Analisi Prestazionale del Radix Sort Parallelo: Confronto tra Architetture Multi-core CPU e Many-core GPU in Ambienti CUDA

Leonardo Bacciocchi

Sommario

Il presente lavoro si pone l'obiettivo di implementare e analizzare in modo comparativo l'algoritmo Radix Sort in contesti di calcolo parallelo. Sfruttando l'infrastruttura del Cluster UNIPR e le potenzialità di architetture locali eterogenee — nello specifico una GPU NVIDIA GeForce RTX 4060 e una CPU AMD Ryzen 7 — lo studio esamina diverse strategie di parallelizzazione. L'analisi si focalizza sull'efficienza computazionale e sulla scalabilità delle diverse implementazioni, mettendo a confronto i paradigmi multi-core e many-core per determinare l'impatto delle specifiche architetture sulle prestazioni dell'ordinamento.

Indice

1 Introduzione

Il Radix Sort è un algoritmo di ordinamento non comparativo per valori numerici interi o stringhe con una solida base storica[1]. Le sue origini risalgono alla fine del XIX secolo con il lavoro di Herman Hollerith sulle macchine tabulatrici a schede perforate. Queste tecnologie furono fondamentali per il censimento degli Stati Uniti del 1890, dimostrando un'efficienza senza precedenti nella gestione di grandi volumi di dati. Il processo meccanico originario seguiva una logica iterativa: le schede venivano smistate in scomparti in base alle cifre, partendo dalla meno significativa fino alla più significativa.

L'algoritmo è stato formalizzato per l'era dei calcolatori elettronici nel 1954 da Harold H. Seward presso il MIT [3]. A differenza degli algoritmi basati sui confronti, come Quicksort o Mergesort, il Radix Sort sfrutta la natura digitale delle chiavi per raggiungere una complessità temporale di $O(nk)$, dove n è il numero di elementi e k la lunghezza delle chiavi (o il numero di passaggi). Questa caratteristica lo rende potenzialmente più veloce dei classici algoritmi $O(n \log n)$, specialmente quando l'intervallo dei valori è limitato.

La metodologia di funzionamento si divide principalmente in due approcci:

- LSD (Least Significant Digit): l'ordinamento avviene partendo dalla cifra meno significativa. È un approccio iterativo che richiede un algoritmo di supporto stabile (come il Counting Sort) per preservare l'ordine relativo acquisito nei passaggi precedenti.

- MSD (Most Significant Digit): l'ordinamento procede dalla cifra più significativa. Questo metodo utilizza un approccio "divide et impera" che partiziona l'input in bucket indipendenti, facilitando la scomposizione del problema in sottoproblemi parallelizzabili.

Nonostante i vantaggi teorici, l'implementazione su architetture moderne deve affrontare sfide legate all'efficienza della memoria e alla parallelizzazione. La natura intrinsecamente sequenziale di alcune fasi, come la permutazione degli elementi nelle versioni in-place, e le difficoltà nel bilanciamento del carico in presenza di distribuzioni di dati asimmetriche (skewed), rappresentano i principali colli di bottiglia per le prestazioni.

L'obiettivo di questo lavoro è l'implementazione e lo studio comparativo del Radix Sort in versione parallela. Sfruttando la potenza computazionale del Cluster UNIPR e le risorse locali offerte da una GPU NVIDIA GeForce RTX 4060 e una CPU AMD Ryzen 7, verranno analizzate diverse tecniche di ottimizzazione. La relazione si focalizzerà sul confronto tra implementazioni multi-thread su CPU e accelerazione many-core tramite CUDA, valutando sistematicamente tempi di esecuzione, scalabilità e stabilità dell'ordinamento su dataset di diverse dimensioni.

2 Stato dell'Arte e Ricerca Bibliografica

2.1 Analisi della Complessità

La complessità del Radix Sort si differenzia dagli algoritmi basati sul confronto poiché non dipende dal numero di comparazioni tra elementi, ma dalla struttura intrinseca delle chiavi. Definendo n come il numero di elementi da ordinare, d come il numero di cifre del valore massimo e k come l'intervallo di valori che ogni cifra può assumere (ovvero la base), l'analisi si articola come segue:

- Complessità Temporale: L'algoritmo presenta una complessità di $O(d \cdot (n + k))$. Poiché il processo deve esaminare ogni cifra di ogni elemento per tutte le d posizioni, il tempo di esecuzione rimane costante nei casi migliore, medio e peggiore [2]. Se d è considerato una costante o è proporzionale a $\log_k(max\_val)$, il Radix Sort può raggiungere prestazioni quasi lineari, superando algoritmi come Quicksort o Mergesort in scenari dove il range delle chiavi non è eccessivamente vasto rispetto a n.

- Complessità Spaziale: L'algoritmo non opera in-place nella sua forma standard (LSD), richiedendo uno spazio ausiliario pari a $O(n+k)$. Questa memoria aggiuntiva è necessaria per la gestione dei bucket (o dei conteggi nel Counting Sort sottostante) e per l'array di output temporaneo che memorizza gli elementi ordinati dopo ogni passata sulla singola cifra [2].

L'efficienza pratica dell'algoritmo è dunque strettamente legata alla scelta della base k: una base più grande riduce il numero di passaggi d, ma aumenta proporzionalmente lo spazio occupato dai contatori e l'overhead di gestione della memoria.

2.2 Metodologie di Ordinamento: LSD, MSD e Approcci Ibridi

Il Radix Sort può essere implementato seguendo diverse strategie che differiscono per la direzione di scansione delle cifre e per le modalità di gestione della memoria e del parallelismo.

2.2.1 LSD (Least Significant Digit)

L'approccio Least Significant Digit elabora le chiavi partendo dalla cifra (o dal blocco di bit) meno significativa e procedendo verso quella più significativa.

- Stabilità: Per garantire la correttezza, ogni passata deve essere necessariamente stabile, ovvero deve preservare l'ordine relativo degli elementi con la stessa cifra acquisito nei passaggi precedenti.

- Implementazione: È tipicamente iterativo e richiede un algoritmo di supporto (spesso il Counting Sort) che operi in tempo lineare.

- Limitazioni: Sebbene efficace su hardware seriale, la parallelizzazione globale è ostacolata dalle dipendenze tra le passate successive sull'intero array.

2.2.2 MSD (Most Significant Digit)

La metodologia Most Significant Digit inizia l'ordinamento dalla cifra più significativa, suddividendo l'array in bucket indipendenti.

- Divide et Impera: Una volta partizionato il dataset iniziale in base ai bit più significativi, ogni bucket può essere ordinato in modo ricorsivo e totalmente indipendente dagli altri.

- Parallelismo: Questa indipendenza rende il paradigma MSD ideale per architetture multi-core (come il Cluster UNIPR), consentendo a thread diversi di elaborare partizioni distinte simultaneamente.

- Varianti In-Place: Algoritmi avanzati come American Flag Sort o Regions Sort utilizzano la logica MSD per minimizzare l'uso di memoria ausiliaria.

2.2.3 Approccio Ibrido (MSD + LSD)

L'approccio ibrido combina i vantaggi di entrambi i paradigmi MSD e LSD per ottimizzare le prestazioni su diverse distribuzioni di dati e dimensioni di bucket.

- Strategia Adattiva: L'algoritmo inizia con una fase MSD per partizionare rapidamente il dataset in bucket indipendenti, poi applica tecniche diverse in base alla dimensione di ciascun bucket.
- Fase 1 - Partizionamento MSD: I bit più significativi vengono utilizzati per suddividere l'array in bucket disgiunti. Questa fase sfrutta il parallelismo intrinseco del paradigma MSD, permettendo l'elaborazione concorrente delle partizioni.
- Fase 2 - Ordinamento Locale: Ogni bucket viene ordinato con la tecnica più efficiente in base alla sua dimensione:
  - Bucket grandi: Si applica LSD Radix Sort, che garantisce stabilità e prestazioni ottimali su sequenze lunghe grazie all'accesso sequenziale alla memoria.
  - Bucket medi: Si continua ricorsivamente con MSD oppure si passa a LSD, a seconda della profondità raggiunta.
  - Bucket piccoli: Si utilizzano algoritmi comparison-based come Insertion Sort, più efficienti per sequenze brevi grazie al minor overhead.
- Vantaggi: Questa strategia riduce la profondità ricorsiva rispetto a MSD puro, evita le dipendenze globali di LSD puro, e adatta dinamicamente la tecnica di ordinamento alle caratteristiche locali dei dati.

2.3 Strutture operative

La struttura di un algoritmo di ordinamento si distingue principalmente in base all'utilizzo della memoria durante l'esecuzione. Le due categorie fondamentali sono il modello out-of-place, che richiede memoria ausiliaria proporzionale alla dimensione dell'input, e il modello in-place, che opera direttamente sull'array originale con memoria aggiuntiva costante.

2.3.1 Out-of-Place

L'approccio out-of-place utilizza un buffer ausiliario di dimensione $O(n)$ dove vengono scritti gli elementi durante l'ordinamento. L'array di input viene letto e gli elementi vengono copiati nelle posizioni corrette del buffer di output.

- Richiede memoria aggiuntiva pari a $O(n)$
- Implementazione generalmente più semplice e diretta
- Accessi in scrittura non conflittuali tra thread diversi

2.3.2 In-Place

L'approccio in-place ordina gli elementi direttamente nell'array originale, utilizzando solo $O(1)$ memoria ausiliaria.

- Consumo di memoria minimo
- Migliore località di cache (nessun buffer esterno)
- Implementazione più complessa (gestione di swap e cicli di permutazione)

2.4 Paradigma Computazionale: Confronto tra CPU e GPU

La scelta dell'algoritmo di ordinamento e della sua efficienza è intrinsecamente legata all'architettura hardware sottostante. Per comprendere l'evoluzione verso sistemi ibridi, è necessario distinguere i due paradigmi principali:

- Central Processing Unit (CPU): Le CPU moderne sono ottimizzate per la bassa latenza e l'esecuzione di logiche di controllo complesse. Grazie a gerarchie di cache profonde e sofisticati sistemi di branch prediction, eccellono nell'implementazione di versioni MSD ricorsive. In questo contesto, il parallelismo viene gestito tramite modelli a grana grossa (come OpenMP), dove pochi thread potenti gestiscono partizioni indipendenti di dati.
- Graphics Processing Unit (GPU): Le GPU sono architetture many-core progettate per il throughput massivo, seguendo il modello SIMT (Single Instruction, Multiple Threads). Sebbene offrano un bandwidth di memoria estremamente superiore, soffrono di un elevato costo di sincronizzazione tra i thread e di latenze significative nel lancio dei kernel. Per queste ragioni, le implementazioni GPU (come CUB o Thrust) prediligono spesso l'approccio LSD, che si presta meglio a primitive parallele come il prefix sum.

Questa dicotomia crea un punto di transizione: per dataset di piccole dimensioni o fasi di partizionamento iniziale, l'overhead della GPU supera i benefici del calcolo parallelo, rendendo la CPU più efficiente. Al contrario, su volumi di dati massivi, la GPU diventa imbattibile. Tale osservazione potrebbe anche costituire la base teorica per lo sviluppo di strategie ibride con un approccio misto tra le diverse metodologie.

2.5 Analisi della Letteratura e Ricerca Bibliografica

La letteratura scientifica sul Radix Sort riflette la transizione dai sistemi sequenziali a quelli massivamente paralleli, affrontando sfide quali la latenza della memoria, la sincronizzazione tra thread e l'ottimizzazione del bandwidth. Questa sezione analizza i contributi tecnici e gli algoritmi specifici che hanno definito lo stato dell'arte su diverse architetture.

2.5.1 Evoluzione su CPU

La formalizzazione informatica dell'algoritmo è dovuta a Harold H. Seward dell' MIT [3], che descrisse il key-indexed counting. Tuttavia, la ricerca si è evoluta per gestire la complessità delle chiavi moderne e la gerarchia delle cache:

- 3-way Radix Quicksort: Sviluppato presso la Princeton University [4], questo algoritmo ibrida la logica del Quicksort con quella digitale. Invece di partizionare rigidamente in R bucket (base del radix), utilizza un partizionamento a tre vie ($<, =, >$) basato sul carattere o blocco di bit corrente. Questa tecnica è estremamente efficiente per gestire chiavi con lunghi prefissi comuni o stringhe di lunghezza variabile, migliorando drasticamente la cache locality e riducendo il numero di accessi alla memoria principale.

2.5.2 Accelerazione su GPU: Ottimizzazioni Hardware e Bandwidth

Il passaggio a CUDA ha richiesto tecniche specifiche per saturare il bandwidth delle memorie video e minimizzare i conflitti di accesso:

- Shared Memory Histogramming (NVIDIA): Satish et al. [7] e lo sviluppo della libreria CUB hanno introdotto strategie per strutturare i buffer nella shared memory on-chip evitando i bank conflicts. Utilizzando istogrammi a livello di warp e block, i thread cooperano per contare i bit in parallelo, riducendo la contesa sui contatori globali e massimizzando l'efficienza dei multiprocessori della GPU.

- Fast 4-way Parallel Radix Sort: L'implementazione della University of Utah [8] propone una tecnica per elaborare 2 bit per radix contemporaneamente, creando 4 vie di smistamento. La metodologia introduce il coalesced block mapping, che garantisce che gli accessi alla memoria globale siano sempre allineati e raggruppati, raddoppiando le prestazioni rispetto ai precedenti approcci basati su API grafiche.

- 8-bit Hybrid Radix Sort (TU Munich): Stehle e Jacobsen [9] affrontano il limite del bandwidth-bound. Proponendo di processare 8 bit per passata (anziché 4 o 5), l'algoritmo dimezza il traffico sulla memoria globale. Il cuore tecnologico risiede nella thread reduction per gli istogrammi e nel look-ahead scattering, che permette di calcolare preventivamente le posizioni di scrittura, sollevando il sistema dal collo di bottiglia del bandwidth.

2.5.3 Innovazioni Algoritmiche e Varianti Specializzate

Recentemente sono nati algoritmi con nomi specifici progettati per risolvere problemi strutturali, come la necessità di memoria ausiliaria o l'inefficienza con dati duplicati:

- PARADIS (IBM/Stanford): Progettato per essere un algoritmo Parallel In-place [5], risolve l'esigenza di memoria ausiliaria $O(n)$ tramite la tecnica della speculative permutation. Poiché gli ordinamenti paralleli in-place soffrono tipicamente di dipendenze in scrittura, PARADIS permette ai thread di scrivere in modo "speculativo", risolvendo eventuali collisioni in una fase di repair parallela. L'efficienza di questa implementazione d'avanguardia [6] risiede nell'integrazione sistematica della vettorizzazione SIMD (Single Instruction, Multiple Data): sfruttando i registri hardware (da 128 a 512 bit), l'algoritmo parallelizza il conteggio degli istogrammi e lo shuffling dei dati all'interno del singolo core, garantendo un consumo di memoria costante senza sacrificare le prestazioni massime della CPU.

- DovetailSort: Affronta la criticità delle chiavi duplicate (heavy keys). Combina l'integer sort (radix) con il comparison sort tramite campionamento probabilistico

[10]. Questa strategia isola i valori frequenti, impedendo all'algoritmo di scendere in ricorsioni MSD profonde e inefficienti su dati con distribuzioni fortemente asimmetriche.

- RadiK (Alibaba Group): Variante specializzata per la selezione dei primi k elementi (Top-K) su GPU [11]. Invece di ordinare l'intero array, RadiK utilizza uno scaling adattivo dei bucket per scartare istantaneamente i rami di ricerca che non contengono i valori desiderati, ottenendo speedup fino a 4.8x rispetto ai metodi tradizionali.

- Fix Sort (Schmid et al.): Emerso dagli studi sul segmented sorting [12], rappresenta un'alternativa agile quando il range di valori è limitato. Ottimizza l'uso dei registri e della memoria condivisa, risultando superiore al Radix Sort classico quando la cardinalità dei segmenti di dati è ridotta.

2.5.4 Scalabilità e Applicazioni in Ambito Big Data

L'integrazione del Radix Sort in sistemi complessi ha portato a nuove metodologie di distribuzione del carico:

- Bucket Classification: Hongdi et al. [13] hanno introdotto una strategia di classificazione dei bucket per ambienti distribuiti. La metodologia bilancia dinamicamente il carico tra più schede video, riducendo i colli di bottiglia del bus PCIe o NVLink durante lo scambio di partizioni di dati tra diverse memorie GPU.

- Radix-Hashing: Mou et al. [14] descrivono l'uso del Radix Sort per accelerare operazioni di Group-By e aggregazione nei database moderni. Il sorting digitale viene impiegato per raggruppare chiavi ad alta cardinalità, offrendo prestazioni più prevedibili e consistenti rispetto alle tabelle hash tradizionali su GPU, che soffrono di collisioni elevate.

2.5.5 Parametrizzazione e Metodologia di Analisi

Infine, la letteratura fornisce gli strumenti metodologici per calibrare le prestazioni del sistema:

- Sub-vectors: La letteratura, analizza e sottolinea l'impatto della creazione dei sottovettori nel paradigma MSD [10], fornendo la giustificazione scientifica per l'utilizzo di soglie di cutoff. La ricerca dimostra come la CPU sia più efficiente nella gestione della logica dei bucket piccoli, mentre la GPU domini nei segmenti massivi.

- Benchmark Suites: L'utilizzo di framework come quelli di Kirill Lykov [15] permette la misurazione sistematica di parametri critici quali il branch misprediction e la pressione sulla cache L3, fondamentali per validare sperimentalmente i vantaggi di un'implementazione rispetto ad un'altra.

3 Architettura e Metodologia di Implementazione

3.1 Ambiente di Sviluppo e Testing

L'implementazione e la validazione degli algoritmi sono state condotte su due ambienti distinti: una workstation locale per lo sviluppo e il debugging, e il cluster HPC dell'Università di Parma per i benchmark su larga scala.

3.1.1 Workstation Locale

Componente	Specifica
CPU	AMD Ryzen 7 8845HS (8 core / 16 thread, 3.80 GHz)
GPU	NVIDIA GeForce RTX 4060 (8 GB GDDR6)
Compute Capability	8.9 (Ada Lovelace)
RAM	16 GB
Sistema Operativo	Windows 11 (64-bit) + WSL2 Ubuntu 24.04 LTS

Tabella 1: Specifiche workstation locale

3.1.2 Cluster HPC UNIPR

I test su larga scala sono stati eseguiti sul cluster HPC dell'Università di Parma, utilizzando nodi della partizione CPU per i test OpenMP e nodi GPU per i test CUDA.

Componente	Specifica
Architettura	Dual-socket
Core fisici	$\sim$32 (16 per socket)
Thread logici	64 (Hyperthreading)
RAM	32-64 GB DDR4
Interconnessione	NUMA (Non-Uniform Memory Access)

Tabella 2: Specifiche nodi CPU cluster HPC UNIPR

Componente	Specifica
GPU	NVIDIA A100 80GB PCIe
Compute Capability	8.0 (Ampere)
GPU per nodo	4
Memoria per GPU	80 GB HBM2
Bandwidth memoria GPU	1935 GB/s
Bandwidth inter-GPU misurato	$\sim$22 GB/s (PCIe)

Tabella 3: Specifiche nodi GPU cluster HPC UNIPR

3.1.3 Software

- Compilatore CPU: GCC 13.2.0
- Compilatore GPU: NVCC (CUDA Toolkit 12.4.1)

- Librerie GPU: Thrust, CUB (incluse in CUDA Toolkit)
- MPI: OpenMPI 2.1.2

3.1.4 Dataset

I dataset di test sono stati generati tramite un programma C++ che produce sequenze di interi casuali a 32 bit senza segno (`uint32_t`) con distribuzione uniforme nell'intervallo $[0, 2^{32} - 1]$. La scelta di chiavi a 32 bit è motivata da:

- Rappresentatività rispetto ad applicazioni reali (indici, hash, ID)
- Compatibilità con la dimensione nativa dei registri GPU
- Numero ottimale di passate per Radix Sort con digit a 8 bit ($32/8 = 4$)

Per garantire la riproducibilità degli esperimenti, il generatore è stato inizializzato con seed fisso.

Dataset	Elementi	Dimensione	Ambiente
10K	10.000	40 KB	Locale (debug)
1M	1.000.000	3.9 MB	Locale
10M	10.000.000	39 MB	Locale
100M	100.000.000	382 MB	Locale + Cluster
1B	1.000.000.000	3.8 GB	Cluster
2B	2.000.000.000	7.5 GB	Cluster
5B	5.000.000.000	19 GB	Cluster

Tabella 4: Dataset utilizzati per i benchmark

I dataset di dimensione superiore a 100M elementi sono stati utilizzati esclusivamente sul cluster HPC, data la limitata disponibilità di memoria RAM e VRAM sulla workstation locale.

3.2 Implementazioni CPU con OpenMP

Le implementazioni CPU utilizzano OpenMP per sfruttare il parallelismo multi-thread. Sono state sviluppate diverse varianti, combinando le metodologie LSD e MSD con le strutture in-place e out-of-place.

3.2.1 LSD Out-of-Place

L'implementazione LSD out-of-place rappresenta l'approccio più diretto e stabile. L'algoritmo esegue $k = \lceil 32/b \rceil$ passate iterative, dove b è il numero di bit per digit (configurabile da 1 a 16, default 8).

Calcolo dinamico delle passate: Il numero di passate viene ottimizzato in base al valore massimo presente nell'array, evitando iterazioni superflue su bit non significativi:

Listing 1: Ottimizzazione del numero di passate

```
// Trova il valore massimo nell'array
uint32_t maxv = a[0];
for(size_t i=1; i<n; i++)
    if(a[i] > maxv) maxv = a[i];

// Se tutti zero, array gia' ordinato
if(maxv == 0) return;

// Calcola l'indice del bit piu significativo
// __builtin_clz contare gli zeri iniziali di un intero. clz conta gli zeri iniziali.
int msb = 31 - __builtin_clz(maxv);

// Numero di passate: ceil((msb+1)/bits)
//msb rappresenta il numero totale di bit "utili" da ordinare. Se il bit piu alto sta in posizione 7, hai bisogno di analizzare 8 bit (da 0 a 7).

int passes = (msb + bits) / bits;
```

Strategia di parallelizzazione: L'algoritmo utilizza una strategia a tre fasi per ogni passata, con sincronizzazione tramite barrier implicite:

1. Istogramma parallelo: Ogni thread calcola un istogramma locale sul proprio chunk, evitando contese sui contatori globali
2. Riduzione e prefix-sum: Un singolo thread combina gli istogrammi e calcola gli offset per-thread per ogni bucket
3. Scatter parallelo: Ogni thread scrive i propri elementi nelle posizioni pre-calcolate, garantendo stabilità

Listing 2: Istogramma parallelo con contatori locali

```
#pragma omp parallel //ogni core eseguira la funzione
{
    // Identificativo del thread corrente (0, 1, 2, ...)
    const int tid = omp_get_thread_num();
    // Numero totale di thread attivi
    const int Tloc = omp_get_num_threads();

    // Ogni thread ha il proprio array di contatori
    // Offset nell'array globale: tid * base
    // divido array di memoria assegnando i bucket senza sovrapposizioni
    size_t *lc = thread_counts + (size_t)tid * base;

    // Inizializza contatori locali a zero
    for(int b=0; b<base; b++) lc[b] = 0;

```

```
16      // Partizionamento bilanciato: ogni thread processa n/Tloc
            elementi
17      // Usando divisione intera per evitare overlap
18      // Divido array di dati che devo ordinare in base ai thread a
            disposizione
19      size_t start = (n * (size_t)tid) / Tloc;
20      size_t end = (n * (size_t)(tid+1)) / Tloc;
21
22      // Ogni thread itera esclusivamente sugli elementi che gli sono
            stati assegnati
23      for(size_t i=start; i<end; i++){
24          // Estrai il digit corrente isolando una parte
25          // Shift sposta la cifra che ci interessa nelle posizioni
                meno significativa
26          // Mask applica una maschera binaria per isolare cifra
                corrente
27          unsigned d = (a[i] >> shift) & mask;
28          //incremento il suo valore nell'array privato
29          lc[d]++;
30      }
31
32      // Barrier implicita: tutti i thread devono completare prima di
            procedere alla riduzione
33      #pragma omp barrier
34
35
36  }
```

Gestione dei buffer (ping-pong): Ad ogni passata i ruoli di sorgente e destinazione vengono scambiati, evitando costose copie intermedie. Se il numero di passate è dispari, viene effettuata una copia finale per riportare il risultato nel buffer originale:

Listing 3: Tecnica ping-pong

```
1  // Alla fine di ogni passata: scambia i puntatori
2  // Costo: O(1) invece di O(n) per una copia
3  #pragma omp single
4  {
5      uint32_t *tmp = a;
6      a = out;
7      out = tmp;
8  }
9
10 // Dopo tutte le passate: verifica dove si trova il risultato
11 // Se passes e dispari, il risultato risulta nel buffer temporaneo
12 if(passes % 2 != 0){
13     // Copia finale necessaria: O(n)
14     memcpy(out, a, n * sizeof(uint32_t));
15 }
```

3.2.2 LSD In-Place: Problema e Soluzione

L'implementazione di un LSD Radix Sort in-place presenta sfide significative legate alla gestione delle permutazioni cicliche tra bucket.

Prima versione (instabile): Il primo tentativo di implementazione in-place processava i bucket in ordine sequenziale, tentando di spostare gli elementi fuori posto tramite swap diretti:

Listing 4: Approccio iniziale (instabile)

```
// PROBLEMA: questo approccio non gestisce correttamente
// i cicli quando elementi di bucket gia processati
// vengono coinvolti in swap con bucket futuri

// Definisco bucket di lavoro
for(int b=0; b<base; b++){
    size_t end_b = (b+1 < base) ? start[b+1] : n;

    // Guardo elementi del mio bucket
    for(size_t i = start[b]; i < end_b; ){
        uint32_t val = a[i];
        unsigned d = (val >> shift) & mask;

        if((int)d == b){
            i++;  // Elemento gia nel bucket corretto
        }
        else if((int)d < b){
            // ERRORE: elemento che dovrebbe stare in un
            // bucket gia completato - impossibile da gestire
            i++;  // Skip forzato -> risultato corrotto
        }
        else{
            // d > b: swap con bucket futuro
            size_t target = pos[d];
            uint32_t temp = a[target];
            a[target] = val;
            a[i] = temp;
            pos[d]++;
        }
    }
}
```

Questa implementazione funziona correttamente solo alla prima passata, quando l'array è ancora in ordine casuale. Nelle passate successive, la struttura parzialmente ordinata causa conflitti irrisolvibili tra bucket già elaborati.

Soluzione: American Flag Sort con cicli completi La versione corretta, basata sull'algoritmo descritto in "Engineering Radix Sort" di Andersson e Nilsson [16], risolve il problema seguendo completamente ogni ciclo di permutazione prima di procedere:

Listing 5: American Flag Sort corretto

```
// Inizializza puntatori head e tail per ogni bucket
// head[b] = prossima posizione libera nel bucket b
// tail[b] = fine del bucket b
for(int b=0; b<base; b++){
    head[b] = start[b];
    tail[b] = (b+1 < base) ? start[b+1] : n;
}

// Processa ogni bucket sequenzialmente
for(int b=0; b<base; b++){
    // Continua finche il bucket non e pieno
    while(head[b] < tail[b]){
        // Prendi l'elemento nella posizione corrente
        uint32_t elem = a[head[b]];
        unsigned digit = (elem >> shift) & mask;

        if((int)digit == b){
            // Caso semplice: elemento gia nel bucket corretto mi limito ad incrementare puntatore
            head[b]++;
        }
        else{
            // segui il ciclo di permutazione COMPLETO fino a tornare al bucket di partenza
            // non avanzo fino a che non ho elemento corretto
            while((int)digit != b){
                // Posizione di destinazione per questo elemento
                size_t swap_pos = head[digit];

                // Salva l'elemento che verra sovrascritto
                uint32_t temp = a[swap_pos];

                // inserisco l'elemento corrente
                a[swap_pos] = elem;
                head[digit]++;

                // L'elemento salvato diventa il nuovo elemento da inserire e continua il ciclo a catena
                elem = temp;
                digit = (elem >> shift) & mask;
            }
            // Il ciclo si e' chiuso: elem appartiene al bucket b
            // e va nella posizione head[b]
            a[head[b]] = elem;
            head[b]++;
        }
    }
}
```

La differenza fondamentale è che ogni ciclo viene seguito fino alla sua chiusura naturale, garantendo che nessun elemento venga "perso" o sovrascritto prima di essere stato riposizionato.

3.2.3 MSD Out-of-Place

L'implementazione MSD out-of-place utilizza un approccio ricorsivo con parallelismo basato su task OpenMP.

Gestione flessibile dei bit: Ad ogni livello di ricorsione, il numero di bit utilizzati viene adattato per non eccedere i bit rimanenti:

Listing 6: Calcolo adattivo dei bit per livello

```
// Calcola quanti bit usare a questo livello
// Non possiamo usare piu bit di quelli rimasti

unsigned use_bits = (32 - shift) < step_bits ?
                    (32 - shift) : step_bits;

// Se non ci sono piu bit da processare, termina
if(use_bits == 0) return;

// Calcola quanti bucket mi servono
const unsigned RADIX = 1u << use_bits;  // 2^use_bits bucket
// Creo maschera binaria
const unsigned MASK = RADIX - 1u;         // use_bits bit a 1
```

Early-exit optimization: Se tutti gli elementi appartengono allo stesso bucket, l'algoritmo procede direttamente al livello successivo senza effettuare scatter, risparmiando tempo e accessi in memoria:

Listing 7: Ottimizzazione early-exit

```
// Dopo il counting, verifica se tutti gli elementi
// hanno lo stesso digit (sono tutti nello stesso bucket)
for(unsigned d = 0; d < RADIX; ++d){
    if(count[d] == n){
        // Tutti gli n elementi hanno digit == d
        // Non serve fare scatter: sono gia contigui
        // Libero memoria e puntatori
        free(bbegin); free(offset); free(count);

        // Procedi direttamente al livello successivo
        // shift e la pos corrente
        int next_shift = shift - (int)step_bits;
        if(next_shift >= 0)
            // Se ho ancora da ordinare richiamo ricorsivamente la
               funzione
            msd_rec_level(src, dst, lo, hi, next_shift, step_bits);
        return;
    }
}
```

Parallelismo con task: I bucket vengono processati ricorsivamente, con creazione dinamica di task OpenMP per bucket sufficientemente grandi:

Listing 8: Ricorsione parallela con taskgroup

```
1
2  // Bucket piu' piccoli vengono processati sequenzialmente
3  // per evitare overhead eccessivo
4
5  #pragma omp taskgroup  // Attende completamento di tutti i task
       figli
6  {
7      for(unsigned d = 0; d < RADIX; ++d){
8          size_t m = b_hi - b_lo;  // Dimensione del bucket
9          if(m <= 1) continue;       // Bucket vuoto o singolo
               elemento
10
11         if(m >= TASK_THRESHOLD){
12             // Bucket grande: crea un task parallelo
13             // firstprivate: ogni task ha la propria copia PRIVATA
                   dei parametri
14             #pragma omp task firstprivate(b_lo, b_hi, next_shift)
15             {
16                 // Nota: dst e src sono invertiti per il ping pong
17                 //ping pong ricorsivo
18                 msd_rec_level(dst, src, b_lo, b_hi, next_shift,
                       step_bits);
19             }
20         } else {
21             // Bucket piccolo: esegui sequenzialmente
22             // Evita overhead di creazione task
23             msd_rec_level(dst, src, b_lo, b_hi, next_shift,
                   step_bits);
24         }
25     }
26 }
27 // Dopo taskgroup: tutti i bucket sono stati ordinati
28
29 // Copy-back per mantenere l'invariante:
30 // il risultato deve essere nel buffer src
31 memcpy(src + lo, dst + lo, n * sizeof(uint32_t));
```

3.2.4 MSD In-Place (American Flag Sort)

L'implementazione MSD in-place combina l'algoritmo American Flag Sort con il parallelismo task-based di OpenMP. Definisce due soglie per ottimizzare il trade-off tra overhead e parallelismo:

Listing 9: Soglie di ottimizzazione

```
1  // Soglia per creazione task paralleli
2
3  #define TASK_THRESHOLD  65536
```

```
4
5  // Soglia per fallback a comparison sort
6  // Bucket <= 2048 elementi: usa qsort (piu' efficiente)
7  #define QSORT_THRESHOLD 2048
```

Fallback a qsort: Per segmenti piccoli, l'overhead del radix sort (allocazioni, counting, permutazioni) supera i benefici. L'algoritmo passa a `qsort` della libreria standard, ottimizzato per piccoli array:

Listing 10: Fallback per segmenti piccoli

```
1  // Funzione di confronto per qsort
2  static int cmp_u32(const void* a, const void* b) {
3      uint32_t x = *(const uint32_t*)a;
4      uint32_t y = *(const uint32_t*)b;
5      return (x > y) - (x < y);  // Evita overflow
6  }
7
8  // All'inizio della ricorsione: verifica soglia
9  if(n <= QSORT_THRESHOLD){
10     // Per n <= 2048, qsort e piu veloce
11     // grazie a ottimizzazioni cache-friendly
12     qsort(a + lo, n, sizeof(uint32_t), cmp_u32);
13     return;
14 }
```

American Flag Positioning: Il cuore dell'algoritmo è il posizionamento in-place tramite swap, identico alla versione LSD corretta:

Listing 11: Posizionamento American Flag per MSD

```
1  // Per ogni bucket d, processa tutti gli elementi
2  for(unsigned d = 0; d < RADIX; ++d){
3      // next[d] = prossimo elemento da verificare/inserire
4      // end[d] = fine del bucket d
5      while(next[d] < end[d]){
6          // Prendi elemento alla posizione corrente
7          uint32_t v = a[next[d]];
8          // Calcola il bucket di appartenenza
9          unsigned dv = digit_u32(v, shift, MASK);
10
11         if(dv == d){
12             // Elemento gia nel bucket corretto e avanzo
13             next[d]++;
14         } else {
15             // Elemento fuori posto: swap con il bucket target
16             // Questo e il caso semplice (non segue cicli)
17             // perche MSD processa bucket indipendenti
18             uint32_t tmp = a[next[dv]];
19             a[next[dv]] = v;
20             a[next[d]] = tmp;
```

```
21                 next[dv]++;
22                 // NON incrementiamo next[d]: l'elemento swappato
23                 // potrebbe essere ancora fuori posto
24             }
25         }
26 }
```

Inizializzazione del parallelismo: Il punto di ingresso crea una regione parallela con un singolo task iniziale che poi si ramifica ricorsivamente:

Listing 12: Entry point con single task

```
1  // Calcola il primo shift (bit piu significativi)
2  //guardo se divisibile per la dimensione di step_bits se non lo e
       parto prima dal residuo poi proseguo
3  const unsigned rem = 32u % step_bits;
4  const unsigned first_bits = (rem == 0u) ? step_bits : rem;
5  const int first_shift = 32 - (int)first_bits;
6
7  #pragma omp parallel  // Crea il team di thread
8  {
9      #pragma omp single nowait  // Un solo thread inizia, in modo da
           ridurre overflow
10     {
11         // La ricorsione crea task per i bucket grandi
12         msd_rec_u32_ip(a, 0, n, first_shift, step_bits);
13     }
14 }
15 // Alla fine: tutti i task completati, array ordinato
```

3.2.5 PARADIS: Parallel In-Place Radix Sort

L'algoritmo PARADIS (PARAllel DIgital Sort) rappresenta un approccio avanzato per l'ordinamento radix in-place parallelo, basato sull'implementazione di Cho et al. [5]. A differenza delle implementazioni tradizionali, PARADIS permette a più thread di eseguire permutazioni simultaneamente tramite una tecnica di speculative permutation seguita da una fase di repair.

Struttura dell'algoritmo: L'algoritmo opera in più fasi per ogni livello di digit (8 bit fissi):

1. Istogramma parallelo: Ogni thread calcola l'istogramma locale sul proprio chunk
2. Riduzione: Combinazione degli istogrammi in conteggi globali
3. Partizionamento: Calcolo dei range di ogni bucket e suddivisione tra thread
4. Permutazione speculativa: Ogni thread esegue swap nel proprio range, potenzialmente creando conflitti
5. Repair: Correzione degli elementi rimasti fuori posto dopo la fase speculativa

6. Ricorsione: Applicazione ricorsiva sui bucket con bilanciamento dinamico dei thread

Listing 13: Permutazione speculativa con repair

```
// Ogni thread lavora sul proprio range.
// per ogni bucket i, eseguendo swap in modo "speculativo"
for(int i = 0; i < kRadixBin; i++){
    long long head = ph[pID][i];

    while(head < pt[pID][i]){
        // Accesso in memoria prendendo valore
        uint32_t v = *(begin_itr + head);
        // Estraggo la chiave cioe porzione di bit/byte da lavorare
        int k = determineDigitBucket(kth_byte, v);

        // Segui la catena di swap finche' possibile
        // nel range locale del thread
        while(k != i && ph[pID][k] < pt[pID][k]){
            //sistemo il valore v
            _swap(&v, (begin_itr + ph[pID][k]));
            //avanzo con puntatore
            ph[pID][k]++;
            //determino bucket e ricomincio giro
            k = determineDigitBucket(kth_byte, v);
        }

        // Inserire l'elemento (potrebbe essere speculativo)
        // Gestisco la scrittura finale di v
        if(k == i){
            *(begin_itr + head) = *(begin_itr + ph[pID][i]);
            head++;
            *(begin_itr + ph[pID][i]) = v;
            ph[pID][i]++;
        } else {
            // Elemento fuori range: sara' corretto nella repair
               phase
            *(begin_itr + head) = v;
            head++;
        }
    }
}
```

Bilanciamento dinamico dei thread: La ricorsione sui bucket utilizza una formula logaritmica per distribuire i thread in base alla dimensione relativa di ogni bucket:

Listing 14: Allocazione dinamica dei thread e strategia ibrida

```
// Calcolo adattivo delle risorse computazionali.
// Non dividiamo i thread equamente, ma usiamo una metrica basata
   sulla complessita attesa O(N log N) per assegnare piu "potenza"
   ai bucket piu densi.
// Questo previene il fenomeno dei thread ritardatari.
```

```
int nextStageThreads = processes * (cnt[i] * (log(cnt[i]) / log(
    kRadixBin)) /
      (elenum * (log(elenum) / log(kRadixBin))));

if(cnt[i] > 64LL){
    // Se il bucket supera la soglia critica, generiamo un task
       asincrono. Lo scheduler OpenMP gestira il lavoro per
       bilanciare il carico tra i core disponibili.
    #pragma omp task
    PARADIS_core(kth_byte - 1, begin_itr + starts[i],
                 begin_itr + (starts[i] + cnt[i]), begin_itr,
                 (nextStageThreads > 1 ? nextStageThreads : 1));
} else if(cnt[i] > 1){
    // Switch ad un algoritmo di ordinamento sul posto, per dataset
        cosi piccoli (N <= 64), l'Insertion Sort e piu veloce con
       overhead nullo.
    insert_sort_u32(begin_itr + starts[i],
                    begin_itr + (starts[i] + cnt[i]));
}
```

Vantaggi e limitazioni: PARADIS offre un vero parallelismo in-place senza buffer $O(n)$, ma introduce overhead dovuto alla fase di repair. È più efficiente su architetture con molti core e quando la memoria è un vincolo critico.

3.2.6 Approccio Ibrido MSD-LSD

L'implementazione ibrida combina i vantaggi di entrambi i paradigmi MSD e LSD, ispirandosi al lavoro di Stehle e Jacobsen [9]. La strategia sfrutta MSD per il partizionamento iniziale e LSD per l'ordinamento efficiente all'interno dei bucket.

Listing 15: Configurazione ibrido MSD-LSD

Strategia a due fasi:

```
// Fase 1: MSD sui primi 8 bit -> 256 bucket indipendenti
#define MSD_BITS 8   // 256 bucket
#define MSD_SHIFT 24 // bit 31-24 (MSB)

// Fase 2: LSD sui rimanenti 24 bit (3 passate da 8 bit)
#define LSD_BITS 8   // 8 bit per passata
#define LSD_PASSES 3 // ceil(24/8) = 3 passate
```

Fase 1: Partizionamento MSD Il primo livello utilizza MSD out-of-place per dividere l'array in 256 bucket disgiunti:

Listing 16: Partizionamento iniziale con MSD

```
// Counting parallelo sui bit piu significativi
// suddivido lavoro su Tloc thread
#pragma omp parallel
{
```

```
    // ogni thread scrive in una zona di memoria esclusiva,
       eliminando la contesa (cache coherence traffic) tra i core
       della CPU.
    const int tid = omp_get_thread_num();
    size_t *lc = thread_counts + tid * MSD_RADIX;

    // Ogni thread conta nel proprio chunk, con determinato
       intervallo ottenendo una gestione del carico
    size_t s = (n * tid) / Tloc;
    size_t e = (n * (tid + 1)) / Tloc;

    for(size_t i = s; i < e; i++){
        // Estrae i primi 8 bit piu significativi, operando con
           maschera 0xFF ed isolando l'ultimo byte per prossima
           passata di sorting
        unsigned d = (a[i] >> 24) & 0xFF;
        lc[d]++;
    }
}

// Riduzione, prefix-sum e scatter (out-of-place) come standard MSD
```

Fase 2: Ordinamento LSD parallelo sui bucket Ogni bucket viene ordinato indipendentemente con LSD, sfruttando il parallelismo a livello di bucket:

Listing 17: LSD parallelo sui bucket

```
// Ogni bucket puo essere processato in parallelo
// schedule(dynamic): Il runtime di OpenMP assegnera i bucket ai
   thread man mano che si liberano, ottimizzando il throughput.
#pragma omp parallel for schedule(dynamic)
for(int bucket = 0; bucket < MSD_RADIX; bucket++){
    // calcolo limiti del bucket corrente, ottendo blocchi contigui
       di memoria
    size_t bucket_start = offset[bucket];
    size_t bucket_size = count[bucket];

    if(bucket_size > 1){
        // LSD sui 24 bit rimanenti
        //Poiche i numeri nel bucket condividono gia i primi 8 bit
           (MSB), ordiniamo i restanti 3 byte.
        //L'LSD e preferito qui perche e iterativo e non richiede
           la creazione di ulteriori task ricorsivi, riducendo
           drasticamente l'overhead di gestione.
        lsd_bucket(a + bucket_start, bucket_size,
                   0,           // start_shift: bit 0
                   lsd_passes); // numero passate: 3
    }
}
```

Vantaggi dell'approccio ibrido:

- Località di cache: I bucket creati da MSD hanno dimensioni ridotte ($\sim n/256$), migliorando l'efficienza delle passate LSD
- Parallelismo a due livelli: MSD parallelizza il counting, LSD parallelizza sui bucket
- Ricorsione limitata: MSD applicato solo al primo livello, evitando l'overhead della ricorsione profonda
- Bilanciamento dinamico: `schedule(dynamic)` gestisce automaticamente bucket di dimensioni diverse

3.2.7 Riepilogo Completo delle Implementazioni CPU

Variante	Metodo	Struttura	Parallelismo	Note
LSD Out-of-Place	LSD	Out-of-place	Data-parallel	Stabile, semplice
LSD In-Place v1	LSD	In-place	Histogram only	Instabile
LSD In-Place v2	LSD	In-place	Histogram only	Stabile (cicli)
MSD Out-of-Place	MSD	Out-of-place	Task-based	Early-exit
MSD In-Place	MSD	In-place	Task-based	American Flag
PARADIS	MSD	In-place	Speculative	Repair phase
Ibrido MSD-LSD	MSD+LSD	Out-of-place	Two-level	Cache-friendly

Tabella 5: Riepilogo completo implementazioni CPU OpenMP

3.3 Implementazioni GPU con CUDA

Le implementazioni GPU sfruttano il parallelismo massivo delle architetture NVIDIA per accelerare l'ordinamento. Sono state sviluppate due varianti basate su librerie ottimizzate: Thrust e CUB.

3.3.1 Thrust: Interfaccia ad Alto Livello

Thrust è una libreria di template C++ per CUDA che fornisce un'interfaccia simile alla STL. Per tipi interi, `thrust::sort` seleziona automaticamente un'implementazione LSD Radix Sort ottimizzata.

Caratteristiche:

- API minimale: una singola chiamata per l'ordinamento
- Gestione automatica della memoria temporanea
- Ottimizzazione trasparente per diversi tipi di dati
- Internamente utilizza primitive CUB

Listing 18: Ordinamento con Thrust

```
1 #include <thrust/sort.h>
2 #include <thrust/device_ptr.h>
3
```

```cpp
void thrust_radix_sort(uint32_t *d_keys, size_t n) {
    // Crea un "device_ptr" dal puntatore raw CUDA
    // Permette a Thrust di avere gia i dati in VRAM della scheda video
    thrust::device_ptr<uint32_t> d_ptr(d_keys);

    // Chiamata all'ordinamento:
    // Thrust rileva il tipo uint32_t, seleziona automaticamente LSD Radix Sort, Gestisce internamente la memoria temporanea e lancia migliaia di piccoli thread GPU
    // L'ordinamento avviene manipolando i bit in parallelo su una banda passante di memoria molto piu larga rispetto a quella del sistema.
    thrust::sort(d_ptr, d_ptr + n);
}
```

Misurazione delle prestazioni: Il tempo di esecuzione viene misurato tramite CUDA Events, che forniscono precisione a livello di microsecondi:

Listing 19: Timing con CUDA Events

```cpp
// Crea eventi CUDA per misurazione precisa
// uso direttamente i timing della scheda video
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Registra timestamp di inizio, come fosse flag di inizio
cudaEventRecord(start);

// Esegui ordinamento sulla GPU, viene lanciata dalla CPU che poi torna disponibile mentre la nostra GPU lavora
thrust_radix_sort(d_data, size);

// Registra timestamp di fine, flag di fine
cudaEventRecord(stop);

// Sincronizza: attendi completamento di tutte le operazioni
cudaDeviceSynchronize();

// Calcola tempo trascorso in millisecondi
float elapsed_ms;
cudaEventElapsedTime(&elapsed_ms, start, stop);
double elapsed_sec = elapsed_ms / 1000.0;
```

3.3.2 CUB: Controllo a Basso Livello

CUB (CUDA UnBound) è la libreria di primitive parallele su cui si basa Thrust. Offre maggiore controllo sulla gestione della memoria e permette ottimizzazioni specifiche.

Caratteristiche:

- Controllo esplicito sulla memoria temporanea
- Pattern a due fasi: query dimensione + esecuzione
- Supporto per `DoubleBuffer` (ping-pong efficiente)
- Prestazioni leggermente superiori a Thrust (3-7%)

Listing 20: Ordinamento con CUB e DoubleBuffer

```
#include <cub/device/device_radix_sort.cuh>
#include <cub/util_type.cuh>  // Per DoubleBuffer

void cub_radix_sort(uint32_t *d_keys, size_t n) {
    // CUB richiede un buffer di appoggio per spostare i dati (Ping-Pong).
    // Questo permette di scrivere in modo sequenziale, massimizzando la banda della VRAM.
    uint32_t *d_keys_out;
    cudaMalloc(&d_keys_out, n * sizeof(uint32_t));

    // DoubleBuffer gestisce automaticamente lo scambio tra buffer sorgente e destinazione ad ogni passata, scambia i ruoli tra d_keys e d_keys_out
    cub::DoubleBuffer<uint32_t> d_keys_buffers(d_keys, d_keys_out);

    // prima fase di Query dimensiona la memoria temporanea
    // Prima chiamata con d_temp_storage = nullptr
    void *d_temp_storage = nullptr;
    size_t temp_storage_bytes = 0;

    cub::DeviceRadixSort::SortKeys(
        d_temp_storage, temp_storage_bytes,  // Output: bytes necessari
        d_keys_buffers, n);                  // Input: buffer e dimensione

    // Alloca memoria temporanea per istogrammi
    cudaMalloc(&d_temp_storage, temp_storage_bytes);

    // seconda fase esegue ordinamento effettivo, avendo ora l'area di lavoro d_temp_storage
    // CUB poi sfrutta come thrust kernel specifici della GPU
    cub::DeviceRadixSort::SortKeys(
        d_temp_storage, temp_storage_bytes,
        d_keys_buffers, n);

    // Per uint32_t con radix 8-bit -> 4 passate (pari)
    // Il risultato e gia in d_keys, non serve copia finale

    // Cleanup delle aree di memoria e hardware utilizzate
```

```
    cudaFree(d_temp_storage);
    cudaFree(d_keys_out);
}
```

Nota sul DoubleBuffer: CUB utilizza internamente la tecnica ping-pong per evitare copie tra passate. Per `uint32_t` con radix a 8 bit, vengono eseguite esattamente 4 passate (numero pari), quindi il risultato finale si trova automaticamente nel buffer originale `d_keys`.

3.3.3 Riepilogo Implementazioni GPU

Variante	Libreria	Architettura	Note
Thrust	Thrust	GPU only	API semplice, gestione auto memoria
CUB	CUB	GPU only	Controllo fine, 3-7% più veloce

Tabella 6: Riepilogo implementazioni GPU CUDA

Confronto Thrust vs CUB: Nei benchmark, CUB ha mostrato prestazioni consistentemente superiori del 3-7% rispetto a Thrust, grazie al controllo più fine sulla memoria e all'eliminazione di overhead di astrazione. Tuttavia, Thrust offre un eccellente compromesso tra semplicità d'uso e prestazioni.

3.3.4 Scaling Multi-GPU

Le implementazioni multi-GPU estendono gli algoritmi single-GPU con una strategia divide-sort-merge che sfrutta più GPU A100 in parallelo. Il sort locale su ogni GPU può utilizzare Thrust o CUB (con CUB leggermente più performante), mentre il merge finale utilizza sempre `thrust::merge`.

Fase 1: Partizionamento e Sort Parallelo I dati vengono suddivisi equamente tra le GPU disponibili. Ogni GPU riceve il proprio chunk tramite trasferimenti asincroni su CUDA Streams dedicati, permettendo overlap tra comunicazione e computazione:

Listing 21: Partizionamento e sort parallelo multi-GPU

```
// Calcola chunk bilanciati in base al numero di GPU e gestendo
   anche i possibili residui
size_t base_chunk = n / num_gpus;
size_t remainder = n % num_gpus;
for (int i = 0; i < num_gpus; i++) {
    chunk_sizes[i] = base_chunk + (i < remainder ? 1 : 0);
    chunk_offsets[i] = offset;
    offset += chunk_sizes[i];
}

// Trasferimento asincrono H2D su ogni GPU
for (int gpu = 0; gpu < num_gpus; gpu++) {
    // Seleziona la GPU target per le operazioni successive
    CUDA_CHECK(cudaSetDevice(gpu));
```

```
    // Crea un flusso (stream) indipendente per ogni dispositivo in
        modo tale da avere un'esecuzione asincrona
    CUDA_CHECK(cudaStreamCreate(&streams[gpu]));
    // Riserva memoria globale sulla VRAM della GPU specifica
    CUDA_CHECK(cudaMalloc(&d_chunks[gpu],
                          chunk_sizes[gpu] * sizeof(uint32_t)));
    //Avvia il trasferimento dati Host-to-Device sempre
       asincornamente
    CUDA_CHECK(cudaMemcpyAsync(d_chunks[gpu],
                               h_data + chunk_offsets[gpu],
                               chunk_sizes[gpu] * sizeof(uint32_t),
                               cudaMemcpyHostToDevice, streams[gpu
                                  ]));
}

// Sort parallelo: ogni GPU ordina il proprio chunk
for (int gpu = 0; gpu < num_gpus; gpu++) {
    CUDA_CHECK(cudaSetDevice(gpu));
    // Thrust: thrust::sort(thrust::cuda::par.on(streams[gpu]),
       ...)
    // CUB:    cub::DeviceRadixSort::SortKeys(..., streams[gpu])
}

// Sincronizzazione  dei flussi di esecuzione.
for (int gpu = 0; gpu < num_gpus; gpu++) {
    // Seleziona il dispositivo target
    CUDA_CHECK(cudaSetDevice(gpu));
    // Mette in pausa il thread della CPU finche lo stream 'gpu'
       non ha terminato tutte le operazioni in coda. Questo
       garantisce che i dati sulla VRAM siano ora completamente
       ordinati.
    CUDA_CHECK(cudaStreamSynchronize(streams[gpu]));
}
```

Fase 2: Comunicazione e Merge I chunk ordinati vengono trasferiti su GPU 0 tramite `cudaMemcpyPeer`, che sfrutta il bus PCIe per comunicazione diretta GPU-to-GPU. I chunk vengono poi combinati iterativamente con `thrust::merge`, che ha complessità $O(n)$ per array già ordinati:

Listing 22: Comunicazione inter-GPU e merge iterativo

```
// Elegge di default la GPU 0 come coordinatore della funzione
CUDA_CHECK(cudaSetDevice(0));

// Inizializza il buffer di destinazione con il primo chunk gia
   presente su GPU 0.
CUDA_CHECK(cudaMemcpy(d_merged, d_chunks[0],
                      chunk_sizes[0] * sizeof(uint32_t),
                      cudaMemcpyDeviceToDevice));
size_t merged_size = chunk_sizes[0];

```

```
// Merge iterativo, unisce i risultati delle altre GPU uno alla volta
for (int gpu = 1; gpu < num_gpus; gpu++) {
    // Trasferimento diretto GPU-to-GPU (non passa per la CPU)
    CUDA_CHECK(cudaMemcpyPeer(d_chunk_on_gpu0, 0,
                              d_chunks[gpu], gpu,
                              chunk_sizes[gpu] * sizeof(uint32_t)));

    // Fusione lineare di due sequenze gia ordinate sfruttando l'algoritmo merge di Thrust
    thrust::merge(merged_ptr, merged_ptr + merged_size,
                  chunk_ptr, chunk_ptr + chunk_sizes[gpu],
                  temp_ptr);

    // Scambia i puntatori dei buffer evitando cosi le copie dei dati
    std::swap(d_merged, d_temp);  // Ping-pong
    merged_size += chunk_sizes[gpu];
}
```

Limiti dello scaling: L'efficienza multi-GPU è limitata da:

- **Bandwidth PCIe:** `cudaMemcpyPeer` è limitato a ∼22 GB/s, molto inferiore alla bandwidth HBM2 (1935 GB/s)
- **Merge sequenziale:** La fase di merge è seriale, limitando lo speedup secondo la legge di Amdahl

Profilazione delle fasi: Per analizzare i colli di bottiglia, la versione profilata misura separatamente i tempi di Sort, Communication e Merge:

Listing 23: Struttura di profilazione

```
struct ProfilingData {
    double sort_time = 0.0;    // Tempo di sort parallelo
    double comm_time = 0.0;    // Tempo di trasferimento inter-GPU
    double merge_time = 0.0;   // Tempo di merge su GPU 0
    double total_time = 0.0;   // Tempo totale end-to-end
};
```

Analisi delle prestazioni multi-GPU: L'efficienza dello scaling multi-GPU è limitata da due fattori principali:

- **Bandwidth PCIe:** Il trasferimento inter-GPU via `cudaMemcpyPeer` è limitato a ∼22 GB/s sul cluster UNIPR, significativamente inferiore alla bandwidth della memoria HBM2 (1935 GB/s)
- **Merge sequenziale:** La fase di merge è intrinsecamente seriale, creando un collo di bottiglia che limita lo speedup massimo ottenibile.

Variante	Libreria	GPU	Note
Thrust Single	Thrust	1	API semplice
CUB Single	CUB	1	3-7% più veloce
Thrust Multi-GPU	Thrust	1-4	Divide-sort-merge
CUB Multi-GPU Profiled	CUB	1-4	Con breakdown temporale

Tabella 7: Riepilogo implementazioni GPU (single e multi)

3.4 Metodologia di Testing su Cluster

Per l'esecuzione dei benchmark sul cluster UNIPR, sono stati sviluppati script SLURM con struttura standardizzata. Ogni script segue il seguente schema:

Direttive SLURM comuni:

- `-partition`: `cpu` per test OpenMP, `gpu` per test CUDA
- `-nodes=1`: esecuzione su singolo nodo
- `-cpus-per-task`: 64 core per test CPU, 8-16 core per test GPU
- `-gres=gpu:a100_80g:N`: richiesta di N GPU A100 80GB (solo per test GPU)
- `-time`: walltime variabile da 5 a 15 minuti in base alla complessità del test

Flusso di esecuzione:

1. Caricamento moduli: `gcc/13.2.0` per compilazione C/OpenMP, `cuda/12.4.1` per CUDA
2. Compilazione in-place: ogni script compila il codice sorgente con flag di ottimizzazione:
   - CPU: `-O3 -march=native -fopenmp`
   - GPU: `-O3 -arch=sm_80`
3. Verifica compilazione: controllo del codice di uscita con terminazione in caso di errore
4. Esecuzione test: iterazione su configurazioni variabili (thread count, GPU count, dataset size)
5. Ripetizioni: ogni configurazione eseguita con `-repeats=5` per affidabilità statistica

Tipologie di test implementate:

- Scaling OpenMP: test da 1 a 64 thread per algoritmi LSD, MSD, PARADIS e Hybrid
- Test GPU singola: CUB e Thrust su dataset da 10K a 100M elementi
- Scaling Multi-GPU: test con 1, 2 e 4 GPU A100 per valutare parallelismo inter-device

- Test profilati: esecuzione su dataset da 1B, 2B e 5B elementi con breakdown temporale (Sort, Communication, Merge)

I dataset di input sono file binari pre-generati contenenti chiavi a 32 bit distribuite uniformemente, garantendo riproducibilità dei risultati tra esecuzioni successive.

4 Analisi e Risultati

Questa sezione presenta i risultati sperimentali ottenuti dalle diverse implementazioni, prima sull'ambiente locale e successivamente sul cluster HPC.

4.1 Risultati in Ambiente Locale

I test locali sono stati eseguiti su workstation con GPU NVIDIA RTX 4060 (8 GB GDDR6, architettura Ada Lovelace) e CPU AMD Ryzen 7 8845HS (16 thread). Per le implementazioni CPU è stato utilizzato radix a 8 bit.

4.1.1 CPU: Confronto Implementazioni

Tabella 8: Prestazioni CPU locali - 16 thread, radix 8 bit

Algoritmo	100M (s)	10M (s)	1M (s)	10K (s)
MSD In-Place	2.160	0.919	0.065	0.006
MSD Out-Place	2.113	1.677	0.085	0.008
LSD In-Place (v2)	4.290	0.593	0.062	0.041
LSD Out-Place	0.431	0.068	0.020	0.011
PARADIS	0.536	0.191	0.011	0.002
Hybrid LSD+MSD	1.963	0.196	0.044	0.010

Osservazioni:

- LSD Out-Place è il più veloce su dataset grandi (100M: 0.431s)
- PARADIS eccelle su dataset piccoli grazie al minor overhead
- Le varianti MSD sono più lente a causa del maggior overhead di ricorsione

4.1.2 GPU: Thrust vs CUB

Tabella 9: Prestazioni GPU locali - RTX 4060, radix 8 bit

Libreria	100M (s)	10M (s)	1M (s)	10K (s)
CUB	0.0396	0.00655	0.00390	0.00147
Thrust	0.0447	0.00743	0.00334	0.00157

Osservazioni:

- CUB risulta 13% più veloce di Thrust su RTX 4060 (100M)
- GPU è 10.9x più veloce della migliore CPU locale (0.0396s vs 0.431s)
- Su dataset piccoli (1M), Thrust è leggermente più veloce

4.2 Risultati sul Cluster HPC

I test sul cluster UNIPR sono stati eseguiti su nodi con GPU NVIDIA A100 80GB e CPU dual-socket (64 thread totali).

4.2.1 CPU: Scaling per Algoritmo (100M elementi)

Tabella 10: Scaling CPU - MSD (100M elementi)

Algoritmo	1T	2T	4T	8T	16T	32T	64T
MSD In-Place	8.888	5.377	3.462	2.622	1.903	1.561	2.218
MSD Out-Place	14.959	11.961	7.430	4.171	3.040	1.903	1.737

Tabella 11: Scaling CPU - LSD (100M elementi)

Algoritmo	1T	2T	4T	8T	16T	32T	64T
LSD In-Place	7.455	8.140	7.712	7.881	7.744	6.932	7.186
LSD Out-Place	2.159	1.580	1.233	0.676	0.388	0.306	0.243

Tabella 12: Scaling CPU - PARADIS e Hybrid (100M elementi)

Algoritmo	1T	2T	4T	8T	16T	32T	64T
PARADIS	5.034	2.606	1.565	0.826	0.449	0.237	0.167
Hybrid LSD+MSD	1.504	1.173	0.931	0.864	0.952	0.786	0.852

Analisi dello scaling:

- LSD Out-Place: scaling eccellente, da 2.159s (1T) a 0.243s (64T) = 8.88x speedup
- PARADIS: miglior scaling assoluto, da 5.034s (1T) a 0.167s (64T) = 30.1x speedup
- MSD In-Place: prestazioni ottimali a 32T (1.561s), degrada a 64T
- LSD In-Place: scaling quasi nullo (problema di contesa)
- Hybrid: scaling irregolare, ottimale a 32T

Tabella 13: Confronto GPU - CUB vs Thrust

GPU	Libreria	100M (s)	10M (s)	1M (s)	10K (s)
RTX 4060	CUB	0.0396	0.00655	0.00390	0.00147
RTX 4060	Thrust	0.0447	0.00743	0.00334	0.00157
A100	CUB	0.00517	0.00151	0.00088	0.00029
A100	Thrust	0.00472	0.00138	0.00055	0.00027

4.2.2 GPU: A100 vs RTX 4060

Osservazioni:

- Su RTX 4060: CUB è 13% più veloce di Thrust
- Su A100: Thrust è 9% più veloce di CUB
- A100 è 7.7x più veloce di RTX 4060 su 100M (0.00472s vs 0.0396s)
- L'inversione CUB/Thrust suggerisce ottimizzazioni Thrust specifiche per architettura del Cluster

4.2.3 Risultati Multi-GPU Thrust

Tabella 14: Multi-GPU Thrust - 2B elementi

GPU	Sort (s)	Comm (s)	Merge (s)	Totale (s)	Throughput	Speedup
1	-	-	-	1.536	1302 M/s	1.00x
2	0.722	0.344	0.054	1.949	1026 M/s	0.79x
4	0.752	0.341	0.154	2.182	917 M/s	0.70x

Analisi dell'overhead Thrust:

- 2 GPU: sorting scala bene (2.13x, 106% efficienza), ma comunicazione (0.344s) e merge (0.054s) annullano i benefici
- 4 GPU: sorting peggiora (2.04x invece di 4x atteso), merge triplica (0.154s) per i 3 merge sequenziali necessari
- Speedup totale decresce passando da 2 a 4 GPU (0.79x $\rightarrow$ 0.70x)

4.2.4 Risultati Multi-GPU CUB

Tabella 15: Multi-GPU CUB - 2B elementi

GPU	Sort (s)	Comm (s)	Merge (s)	Totale (s)	Throughput	Speedup
1	-	-	-	1.436	1393 M/s	1.00x
2	0.689	0.321	0.036	1.882	1063 M/s	0.76x
4	0.749	0.344	0.100	2.026	987 M/s	0.71x

Analisi dell'overhead CUB:

- 2 GPU: sorting scala bene (2.08x, 104% efficienza), comunicazione (0.321s) e merge (0.036s) limitano le prestazioni
- 4 GPU: sorting peggiora (1.92x invece di 4x atteso), merge quasi triplica (0.100s)
- CUB ha merge più efficiente di Thrust (0.100s vs 0.154s con 4 GPU)

Conclusioni Multi-GPU: In entrambe le librerie, l'overhead di comunicazione PCIe (0.32-0.34s) e il merge sequenziale rendono il multi-GPU controproducente:

- La banda PCIe effettiva (22 GB/s) è solo il 34% del teorico
- Il merge scala linearmente con il numero di GPU (più GPU = più merge)
- Single GPU rimane ottimale per dataset $<$ 10-15 miliardi di elementi

4.3 Analisi Multi-CPU

I test CPU con OpenMP mostrano buona scalabilità fino a 8 thread, con efficienza dell'80%. Oltre questo punto, i benefici si riducono a causa di:

- Architettura: latenza di accesso tra socket diversi
- Sincronizzazione: overhead delle barriere OpenMP
- Fase sequenziale: la distribuzione degli elementi nei bucket rimane difficile da parallelizzare efficacemente

4.4 Riepilogo Comparativo

Tabella 16: Classifica prestazioni - 2B elementi

Rank	Configurazione	Tempo (s)	Throughput
1	GPU 1x A100 (CUB)	1.436	1393 M/s
2	GPU 1x A100 (Thrust)	1.536	1302 M/s
3	GPU 2x A100 (CUB)	1.882	1063 M/s
4	GPU 4x A100 (CUB)	2.026	987 M/s
5	CPU 64T (OpenMP)	3.427	584 M/s

Conclusioni principali:

1. Single GPU è la configurazione ottimale per dataset da 100M a 2B di elementi testati
2. CUB supera Thrust del 3-7% grazie al controllo fine sulla memoria
3. Multi-GPU non scala su architettura cluster (overhead comunicazione domina)

5 Discussione dei Risultati

5.1 Validazione dell'Approccio GPU-First

Fin dalle prime fasi di ricerca bibliografica, la letteratura scientifica indicava chiaramente le GPU come piattaforma ideale per il Radix Sort. Algoritmi non comparativi come il Radix Sort, caratterizzati da pattern di accesso alla memoria predicibili e operazioni altamente parallelizzabili, si prestano naturalmente all'architettura massivamente parallela delle GPU moderne.

I test preliminari condotti in ambiente locale hanno confermato questa direzione: su una RTX 4060, sia CUB che Thrust hanno mostrato throughput superiori a 2600 M keys/s, contro i circa 400 M keys/s ottenibili con OpenMP su 16 thread. Il confronto con algoritmi comparison-based tradizionali rafforza ulteriormente questa scelta: Quicksort raggiunge circa 12 M keys/s su singolo thread, mentre le nostre implementazioni GPU risultano oltre 200 volte più veloci.

5.2 Scalabilità e il Problema dell'Overhead Multi-GPU

Incoraggiati dai risultati su singola GPU, l'estensione naturale è stata esplorare configurazioni multi-device sul cluster UNIPR, dotato di 4 GPU A100 80GB per nodo. L'obiettivo era verificare se il parallelismo multi-GPU potesse fornire ulteriori speedup proporzionali al numero di dispositivi.

I risultati hanno rivelato una realtà più complessa. Mentre la fase di sorting pura scala quasi idealmente (2.08x speedup con 2 GPU, efficienza del 104%), il tempo totale risulta paradossalmente peggiore rispetto alla singola GPU:

- 1 GPU: 1.436s (baseline)
- 2 GPU: 1.882s (0.76x, rallentamento del 31%)
- 4 GPU: 2.026s (0.71x, rallentamento del 41%)

Il colpevole è l'overhead di comunicazione e merge, che introduce costi fissi indipendenti dalla dimensione del dataset.

5.3 Break-Even Point e Limiti della Memoria

L'analisi teorica suggerisce che il multi-GPU diventi vantaggioso solo oltre i 10-15 miliardi di elementi, quando l'overhead fisso viene ammortizzato sul tempo di sorting, che scala linearmente con la dimensione del dataset.

Tuttavia, i dati sperimentali mostrano che anche a 2B elementi il multi-GPU rimane svantaggioso (0.76x), indicando che il modello lineare sottostima gli overhead reali.

Questi risultati sono parzialmente coerenti con la letteratura esistente. Hongdi et al. [13] dimostrano che il radix sort multi-GPU può ottenere speedup significativi (fino a 7x con 16 GPU K80 su 268M elementi), ma riconoscono esplicitamente che: più GPU devono comunicare tra loro e sincronizzare i risultati del calcolo per mantenere la coerenza dei dati, identificando la comunicazione inter-GPU come collo di bottiglia intrinseco.

Abbiamo tentato di validare l'ipotesi del break-even point testando dataset da 5B e oltre, ma ci siamo scontrati con i limiti di memoria del cluster. Nonostante le A100 dispongano di 80GB ciascuna, la gestione di dataset da più 10B elementi (40GB solo per i

dati, più buffer temporanei per sorting e merge) ha generato errori di allocazione. Questo limite pratico ha impedito la verifica sperimentale del break-even point teorico, lasciando aperta la questione per future investigazioni con hardware dotato di maggiore memoria aggregata.

6 Conclusioni

Questo studio è nato dalla domanda fondamentale: come è possibile sfruttare efficacemente la parallelizzazione per l'algoritmo Radix Sort?

L'analisi condotta ha dimostrato che il Radix Sort è un algoritmo intrinsecamente adatto alla parallelizzazione: la sua natura non comparativa, con complessità O(d·n), e i pattern di accesso alla memoria predicibili lo rendono ideale per architetture massivamente parallele. I risultati confermano questa predisposizione: la fase di sorting pura scala quasi linearmente (efficienza del 104% con 2 GPU), e le implementazioni GPU superano di oltre 200 volte le performance di algoritmi comparison-based tradizionali come Quicksort.

Tuttavia, lo studio ha evidenziato come la gestione dell'overhead rappresenti la sfida principale nel passaggio da single-device a configurazioni multi-device. L'overhead di comunicazione e le fasi di merge sequenziale introducono costi fissi che, per dataset inferiori ai 10 miliardi di elementi, annullano completamente i benefici del parallelismo aggiuntivo. Questo risultato sottolinea un principio generale: aumentare le risorse computazionali non garantisce automaticamente miglioramenti prestazionali.

In conclusione, il Radix Sort si conferma un algoritmo eccellente per l'ordinamento di grandi volumi di dati interi, capace di raggiungere throughput superiori a 1.3 miliardi di chiavi al secondo su hardware moderno.

Riferimenti bibliografici

[1] Wikipedia contributors, Radix sort, Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/wiki/Radix_sort`

[2] GeeksforGeeks, Time and Space complexity of Radix Sort Algorithm, `https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-radix-sort-algorithm/`

[3] H. H. Seward, Information sorting in the application of electronic digital computers to business operations, `https://dspace.mit.edu/handle/1721.1/115112`

[4] R. Sedgewick, K. Wayne, Algorithms: Radix Sorts (Lecture Slides), Princeton University, `https://algs4.cs.princeton.edu/lectures/keynote/51StringSorts.pdf`

[5] J. Cho, S. S. Cho, e B. Egger, PARADIS: An Efficient Parallel In-place Radix Sort Algorithm, `https://dl.acm.org/doi/10.14778/2824032.2824050`

[6] J. Cho, SIMD Radix Sort Implementation, GitHub Repository, `https://github.com/jonicho/simd-radix-sort`

[7] N. Satish, M. Harris, e M. Garland, Designing Efficient Sorting Algorithms for Manycore GPUs, `https://www.nvidia.com/docs/io/67073/nvr-2008-001.pdf`

[8] K. Ha, J. Krüger, e C. T. Silva, Fast 4-Way Parallel Radix Sort, http://sci.utah.edu/~csilva/papers/cgf.pdf

[9] V. Stehle e H.-A. Jacobsen, A Hybrid MSD-LSD Radix Sort, https://dl.acm.org/doi/10.1145/3035918.3064043

[10] Z. Wei, J. L. Träff, e G. Guidi, DovetailSort: A Practical Parallel Radix Sort for Distributions with Heavy Keys, . https://dl.acm.org/doi/10.1145/3627535.3638483

[11] C. Li et al., RadiK: Accelerating Top-k Selection with Adaptive Bucket Scaling on GPUs, https://arxiv.org/abs/2501.14336

[12] M. G. Schmid, M. Korch, e T. Rauber, Fix Sort: A fast and efficient sorting algorithm for small fixed-size arrays on GPUs, https://doi.org/10.1016/j.parco.2021.102864

[13] H. Li et al., An Efficient Multi-GPU Radix Sort with Dynamic Bucket Classification, https://ieeexplore.ieee.org/document/10673948

[14] C. Mou et al., A GPU-Accelerated Radix-Hashing Scheme for Group-By Aggregation, https://ieeexplore.ieee.org/document/10386547

[15] K. Lykov, int-sort-bmk: Benchmark Suite for Integer Sorting Algorithms, https://github.com/KirillLykov/int-sort-bmk

[16] A. Andersson e S. Nilsson, *Implementing Radixsor*, https://doi.org/10.1145/297096.297136