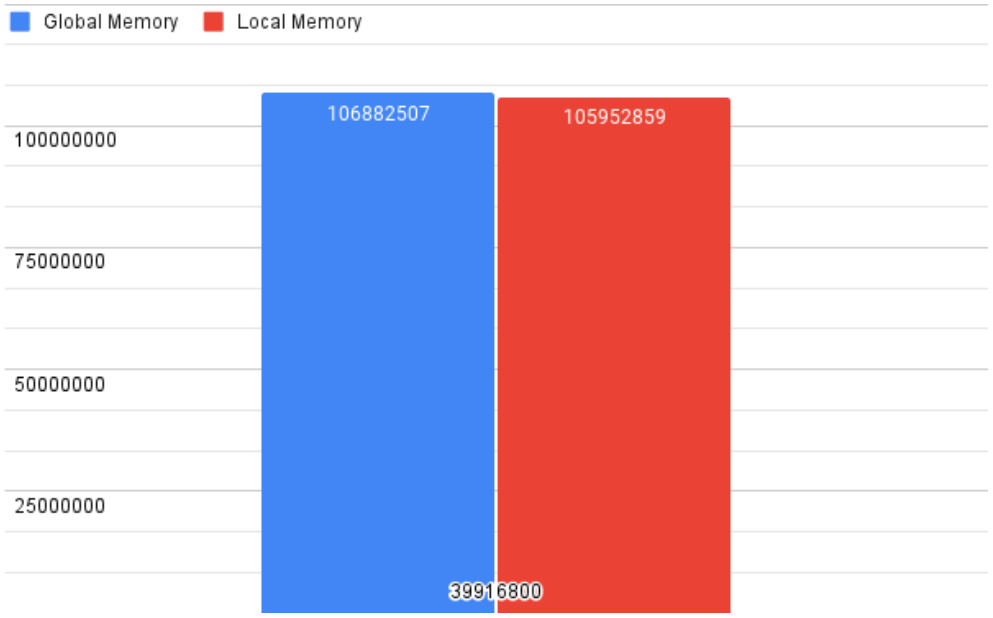


Media di permutazioni al secondo su un campione di 20 esecuzioni dei tre codici sullo stesso task di 12 vertici, per un totale di 39.916.800 permutazioni.

2.1.2 Migliorie lato device

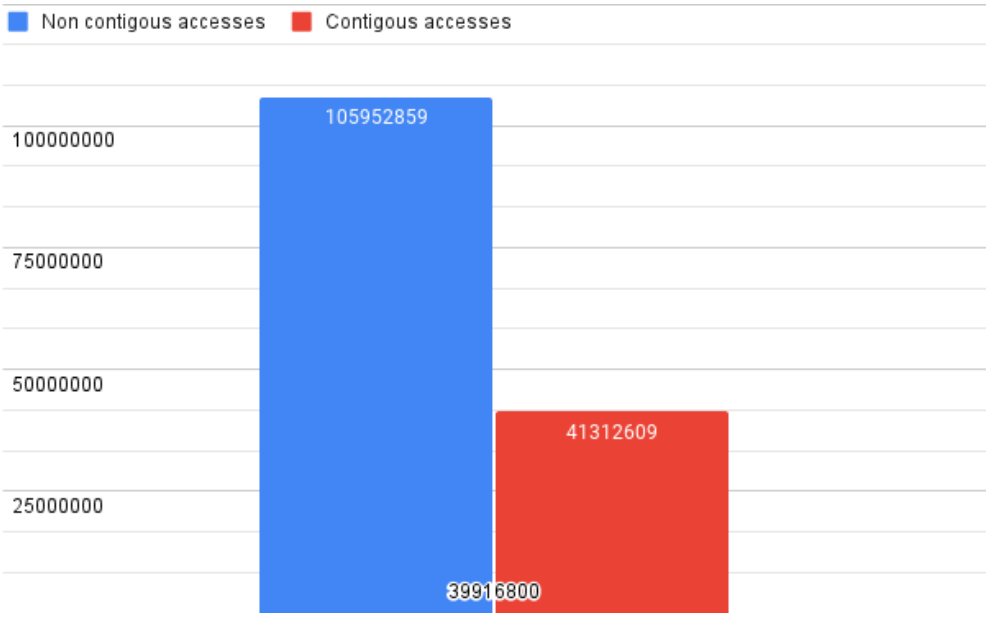
La adjacency matrix è stata utilizzata nel kernel direttamente dalla global memory fino ad ora, quindi si è tentato di spostarla in local memory per ridurre gli accessi alla global memory, ma questo ha avuto effetti avversi dato che il numero di accessi alla adjacency memory è equivalente ad un accesso per ogni vertice di una permutazione, questo è equivalente al numero di accessi eseguiti per caricare la matrice in memoria dato che viene eseguito per ogni work-group.

Quindi si è concluso che l'unico modo in cui la adjacency matrix caricata in local memory possa produrre un risultato favorevole è nel caso di un unico work-grop che lavori in sliding window su tutto il task.



Media di permutazioni al secondo su un campione di 20 esecuzioni dei due codici
sullo stesso task di 12 vertici, per un totale di 39.916.800 permutazioni.
Si nota la degradazione seppur minima dopo il caricamento in local memory della adjacency matrix

Si è tentato inoltre di rendere contigui gli accessi all'array di permutazioni, ma ha avuto effetto opposto:



Media di permutazioni al secondo su un campione di 20 esecuzioni dei due codici sullo stesso task di 12 vertici, per un totale di 39.916.800 permutazioni

2.2 Versione procedurale

Si è progettato una funzione biettiva capace di generare in modo univoco una permutazione a partire da un intero, tramite l'uso di questo metodo si ha ogni work-item far riferimento ad una permutazione tramite il proprio global id.

Il metodo utilizza una mappatura dei vertici in un dominio di scelta ordinata per indice, il dominio risultante viene riordinato tramite l'utilizzo di numeri a base incrementale che permette di descrivere ogni elemento del dominio come un numero univoco ed inoltre disporre tutti gli elementi in un range contiguo.

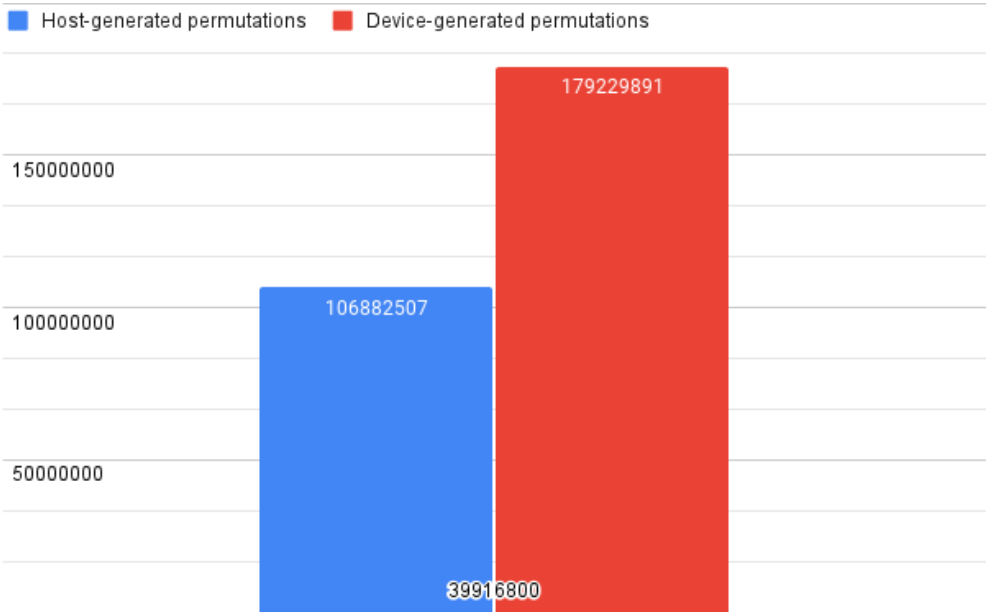
```
// Si itera sul numero di vertici da generare da questa permutazione
for (int i = n; i > 0; perm /= i--) {
  // Si esegue una normalissima operazione di estrazione della base i-esima che
  // quindi cambia ad ogni iterazione
  current = perm % i;
  // current contiene un identificativo del vertice da selezionare tra quelli rimanenti

  // Per recuperare efficientemente l'i-esimo vertice tra quelli rimasti si utilizza
  // una struttura dati di supporto che frammenta il dominio dei vertici in gruppi
  // contigui di vertici ancora disponibili
  for (int j = 0; j < n_j; j++) {
    current += start[l_id + l_size * j];
    if (current > end[l_id + l_size * j]) {
      current -= end[l_id + l_size * j] + 1;
      continue;
    }
    if (start[l_id + l_size * j] == current) {
      if (end[l_id + l_size * j] == current) {
        start[l_id + l_size * j] = 0;
        end[l_id + l_size * j] = -1;
        break;
      }
      start[l_id + l_size * j] += 1;
      break;
    }
    if (end[l_id + l_size * j] == current) {
      end[l_id + l_size * j] -= 1;
      break;
    }
    start[l_id + l_size * n_j] = current + 1;

    next[l_id + l_size * n_j] = next[l_id + l_size * j];
    next[l_id + l_size * j] = n_j;
  }
}
```

L'applicazione di questo metodo ha alcune richieste di memoria locale ed aggiunge un carico di elaborazione spostato sulla gpu, rendendo possibile l'avvio di un task su tutte le permutazioni senza caricare nella memoria del device alcuna informazione delle permutazioni.

Questo ha permesso di creare un kernel che possa elaborare tutte le permutazioni con lo stesso work-group tramite una sliding window che a sua volta rende più efficiente l'utilizzo della adjacency matrix in local memory, come sopra citato, dato che verrà caricata dall'intero work-group in memoria locale una sola volta e poi usata per tutte le permutazioni.

Per quanto riguarda il risultato, cioè la ricerca del costo minimo, si è eseguito un controllo in gpu del minimo che il work-item ha visto nelle sue permutazioni salvandolo in un array locale e poi caricarli in un array in memoria globale alla fine del processo, per poi essere letto dall'host.

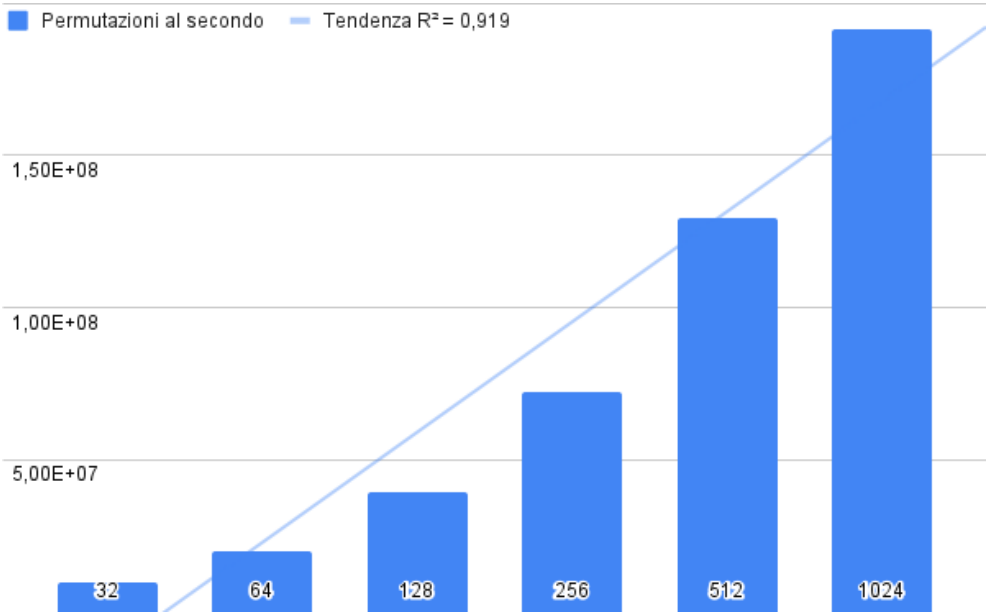
Con questo metodo si riducono tutte le tempistiche di creazione ed avvio del kernel nonché di lettura e scrittura della memoria device ottenendo un aumento prestazionale del ~150%



Media di permutazioni al secondo su un campione di 20 esecuzioni dei due codici sullo stesso task di 12 vertici, per un totale di 39.916.800 permutazioni

3 Versione finale

L'ultima versione del codice include un parametro per la scelta del work-group size sostituendo il parametro del Chunk coefficient. Inoltre si è reso più user-friendly l'output del programma dando informazioni in stdout invece che stampando i dati in un csv.



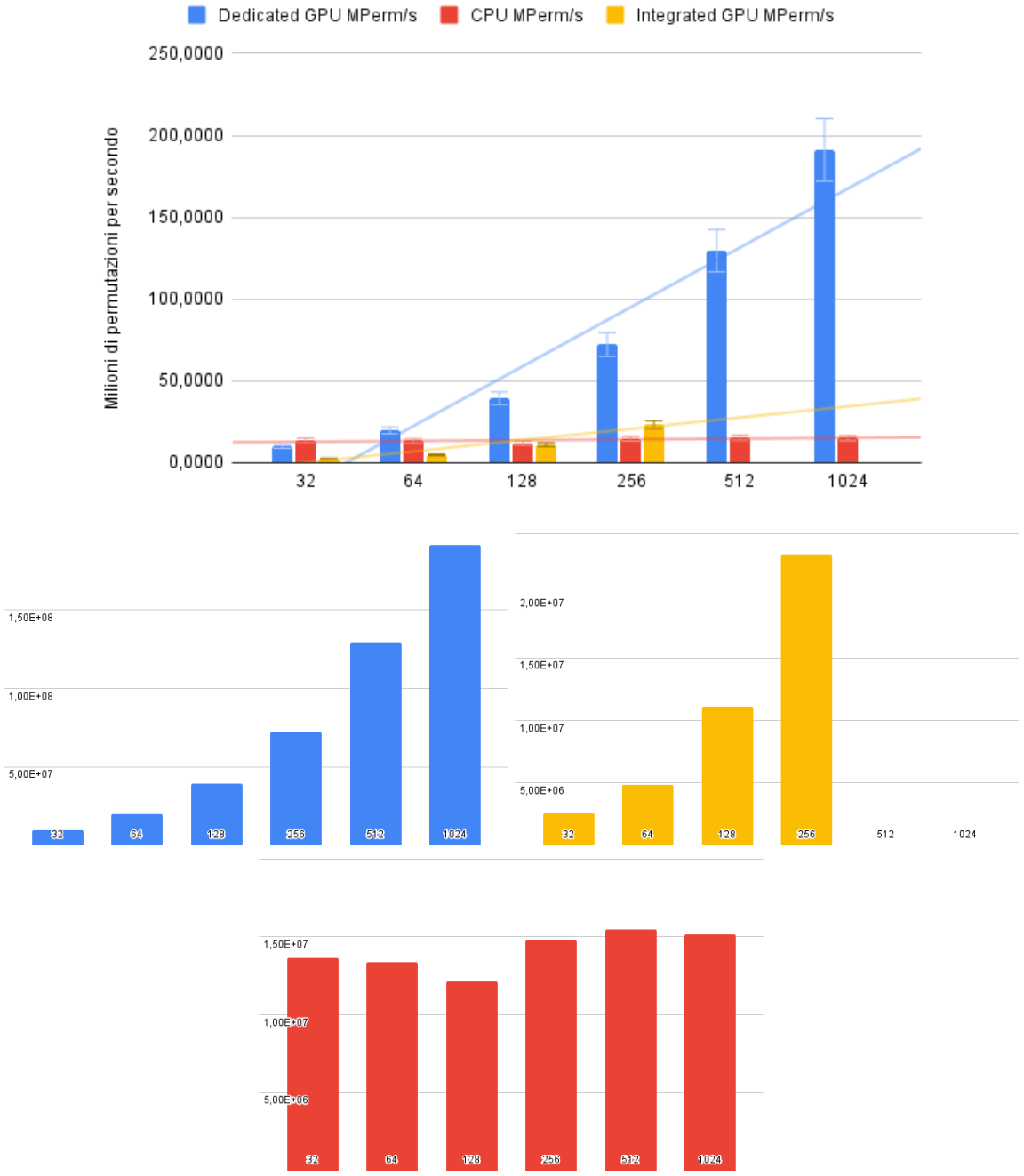
Media di permutazioni al secondo su campioni di 20 esecuzioni di task variando dimensione del grafo (da 10 a 12 vertici) e parametro lws

3 Analisi delle prestazioni su diversi device

Si riportano i risultati dei test eseguiti su diversi dispositivi:

- NVIDIA GeForce RTX 3080
- Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz
- Intel(R) UHD Graphics

Su grafi di diverse dimensioni tenendo conto del numero di permutazioni elaborate al secondo al variare del work-group size (da 32 a 1024 raddoppiando ad ogni step). I task sono stati ripetuti 20 volte.



Analizzando le prestazioni dell'algoritmo presenti nei grafici precedenti si nota come sulla cpu si abbia una maggiore varianza nelle misure, probabilmente dato dall'utilizzo del dispositivo per gestire anche il sistema operativo. La particolarità da notare è che entrambe

le gpu sembrano non aver raggiunto la saturazione nonostante siano state spinte fino a saturare il numero di work-item che possono gestire rendendo considerabile un aumento prestazionale dovuto all'hardware.

4 Considerazioni finali

Avendo trovato un metodo procedurale che permette di testare tutte le permutazioni senza duplicati, si è potuto scrivere un algoritmo quasi 100% parallelizzabile capace di elaborare centinaia di milioni di permutazioni al secondo rendendo possibile l'elaborazione di grafi con un massimo di 20 vertici per limitazioni architetturali, dato che il numero di permutazioni che si può sviluppare con 21 vertici supera 2^{64} ed è quindi impossibile eseguirlo in un unico invio del kernel, ed il metodo di generazione delle permutazioni non è capace di gestire identificativi di permutazione che richiedono più di 64 bit di memoria. E' comunque presente un limite temporale dato che elaborare un grafo con 20 vertici richiederebbe più di 21 anni per completarsi.

Riscrivere il sistema in modo da poter partizionare il numero di permutazioni in molteplici esecuzioni potrebbe rendere l'esecuzione del kernel su subset del problema possibile, ottenendo così la capacità di andare oltre i 20 vertici. Ma il limite temporale non è possibile superarlo se non con hardware ancora più potente, aumentando il livello di parallelizzazione.