



UNIVERSITÀ
DI SIENA
1240

DIPARTIMENTO DI
INGEGNERIA DELL'INFORMAZIONE
E SCIENZE MATEMATICHE

Corso di Laurea Magistrale in
ARTIFICIAL INTELLIGENCE AND AUTOMATION ENGINEERING

Realizzazione dell'Algoritmo Ungherese su Modello di Programmazione SYCL

Studenti:
Edoardo Caproni
Tommaso Quintabà

Anno Accademico 2024-2025

Indice

Introduzione	3
0.1 SYCL	3
0.2 Algoritmo Ungherese	3
0.3 Outline del processo di lavoro	4
1 Implementazione del codice in C++	5
1.1 Algoritmo Ungherese	5
1.2 Implementazione in C++	6
2 SYCL e oneAPI	9
2.1 SYCL	9
2.2 OneAPI	9
2.3 Intel Tiber AI Cloud e Jupyter Notebook	10
2.4 Modelli di Programmazione di oneAPI	10
2.4.1 Modello di piattaforma	10
2.4.2 Modello di esecuzione	10
2.4.3 Modello di memoria	12
2.4.4 Modello di programmazione kernel	12
3 Implementazione in SYCL	14
3.1 Modello ad accessori e buffer	14
3.2 Modello con Unified Shared Memory	15

4 Guida al kernel SYCL & implementazione	18
4.1 Grammatica di un kernel	18
4.1.1 Kernel di esempio	18
4.1.2 Legenda	19
4.2 Excursus sui kernel implementati	22
5 Test e risultati	26
5.1 Determinazione della dimensione ottimale dei subgroup	27
5.1.1 Conseguenze della scelta della dimensione subgroup	27
5.1.2 Presentazione dei risultati	27
5.2 Test di scaling	33
5.2.1 Strong scaling	33
5.2.2 Weak scaling	37
6 Risorse aggiuntive	41
6.1 Intel Advisor	41
6.2 Intel VTune Profiler	42
6.3 SYnergy	42
6.4 CUDA & SYCL	43
6.5 FPGA	43
7 Conclusioni	44
A Specifiche del sistema locale	47
B Specifiche del sistema hostato da Intel Tiber AI Cloud	49
Bibliografia	51

Introduzione

L’obiettivo di questo progetto è testare le capacità computazionali di SYCL, lanciando lo stesso programma parallelizzato su diversi dispositivi (acceleratori), ed eseguendo per ciascuno test di scalabilità forte e debole.

0.1 SYCL

SYCL è un modello di programmazione di alto livello sviluppato da Khronos Group, che lavora sulla grammatica del C++ (dialetto C++17). Le sue funzionalità consentono la programmazione parallela su svariati acceleratori hardware, incluse schede FPGA. Dalla versione SYCL 2020, è possibile interagire con diversi backend, lasciando alle API di accelerazione la scelta di come ottimizzare l’esecuzione del codice sulla specifica piattaforma hardware. Questo ad esempio consente compatibilità con ROCm e CUDA [8].

L’elenco di acceleratori utilizzati si trova in Appendice A, e consiste delle risorse messe a nostra disposizione dal servizio Intel Tiber AI Cloud [1], e delle nostre risorse personali su cui abbiamo installato l’ambiente di lavoro oneAPI Base Toolkit.

0.2 Algoritmo Ungherese

Come test abbiamo scelto l’Algoritmo Ungherese [3], che risolve il problema dell’assegnamento in tempo polinomiale $O(n^3)$, dove n è la dimensione della matrice quadrata di assegnamento. Abbiamo scelto questo metodo con l’intenzione di ap-

plicarlo alla risoluzione del Traveling Salesman Problem [2], ma abbiamo rinunciato a causa della scarsa affidabilità della fonte citata, e del tempo di risoluzione eccessivamente lungo (essendo il problema NP-hard). L’Algoritmo Ungherese in sé resta comunque computazionalmente significativo, e quindi adatto a testing intensivo.

0.3 Outline del processo di lavoro

Innanzitutto abbiamo realizzato una demo del codice in C++, che ci ha consentito di testare la correttezza dell’implementazione usando un dataset pre-esistente con soluzioni [5, 7].

Il passo successivo è stato di trasferire il codice in SYCL (usando Data Parallel C++). Inizialmente abbiamo generato un prototipo usando accessori per la loro facilità di implementazione. Per quanto questa versione fosse tecnicamente funzionante, la costruzione e distruzione degli accessori rallentava eccessivamente il programma, quindi abbiamo rifattorizzato il codice stavolta facendo uso di Unified Shared Memory (USM).

Finalizzato l’aspetto di programmazione, abbiamo lanciato una batteria di test per individuare la sub_group size migliore per ogni dispositivo, e poi abbiamo concluso con i test di scalabilità.

Capitolo 1

Implementazione del codice in C++

Questa versione del codice scritta in C++ è intesa solo come prototipo funzionante dell’Algoritmo Ungherese (allegato alla relazione). Di seguito è fornita una spiegazione sommaria delle varie componenti.

Cogliamo anche l’occasione per dimostrare il nostro disappunto nei confronti della pubblicazione IEEE [2], che pensavamo potesse portarci ad applicare l’algoritmo alla risoluzione del Traveling Salesman Problem. La loro proposta però è equivalente a un brute forcing: partendo dalla soluzione ottimale dell’Ungherese, testano tutte le soluzioni subottimali fino a cadere in quella ottimale per TSP. Questo è possibile perché l’Ungherese è un rilassamento del TSP, ma ciò non significa affatto che i due problemi abbiano la stessa complessità computazionale.

1.1 Algoritmo Ungherese

L’algoritmo Ungherese in breve si compone di quattro passaggi [10]:

1. **Sottrazione del minimo per riga**
2. **Sottrazione del minimo per colonna**

3. **Copertura di tutti gli zeri col minor numero di righe:** Durante questo passaggio, se il numero di linee richieste per coprire tutti gli zeri è uguale a n , esiste un assegnamento ottimo e l'algoritmo si ferma. Se le linee di copertura sono meno di n , si prosegue col passo successivo.
4. **Creazione di zeri aggiuntivi:** Si sceglie l'elemento minore tra quelli non coperti dalle linee, quindi si sottrae a tutti gli elementi scoperti e si somma a quegli elementi che sono coperti da due linee. Quindi si ripete dal passaggio precedente.

1.2 Implementazione in C++

La seguente è una spiegazione in maggior dettaglio dell'implementazione da noi realizzata in C++, usata successivamente come modello per l'implementazione in SYCL. È stata di ispirazione la repository di Fernando B. Giannasi [9].

Premessa: matrice quadrata ($n \times n$) a numeri interi non negativi.

1. **Preprocessing:** Prima di avviare il ciclo principale, occorre fare riduzione della matrice prima per righe e poi per colonne. Questo vuol dire trovare il minor valore per riga/colonna, e sottrarlo a tutta la riga/colonna. In questo modo garantiamo uno zero in ogni riga e colonna. Per finire il preprocessing si esegue una sola volta **starring the zeroes**, per poi entrare nel ciclo principale con **find prime and uncover star**.
2. **Starring the zeroes:** Si scopre l'intera matrice (se era stata coperta), poi riga per riga trova il primo zero scoperto e lo segniamo come stella, di cui copriamo la colonna. Le stelle saranno parte dell'assegnamento ottimale, quindi quando ne abbiamo n (o abbiamo n colonne coperte), saltiamo a **optimal assignment**. Altrimenti si procede a **find prime and uncover star**.

3. **Find prime and uncover star:** Cerchiamo il primo zero scoperto della matrice, e lo segniamo come prime; se non lo troviamo si salta a **step toward optimality**. Se viene trovato, cerchiamo una stella nella stessa riga del prime: nel caso venga trovata, scopriamo la colonna della stella e copriamo la riga del prime, poi si ripete questo punto da capo; se la stella non viene trovata si passa ad **alternating path**, fornendo le coordinate del prime come inizio del path.
4. **Step toward optimality:** Troviamo il minimo scoperto della matrice, lo sottraiamo a tutte le celle scoperte della matrice, e lo sommiamo a tutte quelle coperte due volte. Poi si procede a **starring the zeroes**, e si ripete il ciclo principale.
5. **Alternating path:** Costruiamo l'alternating path (perché alterna tra guardare un prime e una stella nel suo funzionamento). Su questo "percorso" risolviamo il corrispondente maximal matching problem, con un augmenting path. Il risultato è una matrice in cui le stelle vengono scorate, e i prime diventano stelle. Controlliamo se l'assegnamento ottimo sia possibile, in tal caso si salta a **optimal assignment**; alternativamente ripetiamo il ciclo principale su questa nuova disposizione di stelle (**find prime and uncover star**)
6. **Optimal assignment:** Segniamo le coordinate di ogni stella: queste rappresentano l'assegnamento ottimo. Inoltre calcoliamo il costo di tale assegnamento sommando i termini indicati da ogni stella.

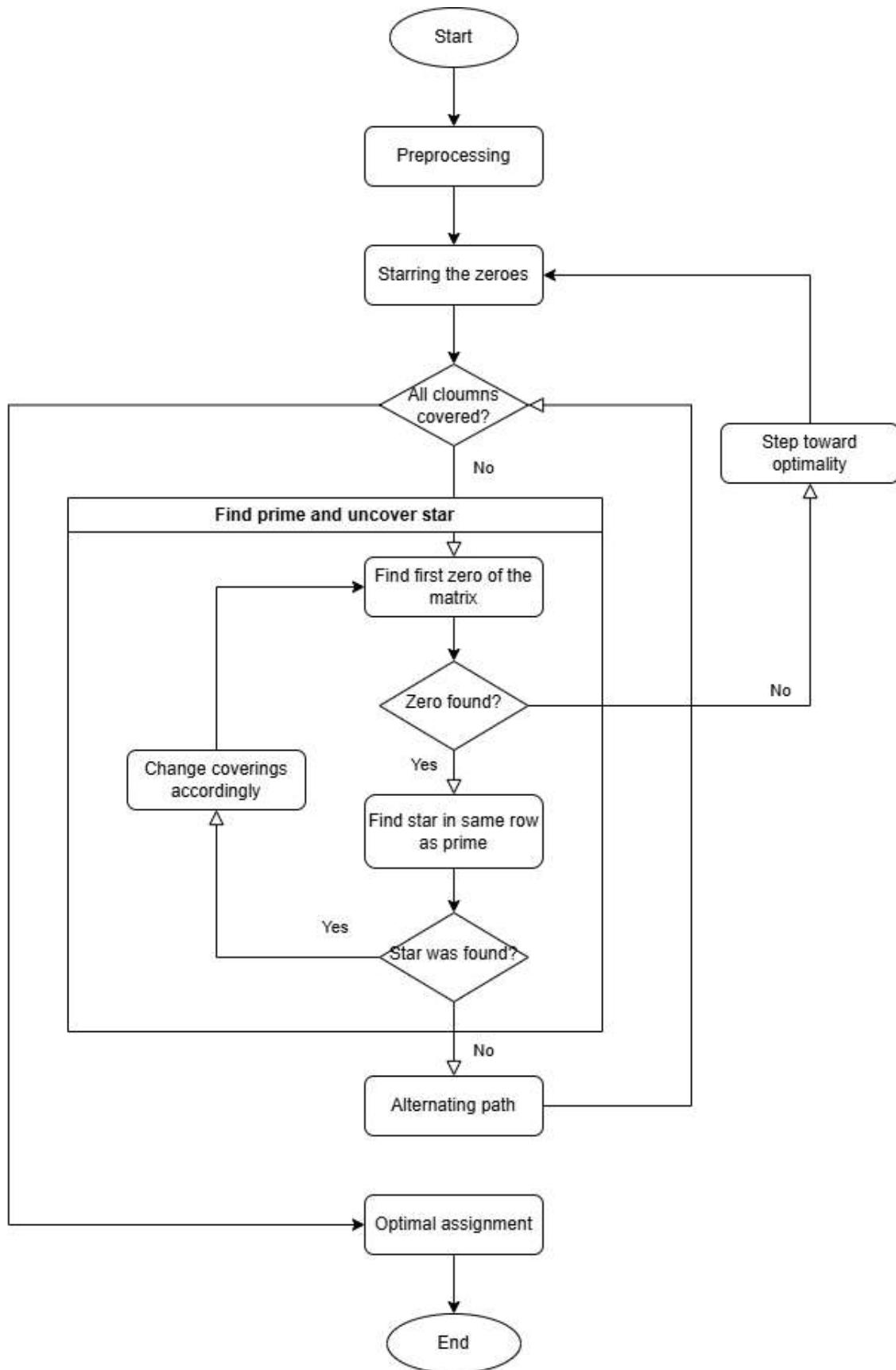


Figura 1.1: Algoritmo Ungherese

Capitolo 2

SYCL e oneAPI

2.1 SYCL

Come già introdotto, SYCL è un layer di astrazione royalty-free sviluppato da Khronos Group che consente di programmare in architetture eterogenee usando un'estensione dello standard C++ 17. Supporta GPU NVIDIA, AMD [8] e Intel, oltre a CPU e FPGA, gestendo memoria e risorse computazionali dei dispositivi.

Alternative a SYCL sono CUDA (modello di programmazione specifico per GPU NVIDIA), HIP (API open source e gratuito, consente di convertire codice CUDA in un programma C++ compilabile in GPU AMD e NVIDIA), oltre a OpenMP, OpenACC e OpenCL.

Il vantaggio di SYCL è l'eterogeneità di esecuzione, con performance comunque comparabili a CUDA [11].

2.2 OneAPI

OneAPI è un pacchetto di strumenti di sviluppo ottimizzati per piattaforme Intel, ma basati su standard aperti a programmazione eterogenea (aderendo a United Acceleration Foundation) [12]. In oneAPI Base Toolkit è presente un compilatore per

Data Parallel C++, quindi compatibile con SYCL, e vari strumenti di supporto alla programmazione ad alta performance, tra cui Intel VTune Profiler e Intel Advisor.

2.3 Intel Tiber AI Cloud e Jupyter Notebook

Intel mette a disposizione risorse di computazione attraverso un servizio chiamato Intel Tiber AI Cloud. Il servizio offre anche un pacchetto di lezioni sulla programmazione in SYCL forniti sulla piattaforma Jupyter Notebooks. Non ci è possibile citare direttamente i contenuti di tali corsi in quanto richiedono un account Intel abilitato al Intel Tiber (a noi possibile attraverso account istituzionale UNISI).

2.4 Modelli di Programmazione di oneAPI

Tutti i seguenti modelli sono basati sullo standard SYCL.

2.4.1 Modello di piattaforma

Specifica un host (tipicamente un sistema basato su CPU) che può controllare uno o più device (acceleratori). L'host coordina e controlla il lavoro che viene dato in offload ai device, che contengono unità computazionali per la parallelizzazione (Figura 2.1).

2.4.2 Modello di esecuzione

Definisce come il codice parallelizzato (chiamato kernel) venga eseguito nei device e interagisca con l'host. La coordinazione dell'esecuzione e gestione della memoria è lasciata ai gruppi di comando (gruppi di invocazioni di accessori e kernel), raccolti a loro volta da una coda. Le code possono essere a loro volta eseguite in-order o out-of-order.

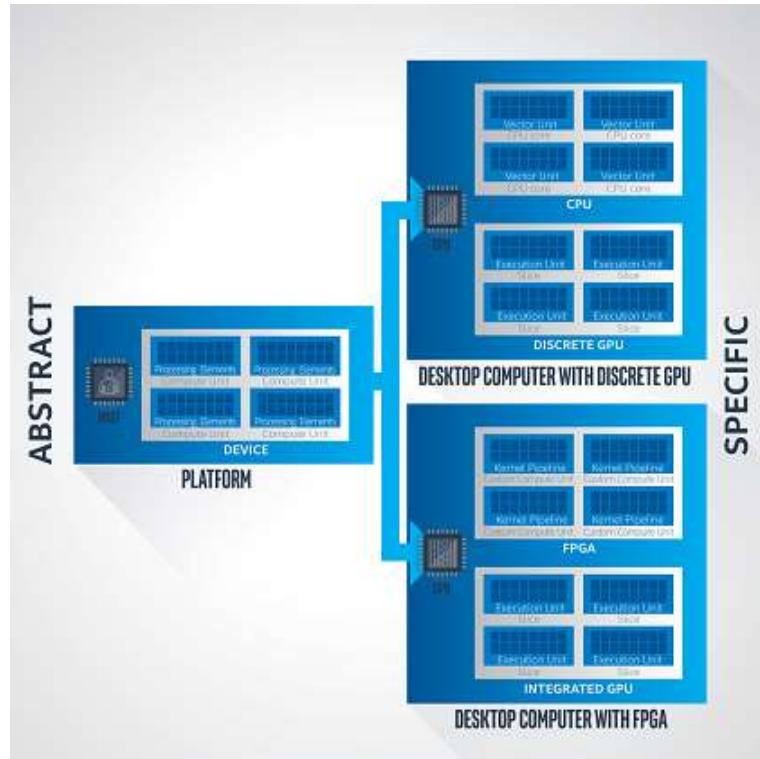


Figura 2.1: Modello di piattaforma

La computazione parallela viene gestita attraverso un’astrazione organizzata gerarchicamente: nel kernel occorre definire il numero totale di work-item (o thread) che si desidera istanziare, e la dimensione dei gruppi di lavoro (work-groups), che deve essere divisore intero della prima quantità.

I work-item sono raggruppati innanzitutto in sub-group, e vengono processati contemporaneamente come SIMD vectors (le GPU Intel posseggono Vector Engines, ovvero processori SIMD multithread) [13]. Il livello di gerarchizzazione successivo è il work-group: un raggruppamento 1-, 2- o 3-dimensionale (ND-range) i cui work-item sono sincronizzabili tra di loro. La sincronizzazione tra work-group differenti può essere ottenuta nativamente solo alla chiusura del kernel. Il livello di astrazione più alto è il global range, che deve essere di dimensione uguale o superiore ai work-group, e definisce il numero totale di work-item (Figura 5.4).

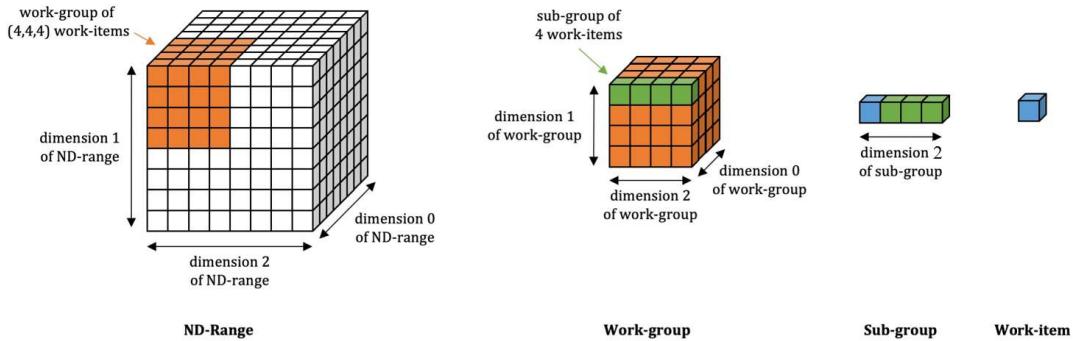


Figura 2.2: Modello di esecuzione

2.4.3 Modello di memoria

Determina l'allocazione e la gestione della memoria da parte di host e device.

Secondo il modello Unified Shared Memory, la memoria può essere allocata attraverso puntatori come tre tipologie: host, device o shared (la più lenta). L'accesso alla memoria device o shared si ottiene attraverso accessori. [14] Alternativamente, esiste il modello a buffer: contenitori 1-, 2- o 3-dimensionalni a cui sia device che host possono accedere attraverso accessori. I buffer sono gestiti in runtime da SYCL, le cui API gestiscono l'allocazione, lettura, scrittura, mentre il runtime SYCL sposta i dati tra host e device, e sincronizza gli accessi ai dati [15]. Le dipendenze tra diverse kernel sono risolte implicitamente.

Abbiamo trovato che questo secondo approccio, di più semplice implementazione, sia macchinoso nella scrittura e comporti una notevole perdita di performance rispetto a USM.

2.4.4 Modello di programmazione kernel

Consente parallelismo esplicito tra host e device; esplicito intendendo che l'offload del codice sull'acceleratore non è automatico, ma esplicitato dallo sviluppatore.

Le porzioni di codice accettate da host e device possono coesistere assieme, ma sono soggette a diverse regole grammaticali di C++.

La porzione host del codice deve selezionare il device e invocare la coda di esecuzione, e specificare le allocazioni di memoria (o buffer); i kernel, aggiunti uno a uno alla coda, sono funzioni lambda che possono essere eseguite sequenzialmente (*single_task*) o in parallelo (*parallel_for*) dal device.

Capitolo 3

Implementazione in SYCL

3.1 Modello ad accessori e buffer

Data la nostra inesperienza con la programmazione SYCL e la relativa semplicità di implementazione del modello a buffer, abbiamo scelto quello nel portare il nostro prototipo da C++ a SYCL.

Come già menzionato, abbiamo trovato che questo modello è più adatto a programmi con un singolo kernel, o comunque programmi in cui l'intero flusso di lavoro è eseguito nell'acceleratore. In questo caso, una volta costruiti accessori e buffer, essi rimangono validi per l'intera durata del programma.

Nella nostra situazione invece il codice parallelizzabile è interlacciato con codice sequenziale (questo a causa di operazioni di cui non è possibile fare offload, come logica di controllo, o che semplicemente non sono parallelizzabili). Questo, unito alla grande dimensione dei dati in uso, rende gli accessori estremamente inefficienti, come confermato dalla *oneAPI GPU Optimization Guide* [15].

La conclusione è che sebbene questo approccio fosse una tappa importante del progetto, non ci sentivamo soddisfatti nel considerarlo come meta finale, nonostante sia perfettamente funzionante per dimensioni di matrici relativamente piccole: sopra certe taglie, la richiesta di memoria era eccessiva, e non potevamo allocarle usando questo paradigma. Lo lasciamo comunque in allegato in caso di necessità di

consultazione.

Da notare che durante questo stadio di lavoro, stavamo usando le risorse computazionali di un servizio di Intel chiamato Devcloud, che senza avviso alcuno è stato chiuso e migrato verso la già citata piattaforma Intel Tiber AI. La transizione ci ha creato non pochi problemi, dovendo rifare e rendere compatibili con il nuovo paradigma i nostri account Intel. Un'altra perdita è stata la possibilità di testare il progetto su FPGA: su Devcloud avevamo a disposizione un emulatore, ma questo è assente su Tiber AI.

3.2 Modello con Unified Shared Memory

Il passaggio da buffer a USM ci ha consentito non solo di raggiungere migliori prestazioni, ma ci ha anche aperto la possibilità di usare appieno il nostro dataset. Infatti la più grande matrice a nostra disposizione ha dimensioni 800 x 800, e l'implementazione con USM consente di usare la piena dimensione dei gruppi di lavoro consentita dall'hardware (1024 x 1024).

Un ulteriore vantaggio alle prestazioni è dato da una migliore gestione dello scope delle variabili: mentre accessori e buffer devono essere costruiti nello scope di una funzione (e alla chiusura dello scope vengono distrutti), i puntatori offerti dal modello USM possono essere dichiarati nel main, e passati come argomenti. Così facendo si eliminano i tempi di allocazione e deallocazione della memoria, che sono deleteri specialmente con il modello a buffer [16].

A questo punto avevamo un altro punto di insoddisfazione: il tempo di esecuzione in CPU era spesso più rapido di quello GPU, mentre intuitivamente ci saremmo aspettati il contrario. Con il crescere della dimensione delle matrici, questa relazione si va ad invertire, con differenze di velocità non particolarmente marcate.

Abbiamo pensato che la causa fosse il tempo di esecuzione legato alla parte sequenziale del codice, o in altre parole, che lo speedup offerto dalla GPU non fosse sostanziale perché la parte di codice parallelizzata era troppo piccola. Test usando

Intel Advisor parevano darci ragione su questa idea, e quindi, per creare uno scenario in cui le differenze di tempi tra CPU e GPU fossero più marcate, ci siamo proposti di generare matrici più larghe, per aumentare il tempo di esecuzione relativo alla parallelizzazione.

Sfortunatamente, così facendo la nostra implementazione eccedeva i limiti imposti dall'hardware, richiedendo un altro refactoring del codice per superare questo vincolo. Abbiamo esplorato due diverse opzioni: una in cui lo stesso work-item operasse (sequenzialmente) su più di una cella della matrice (o vettore), con una relazione uno-a-molti; questa idea è stata scartata per difficoltà di implementazione della logica di pooling dei risultati.

L'altro approccio è stato di istanziare più gruppi di lavoro, in modo che coprano l'intera dimensione della matrice (o vettore), con l'ultimo gruppo di lavoro più piccolo per evitare sovra-allocazione delle risorse. Sfortunatamente nella maggior parte delle GPU non è possibile creare disomogeneità nelle dimensioni dei gruppi di lavoro: quindi abbiamo mantenuto la dimensione costante, aggiungendo una maschera che impone inattività ai work-item che sono in eccesso. Questo approccio mantiene una relazione uno-a-uno molto semplice da concettualizzare, e quindi l'abbiamo mantenuta e applicata all'intero progetto.

L'ultima modifica è nata da un esperimento con l'uso di memoria locale (dell'acceleratore) invece che condivisa. Questo ha portato a un notevole speedup, e ha anzi portato i tempi di esecuzione CPU vs GPU in linea con le nostre aspettative. Questo perché la memoria condivisa è molto più lenta di quella locale; quindi abbiamo cambiato filosofia di allocazione, usando strettamente memoria locale (dell'acceleratore), o (locale dell') host. Questo in effetti rende inutile l'uso di matrici di dimensioni sopra 1024 x 1024.

Su GPU questo impatta leggermente la velocità di computazione perché aggiungiamo branching, che crea divergenza nell'ultimo work_group (che copre solo parzialmente la matrice o il vettore da parallelizzare) [18]. Tutti i work_group i cui work_item

sono unanimi nella scelta del branch non ricevono questa penalizzazione, così come nel caso di esecuzione su CPU. Dunque abbiamo reputato di mantenere questa feature perché la perdita di performance non era significativa, dando più rilevanza alla possibilità di generalizzazione a dimensioni più grandi dei dati.

Capitolo 4

Guida al kernel SYCL & implementazione

4.1 Grammatica di un kernel

Di seguito proponiamo un kernel d'esempio (tratto dalla funzione *row_reduction*) e relativa legenda per spiegare la grammatica SYCL. Sia chiaro che questo esempio non è esaustivo né delle tecniche complessivamente usate nel progetto, né degli strumenti messi a disposizione dal linguaggio.

4.1.1 Kernel di esempio

```
void row_reduction(int* matrix, sycl::queue& Q) {
    int* row_min = sycl::malloc_device<int>(N, Q);
    Q.memcpy(row_min, &MAX_INT, N * sizeof(int)).wait();

    Q.submit([&](sycl::handler& h) {
        h.parallel_for(
            sycl::nd_range<2>(
                sycl::range<2>(N, num_work_groups * local_size),
```

```

    sycl::range<2>(1, local_size)
) ,
[=](sycl::nd_item<2> item)
[[ intel::reqd_sub_group_size(SUB_GROUP_SIZE) ]] {
    size_t row = item.get_global_id(0);
    size_t col = item.get_global_id(1);

if (row < N && col < N) {
    auto atomic_row_min = sycl::atomic_ref<int ,
        sycl::memory_order::acq_rel ,
        sycl::memory_scope::device ,
        sycl::access::address_space::global_space>
    (row_min[row]);
    atomic_row_min.fetch_min(matrix[row * N + col]);
}
};

};
} ).wait();

[...]

    sycl::free(row_min, Q);
}

```

4.1.2 Legenda

- **{int* row_min = sycl::malloc_device<int>(N, Q);}**

Creazione di un puntatore a memoria locale (del device), templatizzato come intero, lungo N celle contigue della dimensione templatizzata; Q è la coda di esecuzione, inizializzata precedentemente nel main.

- `Q.memcpy(row_min, &MAX_INT, N * sizeof(int)).wait();`

Inizializzazione del puntatore con un valore esterno dichiarato precedentemente di dimensione compatibile; il metodo `.wait()` rende la funzione bloccante, ovvero impedisce l'esecuzione della prossima operazione in coda prima che quella attuale sia terminata.

- `Q.submit([&](sycl::handler& h) {`

Il metodo `.submit` è una funzione lambda che genera un handler, che a sua volta gestisce il kernel. Immediatamente dopo questa espressione possono essere introdotte variabili con scope ridotto al solo gruppo di lavoro, attraverso un `local_accessor`; è anche possibile dichiarare uno stream di output testuale in uscita dal kernel, utile al debugging del kernel

- `h.parallel_for(`

Esistono vari tipi di kernel: noi usiamo `parallel_for` per esecuzione in parallelo, e `single_task` per esecuzione seriale che però deve essere svolta nell'acceleratore.

Entrambe sono funzioni lambda.

- `sycl::nd_range<2>(`
`sycl::range<2>(N, num_work_groups * local_size),`
`sycl::range<2>(1, local_size)`
`) ,`

L'`nd_range` fornito come argomento descrive la dimensione globale (`global_range`), in questo caso bidimensionale, e la dimensione del `work_group`. Notare come il numero di dimensioni debba essere lo stesso, e la dimensione globale debba essere multipla intera di quella del `work_group`.

- `[=](sycl::nd_item<2> item)`

Qui si definisce il work_item, un oggetto che rappresenta il singolo thread, e che ha varie proprietà (come indici e gerarchie, spiegate nell'introduzione).

- `[[intel::reqd_sub_group_size(SUB_GROUP_SIZE)]] {`

Questo argomento serve per imporre una specifica dimensione di subgroup; rimuoverlo signifca lasciare che il compilatore faccia una sua valutazione sulla dimensione ottimale in base alle caratteristiche del dispositivo e del kernel.

- `size_t row = item.get_global_id(0);
size_t col = item.get_global_id(1);`

Estraiamo gli indici globali del work_item, ovvero nel sistema di riferimento dettato dal global_range. Alternativamente si possono estrarre indici locali, ovvero nel sistema di riferimento del work_group; in questo caso occorre anche esplicitare l'indice del work_group per mantenere l'univocità del work_item.

- `if (row < N && col < N) {`

La condizione è necessaria in quanto il global size è più largo della matrice: un certo numero di celle devono essere rese inattive, altrimenti si rischiano accessi in memoria non voluti, e consequentemente segmentation faults o comportamento aberrante.

- `auto atomic_row_min = sycl::atomic_ref<int ,
sycl::memory_order::acq_rel , sycl::memory_scope::device ,
sycl::access::address_space::global_space>(row_min[row]);`

Generazione di una variabile temporanea atomica, spesso invocata in questo progetto per evitare race conditions. Alcuni metodi le richiedono obbligatoriamente per poter funzionare correttamente.

- `atomic_row_min.fetch_min(matrix[row * N + col]);`

La variabile atomica viene popolata da un operatore di confronto e assegnazione. L'effetto complessivo del kernel è dunque di trovare il minimo valore per ogni riga (su cui verranno successivamente fatte altre operazioni).

- `}).wait();`

Al solito, `.wait()` viene utilizzato per evitare l'esecuzione del kernel successivo anzitempo. Ciò ha l'effetto aggiuntivo di sincronizzare tra loro tutti i `work_groups`, cosa non possibile all'interno del kernel con l'attuale grammatica SYCL.

- `sycl::free(row_min, Q);`

La funzione viene chiusa deallocando esplicitamente la memoria, per evitare memory leak.

4.2 Excusus sui kernel implementati

Di seguito esponiamo i kernel da noi scritti, cercando di spiegare brevemente come funzionino, e le eventuali limitazioni alla parallelizzazione.

- **row_reduction** Il minimo per riga viene trovato atomicamente con il metodo `fetch_min`, e esposto fuori dal kernel attraverso allocazione di memoria locale. Un secondo kernel lo sottrae a ciascun elemento della riga.
- **col_reduction** Stesso procedimento descritto sopra. Notare come non ci sia bisogno di dichiarare il global size verticalmente nonostante si proceda per colonna: la topologia del global size e dell'oggetto su cui si effettua la parallelizzazione non debbono per forza coincidere.

- **starring_the_zeroes** Riga per riga vogliamo trovare il primo zero scoperto, e segnarcelo. Poiché in questa operazione le iterazioni successive sono influenzate da quelle precedenti, siamo costretti a lanciarle sequenzialmente con un semplice ciclo for. Ad ogni ciclo istanziamo un kernel che controlla la riga in parallelo: segniamo l'indice di ogni work_item corrispondente ad una cella nulla e scoperta, troviamo il minimo di ogni work_group attraverso un'operazione di riduzione, e infine filtriamo i risultati con un fetch_min, risultando nel minimo indice dell'intera riga. Nota: possiamo usare la riduzione in questo caso perché coinvolge tutti gli elementi di un gruppo; nella maggior parte dei nostri kernel vogliamo controllare gli elementi di un gruppo che rispettano determinate condizioni, e ignorare gli altri, quindi non è applicabile.

Prima di passare alla riga successiva del ciclo for, un secondo kernel aggiorna la maschera e segna la copertura della colonna attraverso un single_task: nonostante questa sia un'operazione sequenziale, è più veloce farla eseguire all'acceleratore piuttosto che spostare i dati nell'host ed eseguirla lì.

- **all_columns_covered** Il primo kernel di questa funzione copre l'intera matrice maschera; se un work_item corrisponde con una stella, quel work_item segna la colonna come coperta.

Il secondo kernel controlla il vettore di coperture: viene inizializzato un flag booleano a vero, e se un elemento del vettore risulta scoperto, cambia il valore del flag a falso. Se il flag si mantiene vero all'uscita dal kernel, tutte le colonne della matrice sono coperte, e questo indica che la soluzione è stata trovata.

- **find_prime_and_uncover_star** Questa funzione contiene due kernel, ma sono entrambi brevi single task. Infatti si occupa di controllare le condizioni di esecuzione delle funzioni **find_uncovered_zero** e **there_is_star_in_row**, e, come già menzionato, è più veloce effettuare operazioni sequenziali sull'acceleratore quando i dati su cui si opera sono allocati in memoria locale.

- **find_uncovered_zero** Trova il primo zero scoperto linearizzando le coordinate di ogni cella, in modo che siano considerate in ordine lessicografico. A questo punto è facile trovare il minimo indice che corrisponde ad una cella nulla e scoperta.
- **there_is_star_in_row** Dovendo controllare una sola riga della maschera, è sufficiente un range globale unidimensionale, e ogni cella corrispondente ad una stella scrive (atomicamente) su un flag booleano di uscita.
- **step_towards_optimality** Il primo kernel cerca la cella scoperta della matrice con valore minimo attraverso fetch_min e scrittura atomica.
Il secondo kernel somma o sottrae quel minimo a determinate celle (a seconda del loro stato di copertura) attraverso fetch_add e scrittura atomica.
- **alternating_path** Innanzitutto lanciamo un kernel per inizializzare alternating_path a N; non possiamo usare memcpy() o memset() perché la prima riga deve essere inizializzata diversamente. Potenzialmente potevamo ottenere marginalmente migliore velocità di esecuzione usando memset(), e poi lanciando un kernel con single_task che cambiasse solo la prima riga.
Il resto della funzione, oltre a un kernel che rimuove le stelle dalla maschera, è logica di controllo che lancia le funzioni **find_star_in_col**, **find_prime_in_row** e **augment_path**.
- **find_star_in_col** Questo è un semplice kernel unidimensionale che controlla una riga della matrice, e restituisce l'indice di una stella se presente.
- **find_prime_in_row** Similmente, questo è un kernel unidimensionale che controlla una colonna della matrice, e restituisce l'indice di un prime se presente.
- **augment_path** Questo kernel è interessante perché la sua dimensione viene calcolata durante il runtime, a seconda della dimensione del path. Per il resto l'operazione è molto semplice: prima si controlla se ogni cella contiene una

stella, e in tal caso il relativo work_item la rimuove; successivamente si controlla se la stessa cella contiene un prime, e in tal caso lo si cambia in stella.

- **optimal_assignment** Contiene l'ultimo kernel del progetto, e controlla salva tutte le coordinate globali delle celle contenenti una stella, oltre a sommare i valori di tali celle (atomicamente, con fetch_add) per trovare il costo totale.

Capitolo 5

Test e risultati

Preambolo:

In tutti i test espressi in questa sezione, lasciamo implicite le seguenti assunzioni:

- l'incertezza sulle misure è espressa come deviazione standard.
- rispettiamo le caratteristiche fisiche dei dispositivi (vedi Appendice), vale a dire:
 - massima dimensione globale (ovvero massimo numero di work_item istanziabili);
 - massima dimensione dei work_group (ovvero quanti work_item possono popolare un work_group);
 - dimensione raccomandata del work_group: generalmente multipla di 2, e più larga possibile;
 - dimensioni di sub_group compatibili con l'hardware (CPU accetta più dimensioni di GPU);
 - massimo numero di sub_group per ciascun work_group.

5.1 Determinazione della dimensione ottimale dei subgroup

5.1.1 Conseguenze della scelta della dimensione subgroup

In SYCL è possibile scegliere la dimensione dei subgroup, che sono il primo livello di aggregazione gerarchica dei work_item. Questa scelta influenza la larghezza dei SIMD vector, che sono eseguiti contemporaneamente dall'architettura hardware. A seconda della scelta di subgroup size, si ottengono diverse percentuali di occupazione delle risorse fisiche dell'acceleratore, quindi ci si pone la questione di quale sia il dimensionamento ottimale.

È possibile tentare una previsione teorica della percentuale di occupazione [17], ed esistono persino strumenti di stima per le GPU, dato il modello [19]. Sfortunatamente queste stime si rivelano imprecise all'atto pratico, sia perché non colgono necessariamente le complessità dell'effettiva computazione (ad esempio la gestione della memoria e sub-processori in una CPU multi-core), sia perché un aumento di occupazione delle risorse non comporta necessariamente uno speedup.

Resta sempre la via sperimentale, che è stato esattamente il nostro percorso.

5.1.2 Presentazione dei risultati

Raccogliamo quindi nei seguenti grafici e tabelle i risultati dei tempi di esecuzione del nostro programma, testando su 4 diversi acceleratori, 8 dimensioni crescenti delle matrici, e tutte le subgroup size disponibili per ogni dispositivo. In tutti i dispositivi è stata fissata unanimemente la dimensione di work_group a 1024, essendo la più alta dimensione compatibile tra tutti i dispositivi. Per ogni dispositivo forniamo i dati sia in forma tabellare, sia in forma grafica.

Da notare che, per motivi di tempo, non è stato possibile raccogliere tutti i risultati con lo stesso grado di precisione: alte dimensioni delle matrici (specialmente su CPU) impongono tempi di esecuzione drammatici: basti pensare che ogni test

su una 800 x 800 avrebbe richiesto 18 ore. Inoltre le risorse computazionali messe a disposizione da Intel hanno un timer massimo oltre il quale occorre ricaricare la sessione.

Infine, l'alto numero di test non migliora necessariamente la precisione della misura: poiché non abbiamo pieno controllo delle risorse assegnateci, abbiamo trovato che il tempo di esecuzione segue una distribuzione multimodale, a seconda di quanto il servizio Tiber AI potesse mettere a disposizione in un dato momento.

Per tutti questi motivi, il numero di esecuzioni considerato per ogni misura varia nel modo seguente:

- tutti i risultati su GPU sono stati ottenuti con una media di 500 esecuzioni;
- i risultati su CPU sono stati ottenuti dalla media di un diverso numero di esecuzioni a seconda della dimensione della matrice:
 - per matrici di dimensione inferiore o uguale a 300: 500 esecuzioni;
 - per matrici di dimensione compresa tra 400 e 600: 200 esecuzioni;
 - per matrici di dimensione superiore o uguale a 700: 100 esecuzioni.

Matrice	SGS 4	SGS 8	SGS 16	SGS 32	SGS 64
100x100	672,16 ± 34,38	680,25 ± 30,87	676,95 ± 26,84	682,26 ± 32,12	680,28 ± 32,77
200x200	3367,01 ± 118,43	3414,16 ± 100,09	3489,58 ± 93,36	3451,96 ± 85,54	3613,42 ± 69,93
300x300	7681,78 ± 223,25	7425,37 ± 172,67	7343,73 ± 170,02	7566,93 ± 273,25	8182,27 ± 229,40
400x400	20372,6 ± 531,57	19815,8 ± 462,07	19990,5 ± 441,91	20203,7 ± 323,53	22514,7 ± 354,07
500x500	25047,5 ± 361,12	23896 ± 381,67	24053,2 ± 687,54	23779,2 ± 272,50	27664,1 ± 439,83
600x600	37033,1 ± 358,23	35102,5 ± 561,73	33464,3 ± 345,98	34589,4 ± 451,47	41233,7 ± 303,82
700x700	46061,4 ± 396,92	42208,2 ± 312,84	41257,9 ± 400,89	41256,2 ± 314,29	51273 ± 256,76
800x800	62771,4 ± 947,11	56545,2 ± 363,72	53394,1 ± 504,35	53440,1 ± 242,77	69746 ± 613,87

Tabella 5.1: Tempi (in ms) su Intel Core i5-8300H CPU @ 2.30Ghz.

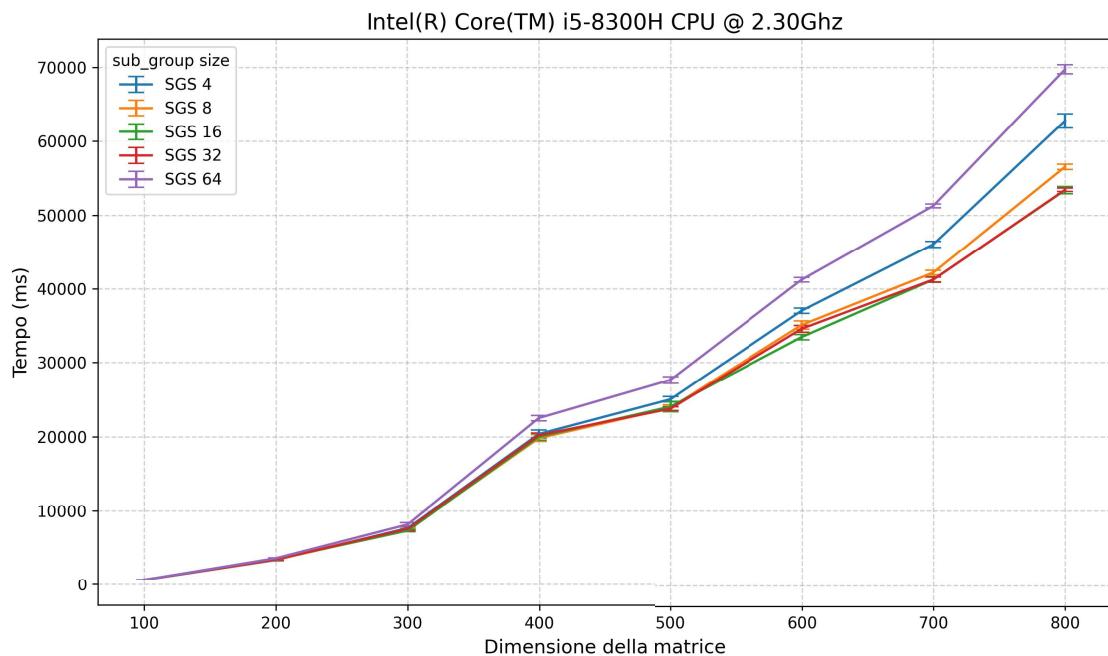


Figura 5.1: Le subgroup size che offrono le migliori prestazioni sono 16 e 32.

Matrice	SGS 4	SGS 8	SGS 16	SGS 32	SGS 64
100x100	661,19 ± 22,13	666,48 ± 16,16	619,47 ± 26,25	616,33 ± 20,93	639,23 ± 21,72
200x200	3952,07 ± 79,54	2820,76 ± 61,73	3026,32 ± 69,49	2910,23 ± 91,22	3140,40 ± 160,77
300x300	5884,67 ± 115,46	5937,32 ± 137,05	5970,84 ± 135,95	5386,92 ± 94,81	5801,86 ± 126,30
400x400	21680 ± 1327	15203 ± 1564	15258 ± 324	21575 ± 1084	21987 ± 375
500x500	17763 ± 299	18924 ± 595	23912 ± 1866	24236 ± 1454	24311 ± 1485
600x600	23862 ± 352	23147 ± 350	24630 ± 306	34797 ± 1450	33621 ± 1915
700x700	29353 ± 395	26069 ± 379	27177 ± 475	26634 ± 371	42604 ± 1141
800x800	35380 ± 476	32445 ± 382	33130 ± 433	32038 ± 386	33148 ± 403

Tabella 5.2: Tempi (in ms) su Intel Xeon Platinum 8468V @ 2.40GHz.

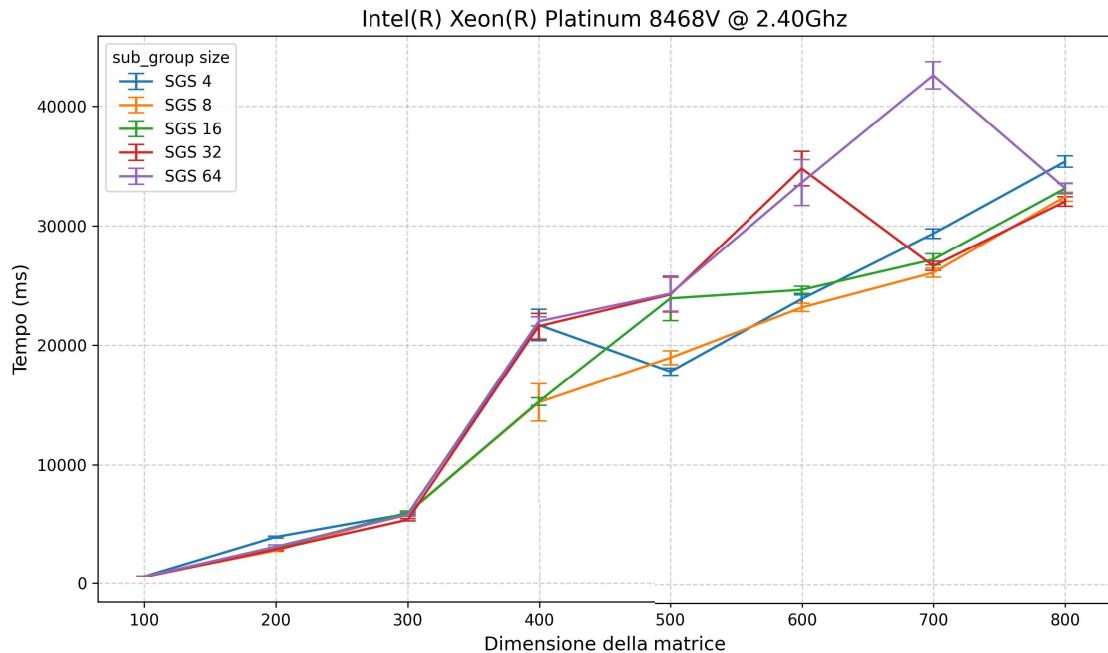


Figura 5.2: La subgroup size migliore è 8.

Matrice	SGS 32
100x100	292,08 ± 7,04
200x200	1438,2 ± 22,76
300x300	3019,95 ± 17,46
400x400	8270,31 ± 91,15
500x500	9479,56 ± 43,51
600x600	12352,7 ± 58,09
700x700	14310,8 ± 241,05
800x800	18777,2 ± 182,80

Tabella 5.3: Tempi (in ms) su NVIDIA GEFORCE GTX 1060.

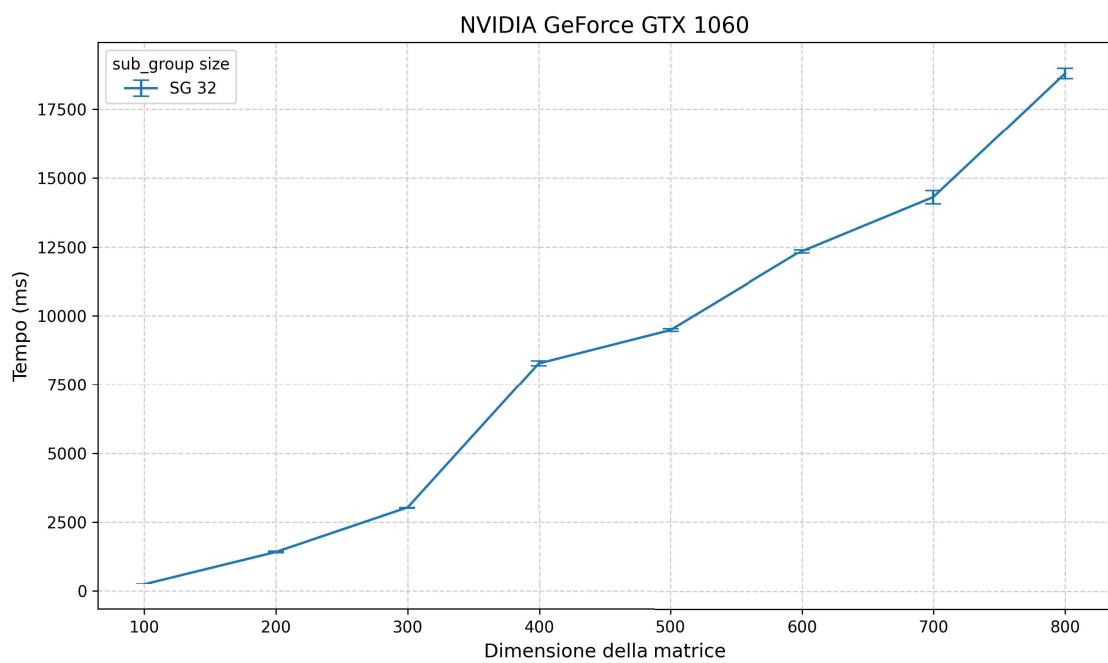


Figura 5.3: In questo caso non abbiamo possibilità di scelta di subgroup.

Matrice	SGS 16	SGS 32
100x100	227,49 ± 5,36	227,50 ± 6,00
200x200	1039,79 ± 10,21	970,27 ± 11,48
300x300	2049,28 ± 18,16	1887,49 ± 7,83
400x400	5262,13 ± 114,00	4979,83 ± 125,40
500x500	5368,69 ± 12,97	5140,06 ± 13,91
600x600	7744,14 ± 19,49	6909,41 ± 19,91
700x700	8389,84 ± 16,65	7875,79 ± 141,02
800x800	10708,3 ± 13,71	9286,69 ± 17,29

Tabella 5.4: Tempi (in ms) su Intel Datacenter GPU MAX 1100.

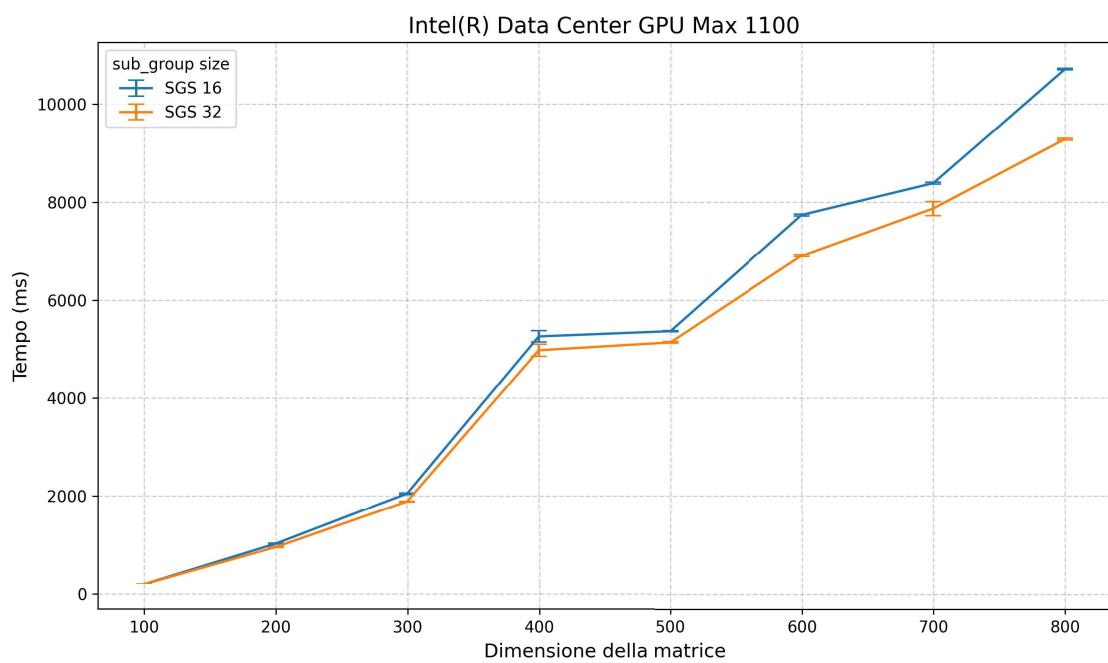


Figura 5.4: La migliore subgroup size è chiaramente 32.

5.2 Test di scaling

La scalabilità è in generale la proprietà di un sistema che descrive come risponde a un aumento del lavoro, o delle risorse. In particolare, nell'ambito software, prende il nome di efficienza di parallelizzazione, ovvero il rapporto tra speedup effettivo ed ideale, dato un certo numero di processori.

Lo speedup, a sua volta, si definisce come :

$$\text{Speedup} = \frac{t(1)}{t(N)}$$

dove $t(1)$ è il tempo di computazione su un singolo processore, e $t(N)$ è il tempo di computazione su N processori. Idealmente vorremmo lo speedup uguale al numero di processori ($\text{Speedup} = N$), ma nella pratica questo è un risultato estremamente difficile da ottenere.

5.2.1 Strong scaling

Lo strong scaling è un indice di quanto le prestazioni di un processo migliorino, dato un incremento di risorse computazionali. Idealmente, dato un maggior numero di nodi computazionali, il tempo di esecuzione dovrebbe decrescere linearmente. In verità, solo la parte parallelizzata del programma beneficia di un aumento di nodi computazionali; questo è il concetto dietro la Legge di Amdahl:

$$\text{Speedup} = \frac{1}{s + \frac{p}{N}}$$

dove s è la proporzione di tempo di esecuzione speso in codice non parallelizzato, p è il complementare, relativo alla parte parallelizzata del codice, e N è il numero di unità di calcolo. Dunque lo speedup non può tendere a infinito, ma converge ad un valore massimo dettato dalla parte seriale del programma.

Nel nostro caso il numero di unità computazionali in uso è direttamente correlato

con il numero di work_group lanciati contemporaneamente da un kernel. Nella nostra implementazione, la relazione tra dimensione di work_group e numero di work_group è data da:

$$\#work_group = \left\lfloor \frac{(N + work_group_size - 1)}{work_group_size} \right\rfloor$$

dove N è la dimensione del lato della matrice.

Dunque, per testare strong scaling, occorre selezionare la matrice e imporre una dimensione di work_group tale che se ne istanzi uno solo (e che tale dimensione sia multipla di quella dei sub_group). Poi aumentiamo il numero di work_group, seguendo le potenze del due, fino al massimo supportato dall'hardware - vedi Appendice B per le specifiche tecniche. Inoltre, la dimensione del work_group deve essere multipla di quella del subgroup (che scegliamo in base ai risultati presentati sopra), e non eccedere il tetto massimo consentito dall'hardware. Globalmente, il massimo numero di unità computazionali è 8, e la work_group size deve essere compresa tra 32 e 1024.

Di seguito forniamo i nostri test su ciascuno dei dispositivi, data la matrice 500 x 500, scelta come compromesso tra dimensione dei dati e tempo di esecuzione. Ogni dato è stato calcolato come la media di 50 esecuzioni.

		Sistema locale		Tiber AI	
WG size	# WG	NVIDIA GeForce GTX 1060	Intel Core i5-8300H	Datacenter GPU MAX 1100	Intel Xeon Platinum 8468V
512	1	8730 ± 11	24280 ± 566	5050 ± 17	17842 ± 210
256	2	8579 ± 9	23911 ± 516	5077 ± 15	17253 ± 289
128	4	8489 ± 11	24530 ± 468	5092 ± 15	17542 ± 219
64	8	7937 ± 12	25495 ± 308	5009 ± 17	18906 ± 253

Tabella 5.5: Tempi di esecuzione (in ms) all'aumentare dei work_group.

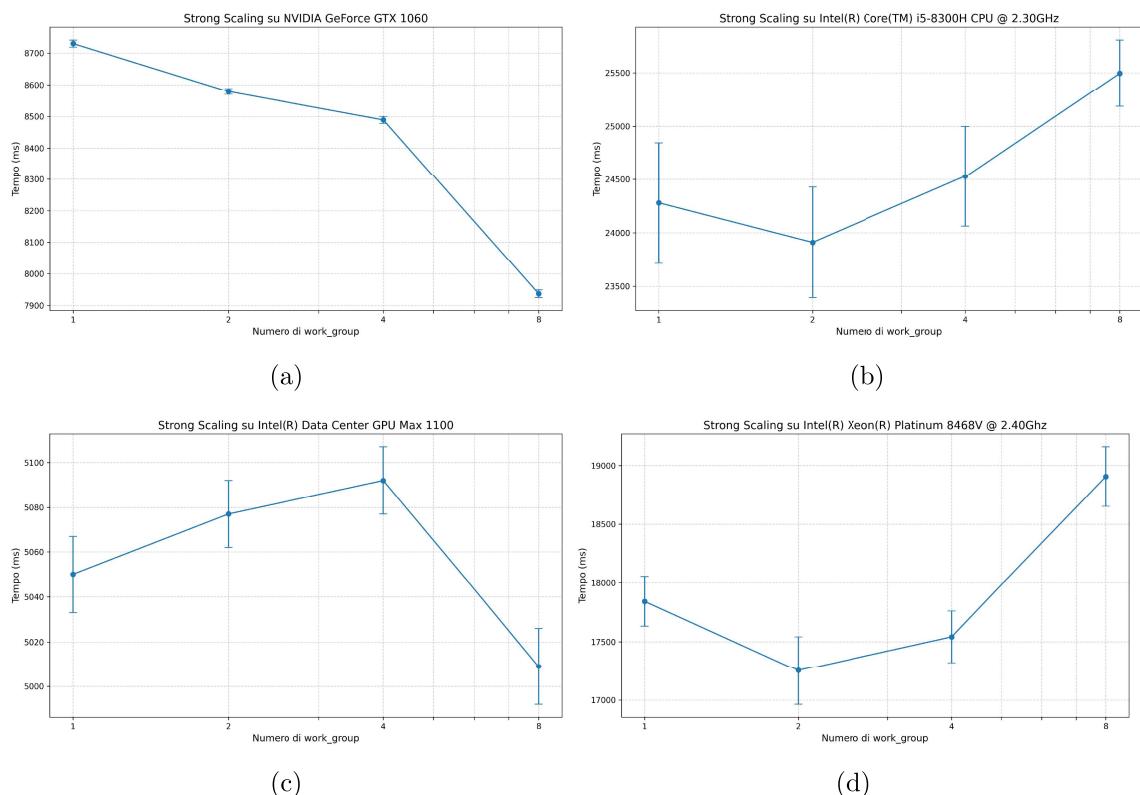


Figura 5.5: Strong scaling con grafici temporali

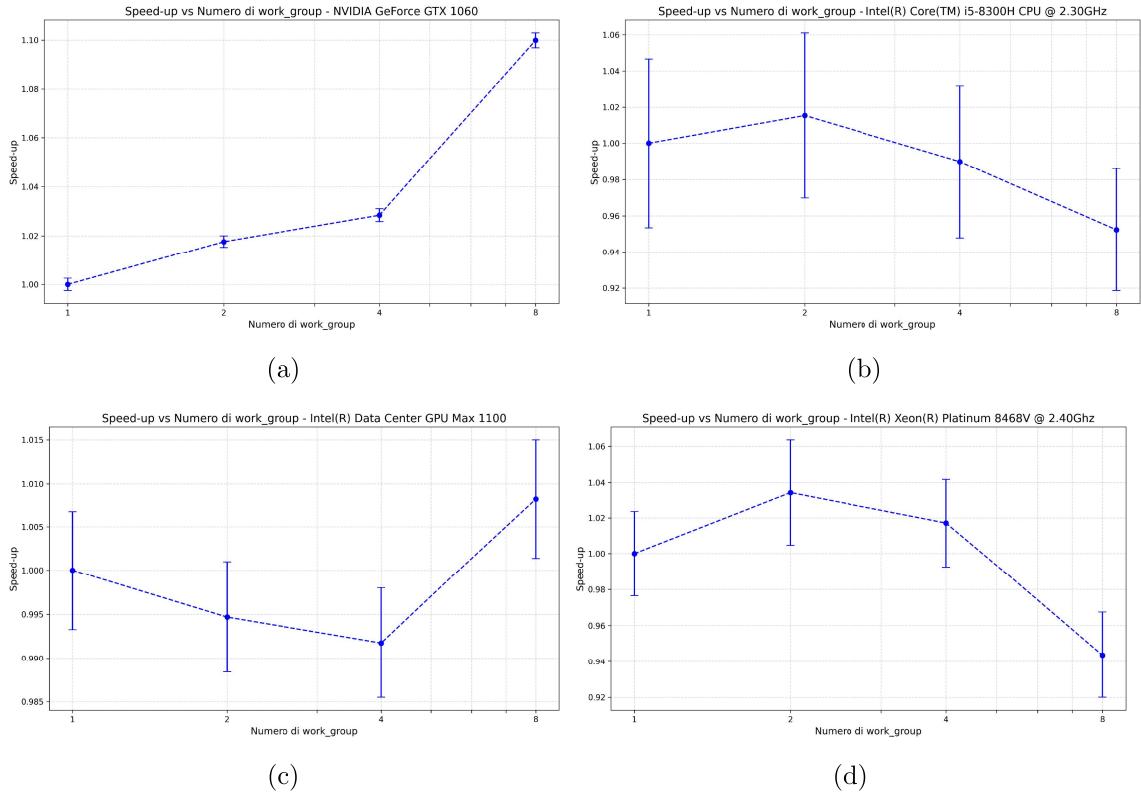


Figura 5.6: Speed-up di strong scaling

Come si può vedere, su GPU effettivamente otteniamo un leggero speedup all'aumento di risorse computazionali; su CPU invece le performance addirittura si logorano al ridursi della sub_group size. Questo risultato non è necessariamente inaspettato: su CPU potrebbero esserci inefficienze legate alla necessità di coordinare più core (sincronizzazione, condivisione di dati), cattiva ripartizione del workload per core (noi effettivamente possiamo solo specificarne il numero), accessi multipli alle stesse cache line, e problemi legati alla banda della memoria.

5.2.2 Weak scaling

Il weak scaling fa riferimento al miglioramento di prestazioni dovuto ad un incremento delle risorse disponibili assieme ad un equivalente incremento del carico di lavoro complessivo. Se il programma scalasse perfettamente secondo questa metrica, la ripartizione di lavoro in ogni nodo computazionale resterebbe la stessa, e quindi anche il tempo di esecuzione si manterebbe invariato.

La Legge di Gustafson ci dà una previsione dello speedup in questo scenario, sotto l'ipotesi che la parte parallela del programma scali linearmente con le risorse, e la parte seriale non scali rispetto alla dimensione del problema. Tradotto in formula:

$$\text{Speedup} = s + p * N$$

dove s è la percentuale di tempo spesa nell'esecuzione della parte seriale del programma, p è la percentuale complementare relativa alla parte parallela, e N è il fattore di aumento di risorse e carico di lavoro.

Secondo questa relazione, lo speedup incrementa linearmente rispetto al numero di nodi computazionali (con coefficiente angolare minore di 1), e non c'è un limite massimo.

Inoltre possiamo introdurre il concetto di efficienza di weak scaling:

$$\text{Efficienza} = \frac{t(1)}{t(N)}$$

dove $t(1)$ è il tempo di esecuzione relativo a un certo carico di lavoro, eseguito su un nodo computazionale, e $t(N)$ è il tempo di esecuzione relativo a N volte il carico di lavoro, eseguito su N nodi.

Da notare che la dimensione del problema non scala allo stesso modo della dimensione della matrice: essendo la complessità computazionale dell'Algoritmo Ungherese pari a $O(n^3)$, raddoppiare il carico di lavoro significa aumentare di $\sqrt[3]{2}$ il

lato della matrice.

Nella nostra implementazione scalare la dimensione della matrice assieme al numero di unità computazionali non è banale: non è scontato che esista una dimensione di work_group che consenta la relazione desiderata tra le due grandezze. Dobbiamo ricordare che esiste già una dipendenza tra dimensione della matrice, numero di work_group e dimensione dei work_group (quest'ultima a sua volta deve essere multipla della dimensione già fissata dei sub_group).

La migliore combinazione di queste grandezze è data dalla seguente serie, ricordando che il numero di work_group ($\#WG$) deve aumentare moltiplicando per 2, mentre la dimensione del lato della matrice (N) deve aumentare moltiplicando per $\sqrt[3]{2}$:

- $N_1 = 300, WGS_1 = 512 \Rightarrow \#WG = \lceil \frac{N}{WGS} \rceil = 1$
- $N_2 = 378, WGS_2 = 256 \Rightarrow \#WG = \lceil \frac{N}{WGS} \rceil = 2$
- $N_3 = 476, WGS_3 = 128 \Rightarrow \#WG = \lceil \frac{N}{WGS} \rceil = 4$
- $N_4 = 600, WGS_4 = 64 \Rightarrow \#WG = \lceil \frac{N}{WGS} \rceil = 10 \neq 8$

La serie finisce con N_3 perché allo step successivo non è possibile impostare una work_group size (WGS) tale che i work_group siano 8.

Dunque generiamo randomicamente nuove matrici per le dimensioni che non abbiamo, assicurandoci che contengano la stessa distribuzione incontrata nel dataset pre-esistente.

Di seguito riassumiamo i risultati, ottenuti come media di 50 esecuzioni.

Dimensione della matrice	# WG	Sistema locale		Tiber AI	
		NVIDIA GeForce GTX 1060	Intel Core i5-8300H	Datacenter GPU MAX 1100	Intel Xeon Platinum 8468V
300	1	2780 ± 23	7635 ± 804	1918 ± 7	5663 ± 184
378	2	4810 ± 55	14646 ± 1045	2982 ± 12	9702 ± 161
476	4	4914 ± 44	15312 ± 974	2763 ± 9	10499 ± 149

Tabella 5.6: Tempi di esecuzione (in ms) all'aumentare della dimensione della matrice e del numero di work_group.

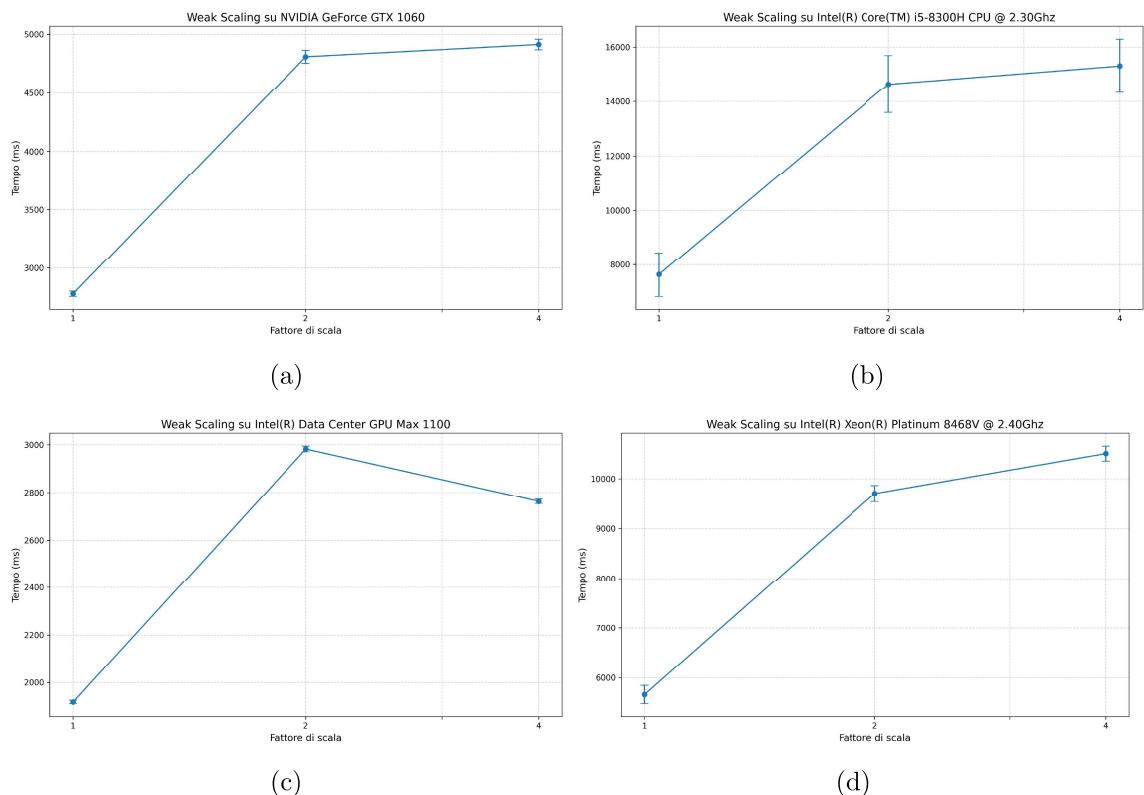


Figura 5.7: Weak scaling con grafici temporali

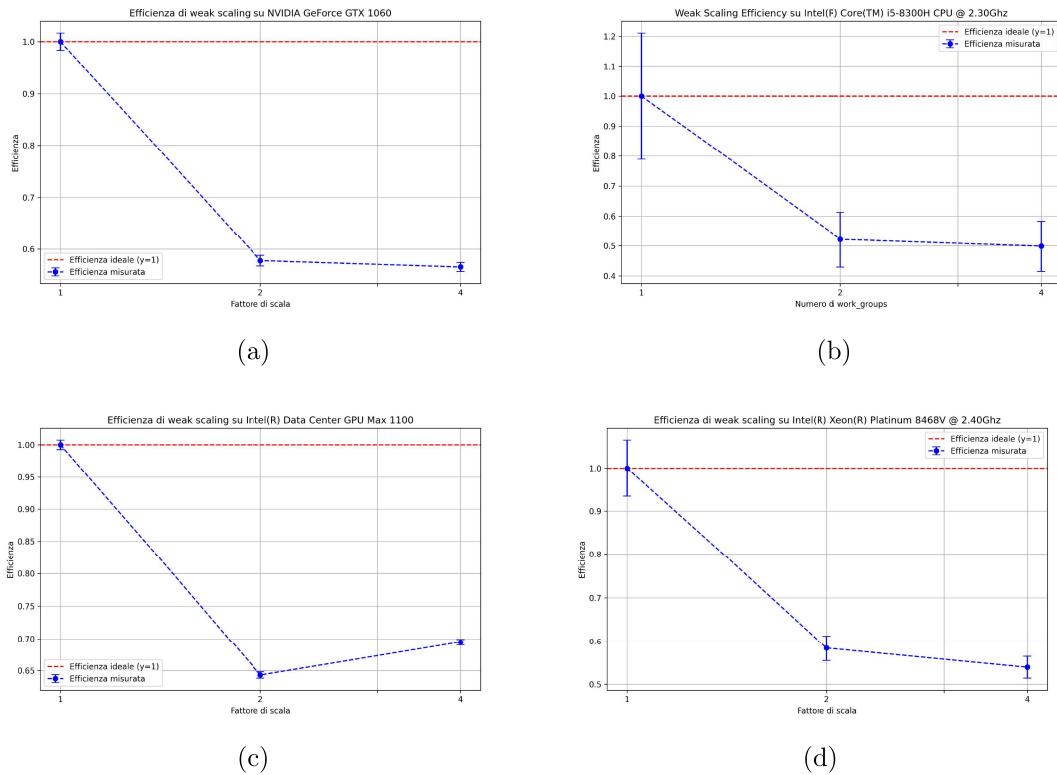


Figura 5.8: Efficienza di weak scaling

Come è possibile vedere, i risultati sono piuttosto erratici: questo perché nonostante l'incremento di dimensione della matrice, potrebbe essere richiesto un minor numero di cicli per trovare la soluzione dell'Algoritmo Ungherese, producendo tempi di esecuzione non intuitivi. Anche contando il numero di cicli non otteniamo un buon predittore del tempo di esecuzione perché ogni ciclo è più o meno lungo, a seconda dell'attivazione della logica di controllo, e da quante volte debba essere ripetuta un'operazione prima di raggiungere un risultato parziale. Riconosciamo che un approccio migliore al test di weak scaling sarebbe stato generare una nuova matrice casuale ad ogni esecuzione, e aumentarne significativamente il numero: così facendo otterremmo risultati più consistenti. Ricordando però che abbiamo un limite di tempo sulle risorse messe a disposizione da Tiber AI Cloud, abbiamo deciso di non procedere in questa direzione.

Capitolo 6

Risorse aggiuntive

Di seguito presentiamo una carrellata di argomenti che abbiamo esplorato nel corso della redazione di questo progetto, ma che non meritano un capitolo a sé stante a causa di difficoltà incontrate nell'utilizzo, o scarsa rilevanza complessiva.

6.1 Intel Advisor

Intel Advisor è uno strumento di analisi e design contenuto nel Base Toolkit di OneAPI. Questo permette di analizzare il codice ed ottenere informazioni riguardo alla sua ottimizzazione, ad esempio tramite analisi di roofline su CPU e GPU, proiezione della performance dell'offload, identificazione di potenziali colli di bottiglia e raccomandazioni sulla parallelizzazione (specifiche per GPU) [22]. È possibile utilizzarlo sia tramite GUI, sia attraverso linea di comando.

L'abbiamo lanciato sui dispositivi di Tiber AI Cloud (via Jupyter con linea di comando) e ne abbiamo estratto il tempo passato in cicli vettorizzati rispetto al tempo totale. Sulla GPU locale non è possibile lanciarlo, dato che per poter compilare codice SYCL su GPU NVIDIA occorre usare un plugin per la compatibilità con CUDA, il quale non permette l'esecuzione dello strumento. Un'altra limitazione è sulla modellizzazione di roofline: Intel Advisor è automaticamente configurato per

l’analisi basata su FLOP (floating point operations), ma nel nostro caso il codice effettua solo operazioni intere (richiedendo quindi un’analisi basata su INTOP).

6.2 Intel VTune Profiler

Altro strumento offerto dal Base Toolkit di OneAPI, utile ad ottimizzare le performance di applicazioni SYCL, ma anche OpenCL o OpenMP, nella loro interezza: studia sia la parte parallelizzata che quella seriale. Serve anche a mettere in risalto i punti del codice più dispendiosi in tempo, a identificare colli di bottiglia relativi all’uso di memoria e a limitazioni hardware. [23].

In locale non è stato possibile testarlo per problemi relativi all’installazione di oneAPI (in particolare componenti mancanti). L’abbiamo potuto lanciare su Jupyter, ma ci asteniamo dal presentarne i risultati qui, preferendo consegnarli come allegati. Questo perché non possiamo fare affidamento sulla correttezza dell’output: rileva che il nostro programma alloca un totale di 72.5 TB di memoria, per poi deallocare solo 1.1 TB. Troviamo improbabile un memory leak di più di 70 TB, poiché la RAM del sistema locale è di 16 GB.

6.3 SYnergy

SYnergy è una API basata su SYCL, sviluppata dal Prof. Biagio Cosenza dell’Università di Salerno, in collaborazione con il CINECA Supercomputing Center e del ”SYCL Working Group in Khronos” [24]. Il suo obiettivo è la profilazione del consumo energetico di programmi eterogenei parallelizzati.

Per quanto riguarda il sistema locale, è garantita la compatibilità con GPU NVIDIA tramite NVML; invece per GPU Intel - com’è il caso su Tiber AI Cloud - usa LevelZero; può funzionare anche su schede grafiche AMD, attraverso ROCm SMI.

Nonostante ciò, l'utilizzo del programma resta infattibile su Jupiter, a causa di errori di compilazione (static assertion failed: no kernel name provided without -fsycl-unnamed-lambda). Aggiungere l'argomento manualmente resta infruttuoso perché viene ignorato, finendo nello stesso errore.

Anche in locale si incontrano problemi: riusciamo ad effettuare il build, ma il runtime fallisce perché non supporta il wrapper SYnergy NVML. Noi utilizziamo l'API LLVM per poter compilare codice SYCL, probabilmente questa è l'origine dell'errore.

6.4 CUDA & SYCL

Aggiungiamo anche delle risorse di confronto tra i modelli di programmazione CUDA e SYCL. Per quanto non rilevanti al nostro progetto, le abbiamo trovate di sufficiente interesse per menzionarle [20]. È altresì importante menzionare che portare un programma CUDA in SYCL è relativamente semplice, avendo cura di seguire le linee guida sulla migrazione [21].

6.5 FPGA

All'inizio della stesura del progetto, uno dei nostri obiettivi era testare la compatibilità e performance di SYCL su FPGA. Questo era reso possibile dall'emulatore offerto dal servizio Intel DevCloud. Come già accennato, questa possibilità è sfumata con la migrazione del servizio verso Tiber AI Cloud e la chiusura definitiva di DevCloud.

Capitolo 7

Conclusioni

Con questo progetto sono state messe in luce diverse funzionalità di SYCL, e anche le insidie che si presentano ad un programmatore non esperto delle caratteristiche del linguaggio. Ad esempio, non è necessariamente chiaro quale sia il modo più corretto per approcciarsi all'utilizzo di memoria (buffer rispetto a USM), o quali siano i paradigmi di parallelizzazione più corretti (ad esempio che occorre inizializzare un puntatore con un'allocazione di memoria come *memset* o *memcpy*, piuttosto che inizializzare un kernel che compia la stessa operazione).

Un altro punto di interesse è la filosofia di programmazione: SYCL si propone come un linguaggio di alto livello, e quindi non lascia scelte dirette su alcuni aspetti di gestione dell'hardware, preferendo astrazioni a volte lontane dagli effettivi meccanismi di esecuzione. Questo è indubbiamente dovuto alla necessità di calzare acceleratori con architetture molto diverse tra loro. L'altra faccia della medaglia però è appunto l'assenza di specifiche funzionalità, come barriere globali che possano sincronizzare work_group tra loro all'interno di uno stesso kernel, e la possibilità di lanciare funzioni di riduzione tra specifici elementi di un gruppo o sottogruppo (invece di essere costretti a lanciarle su tutti gli elementi indiscriminatamente).

La conseguenza negativa di questo approccio di alto livello si ripercuote anche sui test di weak e strong scaling, che non sono particolarmente incisivi. A tal riguardo, riconosciamo che una concausa potrebbe stare nella nostra implementazione, magari

intrinsecamente poco scalabile. Ma potrebbe altresì essere una questione legata al modo con cui il codice viene effettivamente interpretato dall'API di destinazione.

Altra pratica fastidiosa è che il miglior metodo per trovare la combinazione di parametri ottimali per l'esecuzione sia quello di lanciare un grid-search con tutti i dimensionamenti possibili, dato che non è possibile sapere a priori gli effetti che un cambiamento parametrico avrà sulle performance.

OneAPI si propone di ovviare a quest'ultimo problema grazie agli strumenti di profiling (Intel VTune e Advisor), che per noi però sono stati in larga parte inaccessibili. Troviamo che ciò sia particolarmente grave quando lanciati sulla piattaforma offerta da Intel, dato che quell'ambiente dato dovrebbe essere uno strumento di onboarding. Anzi, l'assenza di consistenza delle risorse computazionali offerte, e la mancanza completa di assistenza nei confronti degli utenti non paganti, ha rallentato notevolmente i nostri sforzi.

In ogni caso, l'obiettivo principale del progetto è stato di successo: ci proponevamo di sfruttare l'Algoritmo Ungherese come banco di prova per l'eterogeneità di SYCL, e quindi confrontare le prestazioni di codice parallelizzato attraverso uno spettro di dispositivi. Il grafico sottostante riporta tale confronto, di cui vogliamo mettere in luce il punto di incrocio tra il tempo di esecuzione seriale rispetto a quello vettorizzato, entrambi sulla stessa piattaforma. Tale punto denota infatti la dimensione della matrice oltre la quale la parallelizzazione risulta più efficiente del codice C++ originale.

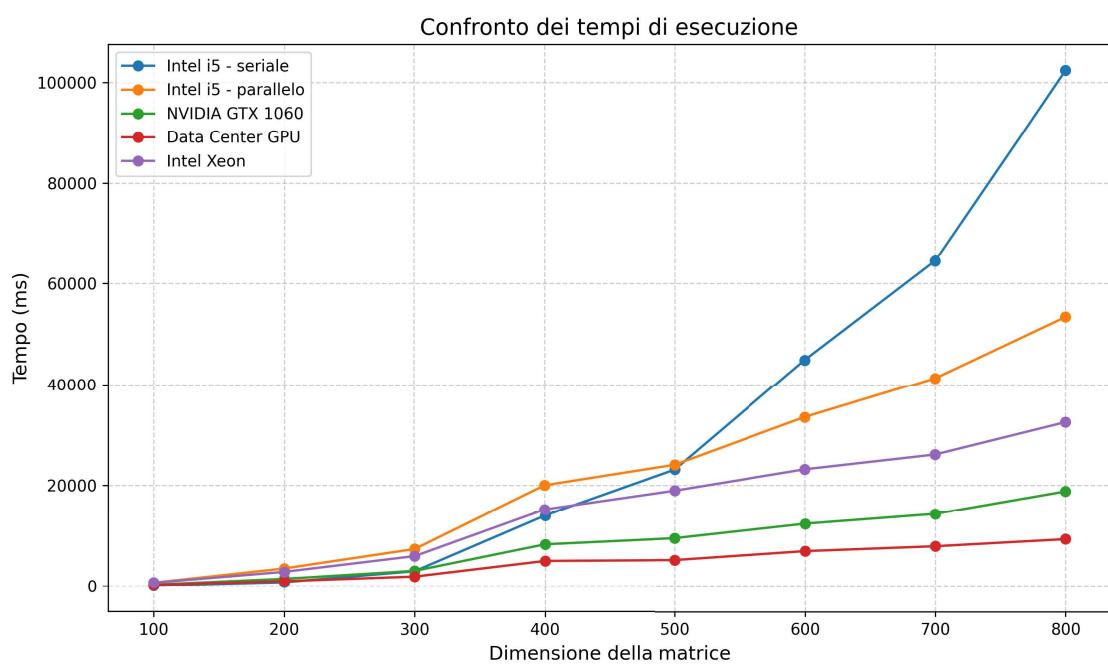


Figura 7.1: Tempi di esecuzione.

Appendice A

Specifiche del sistema locale

- *OS*: Ubuntu 24.04.1 LTS
- *Kernel OS*: Linux 6.8.0-51-generic
- *Architettura OS*: x86-64
- *RAM*: 16GB
- *CPU*: Intel(R) Core(TM) i5-8300H CPU @ 2.30 GHz
 - *Microarchitettura*: Coffee Lake
 - *Core*: 4
 - *Unità computazionali*: 8
 - *Dimensione globale massima*: 8192x8192x8192
 - *Dimensione massima del work-group*: 8192
 - *Dimensione del work-group preferita (multipli di)*: 128
 - *Dimensioni di sub-group supportate*: 4, 8, 16, 32, 64
 - *Massimo numero di sub-group per work-group*: 2048
- *GPU*: NVIDIA GeForce GTX 1060
 - *Device RAM*: 6GB

- *Architettura:* Pascal
 - *Compute capability:* 6.1
 - *CUDA Version:* 12.6
 - *CUDA core:* 1280
 - *Unità computazionali:* 10
 - *Dimensione globale massima:* 1024x1024x1024
 - *Dimensione massima del work-group:* 1024
 - *Dimensione del work-group preferita (multipli di):* 32
 - *Dimensioni del warp [20]:* 32
 - *Massimo numero di sub-group per work-group:* 0
- *Compilatore:* Intel(R) oneAPI DPC++/C++ Compiler 2024.2.1 (2024.2.1.2024711)

Appendice B

Specifiche del sistema hostato da Intel Tiber AI Cloud

- *OS*: Ubuntu 22.04.2 LTS
- *Kernel*: Linux 5.15.0-73-generic
- *Architettura*: x86-64
- *CPU*: Intel(R) Xeon(TM) Platinum 8468V @ 2.30 GHz
 - *Microarchitettura*: Golden Cove
 - *Core*: 48
 - *Unità computazionali*: 10
 - *Dimensione globale massima*: 8192x8192x8192
 - *Dimensione massima del work-group*: 8192
 - *Dimensione del work-group preferita (multipli di)*: 128
 - *Dimensioni di sub-group supportate*: 4, 8, 16, 32, 64
 - *Massimo numero di sub-group per work-group*: 2048
- *GPU*: Intel(R) Data Center GPU Max 1100

- *Device RAM*: 48GB
 - *X^e Core*: 56
 - *X^e Vector Engines*: 448
 - *Dimensione globale massima*: 1024x1024x1024
 - *Dimensione massima del work-group*: 1024
 - *Dimensione del work-group preferita (multipli di)*: 32
 - *Dimensioni di sub-group supportate*: 16, 32
 - *Massimo numero di sub-group per work-group*: 64
- *compilatore*: Intel(R) oneAPI DPC++/C++ Compiler 2024.2.1 (2024.2.1.20240711)

Bibliografia

- [1] Intel, *Intel® Tiber™ AI Cloud*, intel.com, <https://www.intel.com/content/www/us/en/developer/tools/tiber/ai-cloud.html>
- [2] Akshitha, S., Sowmya, R., Ananda Kumar, K. S., Suman Pawar, R., NeethrithaMeda, M. (2018), *Implementation of Hungarian Algorithm to obtain Optimal Solution for Travelling Salesman Problem.*, IEEE.
- [3] Kuhn, H. W., Marzo 1955, *The Hungarian Method for the Assignment Problem*
- [4] Edmonds, J., Karp, R. M. (1972). *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*. In Journal of the Association for Computing Machinery (Vol. 19, Issue 2).
- [5] Beasley, J. E. (1990). *Linear Programming on Cray Supercomputers*. The Journal of the Operational Research Society (Vol. 41, Issue 2).
- [6] Jonker, R., Volgenant, A. (1987). *Computing A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems*. Computing (Vol. 38).
- [7] J E Beasley, *OR-Library*, people.brunel.ac.uk, <https://people.brunel.ac.uk/~mastjjb/jeb/orlib/assigninfo.html>
- [8] Ruyman Reyes, *Bringing Nvidia® and AMD support to oneAPI*, codeplay.com, 16 Dicembre 2022, <https://codeplay.com/portal/blogs/2022/12/16/bringing-nvidia-and-amd-support-to-oneapi.html>
- [9] Fernando B. Giannasi, *hungarian_algorithm*, github.com, Febbraio 2018, https://github.com/phoemur/hungarian_algorithm?tab=readme-ov-file
- [10] *The Hungarian algorithm*, www.hungarianalgorithm.com, 2013, <https://www.hungarianalgorithm.com/index.php>
- [11] Intel, *C++ SYCL Introduction*, Jupyter Notebook, <https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/sycl-essentials.html>

- [12] Intel, *What is oneAPI?*, intel.com, 20 Giugno 2024, <https://www.intel.com/content/www/us/en/developer/articles/technical/oneapi-what-is-it.html>
- [13] Intel, *Execution Model Overview*, intel.com, 19 Novembre 2024, <https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/execution-model-overview.html>
- [14] Intel, *Unified Shared Memory Allocations*, intel.com, 19 Novembre 2024, <https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/unified-shared-memory-allocations.html>
- [15] Intel, *Performance Impact of USM and Buffers*, intel.com, 19 Novembre 2024, <https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/performance-impact-of-usm-and-buffers.html>
- [16] Intel, *Avoiding Declaring Buffers in a Loop*, intel.com, 19 Novembre 2024, <https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/avoiding-declaring-buffers-in-a-loop.html>
- [17] Intel, *Thread Mapping and GPU Occupancy*, intel.com, 19 Novembre 2024, <https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/thread-mapping-and-gpu-occupancy.html>
- [18] Intel, *Removing Conditional Checks*, intel.com, 19 Novembre 2024, <https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/removing-conditional-checks.html>
- [19] oneAPI Github, *Intel® GPU Occupancy Calculator*, oneapi-src.github.io, 31 Maggio 2024, <https://oneapi-src.github.io/oneAPI-samples/Tools/GPU-Occupancy-Calculator/>
- [20] oneAPI Github, *CUDA* and SYCL* Programming Model Comparison*, oneapi-src.github.io, 30 Ottobre 2024, https://oneapi-src.github.io/SYCLomatic/dev_guide/reference/compare-prog-models.html
- [21] oneAPI Github, *CUDA API Migration Support*, oneapi-src.github.io, 30 Ottobre 2024, https://oneapi-src.github.io/SYCLomatic/dev_guide/api-mapping-status.html
- [22] Intel, *Intel® Advisor*, intel.com, <https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html#gs.j2a26s>
- [23] Intel, *Intel® VTune™ Profiler*, intel.com, <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.j2dyw7>

- [24] Khronos Group, *SYCL goes Green with SYnergy*, Biagio Cosenza, khronos.org, 11 Dicembre 2023, <https://www.khronos.org/blog/sycl-goes-green-with-synergy>