SAPIENZA
UNIVERSITÀ DI ROMA

Operazioni collettive compresse su GPU e compressione omomorfica

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Informatica

Candidato

Francesco Carboni
Matricola 2058986

Relatore

Daniele De Sensi

Anno Accademico 2024/2025

Operazioni collettive compresse su GPU e compressione omomorfica
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: carbonifrancesco2003@gmail.com

«Science is what we understand well enough to explain to a computer. Art is everything else we do. [...] Science advances whenever an Art becomes a Science. And the state of the Art advances too, because people always leap into new territory once they have understood more about the old.»

— Donald Knuth

Sommario

L'efficienza delle operazioni collettive rappresenta un aspetto cruciale nelle applicazioni scientifiche che vengono eseguite sui moderni sistemi exa-scale. Queste operazioni soffrono spesso di colli di bottiglia a livello di interconnessione, soprattutto quando la trasmissione dei messaggi avviene tra nodi distinti. Per ridurre tale overhead, è possibile introdurre tecniche di compressione *lossy*, accettando un piccolo errore sui dati, così da diminuire il carico sulla rete.

In questo lavoro analizziamo compressori progettati per sfruttare al massimo la potenza di calcolo offerta dalle GPU, e approfondiamo diversi framework che integrano la compressione negli algoritmi di comunicazione collettiva. In particolare, viene studiato il framework hZCCL, che introduce la compressione omomorfica su CPU migliorando significativamente lo stato dell'arte. Si propone inoltre un nuovo compressore omomorfico, `HomComp`, basato su cuSZp2, dal quale derivano due nuovi algoritmi che integrano la compressione omomorfica nella collettiva di *ring allreduce*.

Infine, vengono analizzate le prestazioni delle diverse collettive e viene presentato un confronto con ghZCCL, un framework *GPU-centrico* di collettive con compressione omomorfica.

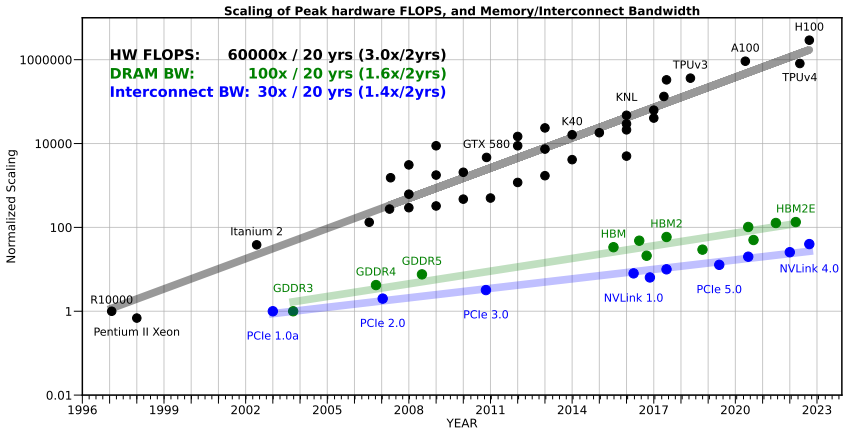
Indice

Introduzione

Le applicazioni scientifiche moderne e l'addestramento di LLM generano un enorme volume di dati e il trasferimento di questi ultimi costituisce uno dei maggiori colli di bottiglia all'interno dei sistemi di calcolo ad alte prestazioni (HPC). Da una parte c'è la rapida crescita nella potenza di calcolo delle GPU/CPU, dall'altra c'è una crescita molto più lenta nelle performance della memoria e nella larghezza di banda delle reti che interconnettono nodi di calcolo. La disparità di crescita delle due componenti porta a forti limitazioni nelle classi di applicazioni citate in precedenza, che richiedono calcolo parallelo, sincronizzazione e comunicazione dei risultati. Il fenomeno non è nuovo, era già stato notato in passato per le CPU. Nel 1995 William Wulf e Sally Mckee [32] coniarono il termine *memory wall*, e l'idea alla base è molto semplice:

> *il tempo per completare un'operazione è dipendente da quanto veloce si fanno i calcoli, ovvero eseguire operazioni aritmetiche su i dati, e da quanto velocemente possiamo fornire dati all'hardware che esegue i calcoli.*

Il grafico qui sotto mostra chiaramente come la potenza di calcolo sia cresciuta molto più velocemente rispetto alle capacità di memoria e interconnessioni. Questa discrepanza mette in evidenza uno dei colli di bottiglia più rilevanti nell'architettura moderna dei sistemi: il calcolo cresce più rapidamente rispetto alla capacità di alimentarlo con dati.



La scala della larghezza di banda delle diverse generazioni di interconnessioni e memorie e il picco di FLOPS. Come si può vedere la larghezza di banda delle interconnessioni e della memoria (rispettivamente blu e verde) aumenta molto più lentanmente rispetto alla potenza di calcolo[5].

Nel corso degli anni sono state esplorate diverse soluzioni sia hardware (Infiniband, HBM, NVlink [12, 18]), che implementazioni di algoritmi di comunicazione efficienti per far fronte al problema.

Un filone particolarmente promettente è quello che mira a ridurre il volume dei dati trasmessi tramite procedure di compressione o approssimazione. Tra le tecniche più studiate troviamo la *sparsificazione*, che elimina le componenti meno rilevanti riducendo il numero di elementi effettivamente inviati; la *quantizzazione*, che riduce il numero di bit necessari per rappresentare i dati; e la *low-rank decomposition*, che approssima tensori o matrici con rappresentazioni più compatte. Sebbene tutte queste strategie offrano vantaggi in scenari distribuiti, in questo contesto ci concentreremo in particolare sulla compressione *lossy* (con perdita), che consente di ottenere compression-rate elevati a fronte di un errore controllato sui dati. La compressione lossy, combinata con algoritmi che sovrappongono fasi di comunicazione e calcolo, rappresenta un approccio particolarmente efficace per mitigare il memory wall. Bisogna però fare attenzione perché comprimere ha un costo in termini di tempo non trascurabile, soprattutto se tale operazione deve essere ripetuta più volte.

Immaginiamo lo scenario di un insieme di nodi, ognuno con il proprio vettore con risultati parziali di una computazione. Vogliamo che ogni nodo abbia il vettore che contiene la somma di tutti i vettori parziali. Supponiamo che questi vettori siano di grande dimensione quindi, per non congestionare la rete, si decide che ogni nodo invierà una porzione compressa del vettore al proprio vicino. Il vicino, seguendo la procedura ad anello, dovrà decomprimere il blocco, sommarlo al blocco locale corrispondente, ricomprimere e inviare il blocco somma al rispettivo vicino. Questo semplice esempio ci permette di comprendere i limiti della compressione che introduce un overhead non trascurabile, se non viene utilizzata nel giusto contesto. In aggiunta, molti algoritmi di compressione sono inerentemente sequenziali e non si adattano bene all'architettura della GPU. Nonostante questo, sono stati creati dei compressori (cuSZp, ZFP, nvCOMP [11, 14, 19]) che riescono a sfruttare al massimo la potenza di calcolo dell' hardware. Insieme a questi compressori sono state ideate una serie di librerie (C-coll, gZCCL [9, 6]) che integrano l'utilizzo dei compressori negli algoritmi di comunicazione e computazione collettiva (AllGather, Reduce-scatter e AllReduce).L'obiettivo è sfruttare la grande potenza di calcolo offerta dalle GPU per comprimere dati, riducendo al minimo l'overhead di compressione e decompressione.

A tale scopo si possono impiegare algoritmi "leggeri" che sfruttino l'hardware altamente parallelo, ridurre i trasferimenti tra memoria host/device e adottare tecniche di compressione omomorfica. Quest'ultima permette di comprimere una sola volta i dati e sommarli tra di loro mentre sono in forma compressa, il che si rivela utile se i dati compressi devono essere utilizzati per fare una computazione collettiva. In realtà quello che succede nelle implementazioni attuali non è proprio questo, ma qualcosa di più semplice: i dati vengono compressi e quando è necessario effettuarci un'operazione, vengono decompressi parzialmente, combinati con l'operazione decisa e poi nuovamente compressi. Quindi attualmente non è proprio un omomorfismo tra dati originali e dati compressi, ma più una decompressione parziale. hZCCL [8] è un' implementazione del compressore omomorfico e di alcune collettive su CPU. L'idea è quella di utilizzare le procedure presenti in hZCCL per la compressione omomorfica e farne un porting su GPU con l'ausilio di un compressore *GPU-centrico* (ovvero in grado di comprimere senza allocazioni sulla memoria host) come cuSZp2 [10].

Nella **prima parte** della trattazione vengono introdotti i concetti preliminari necessari per comprendere il contesto e le motivazioni del lavoro. In particolare, si

presentano le principali tecniche di compressione e le metriche utilizzate per valutarne l'efficacia. Successivamente, si illustra lo standard MPI [16], concentrandosi sulla sua implementazione in C, sugli algoritmi di comunicazione collettiva e sul modello di costo associato. Infine, si fornisce una panoramica sul modello di calcolo delle GPU, trattando i concetti fondamentali dell'API CUDA[17] e la terminologia propria della programmazione parallela su GPU.

La **seconda parte** approfondisce lo stato dell'arte dei compressori e delle collettive che integrano la compressione. Viene analizzata l'architettura generale di un compressore su GPU, con un focus particolare su due implementazioni, cuSZp e cuSZp2, che costituiranno il punto di partenza per lo sviluppo del compressore omomorfico. Inoltre, vengono presentati diversi framework che combinano compressione e comunicazioni collettive, introducendo il concetto di compressione omomorfica e le motivazioni alla base della sua adozione.

Nella **terza parte** si descrive il compressore omomorfico sviluppato durante il tirocinio e ne vengono analizzate le prestazioni. Viene illustrato il funzionamento del compressore su GPU e alcune strategie algoritmiche derivanti dalla sua integrazione nelle operazioni collettive. Infine, le prestazioni delle collettive e del compressore omomorfico vengono valutate sul supercomputer Leonardo.

Parte I

Nozioni preliminari

Capitolo 1

Cenni sulla compressione

La moderna disciplina della compressione ha l'obiettivo di rappresentare l'informazione in modo compatto, riducendo le dimensioni dei dati digitali rappresentati in codifica binaria. Prima di spiegare come funziona la compressione, è necessario capire il *perché funziona*. La chiave è comprendere la differenza tra *dati* e *informazione*: i *dati* incarnano la rappresentazione fisica dell'*informazione*. Dall'esperienza personale sappiamo che si possono dare rappresentazioni diverse della stessa informazione, alcune più lunghe, altre più concise. La compressione è possibile proprio perché spesso la rappresentazione originale non è la più corta, quindi si può sfruttare la ridondanza per diminuire la dimensione dei dati.

1.1 Compressione lossy e lossless

Quando si parla di algoritmo di compressione ci si riferisce ad una procedura che riceve l'input X e restituisce una rappresentazione X_c che richiede meno bit per essere codificata. L'algoritmo di decompressione riceve in input una rappresentazione compressa X_c e genera una ricostruzione Y dei dati originali. Se Y è identica ad X allora si avrà una ricostruzione dei dati senza perdita quindi la compressione è *lossless*, se invece Y è diversa da X allora la compressione è con perdita e quindi *lossy*. Se il contesto richiede che i dati siano ricostruiti con la massima precisione allora si sceglierà l'approccio *lossless*, altrimenti è possibile avere un po' di distorsione sui dati, in cambio di una compressione più efficace in termini di tempo o spazio, come nella compressione *lossy*.

La valutazione degli algoritmi di compressione si basa su due principali metriche: il *compression rate* e la misura della distorsione. Il compression rate fornisce una misura dell'efficacia della compressione. La misura della distorsione invece, quantifica la differenza tra dato originale e dato ricostruito. In altri termini è un indicatore della *qualità* o fedeltà della ricostruzione, quindi maggiore è la qualità, minore è la differenza tra X e Y. Quando la compressione è lossless, non occorre preoccuparsi della sua qualità, per definizione quest'ultima è senza perdita e quindi il dato ricostruito è identico all'originale. Come si può immaginare esiste un limite superiore al compression rate ottenibile senza alterare la ricostruzione del dato originale. Intuitivamente, il limite è dettato dalla quantità di informazione (in questo caso in bit) che il dato, o la sequenza di dati, contiene. Al contrario, per la compressione lossy, potendo accettare la perdita di informazione, si possono ottenere compression rate più alti.

1.2 Caratteristiche e prestazioni dei compressori

I compressori possono operare in modalità stream oppure a blocchi. Nel primo caso, i dati vengono forniti al compressore come un flusso continuo e processati fino alla sua interruzione, ad esempio alla fine di un file. La modalità a blocchi, invece, prevede che i dati siano letti e compressi in segmenti distinti, ciascuno dei quali viene elaborato separatamente. La dimensione dei blocchi, spesso definita dall'utente, può influire notevolmente sulle prestazioni complessive del compressore.

Per valutare l'efficacia di un compressore si utilizzano diverse metriche. Tra le principali vi sono il *compression ratio* e il *compression rate*. Indichiamo con D la dimensione dei dati originali e con C quella dei dati compressi; il rapporto di compressione è definito come

$$\text{Cmp-ratio} = \frac{C}{D},$$

mentre il *compression rate*, espresso in percentuale, rappresenta la riduzione dei dati e si calcola come

$$\text{Cmp-rate} = (1 - \text{Cmp-ratio}) \times 100.$$

Un'altra metrica utile è il *compression factor*, definito come

$$\text{Cmp-factor} = \frac{1}{\text{Cmp-ratio}}.$$

Oltre alla quantità di dati ridotti, è fondamentale valutare la distorsione introdotta dalla compressione. I metodi più comuni per confrontare una sequenza originale con la corrispondente sequenza ricostruita sono la misura dell'errore quadratico

$$d(x, y) = (x - y)^2$$

e quella della differenza assoluta

$$d(x, y) = |x - y|.$$

Quando la sequenza contiene molti elementi, è utile sintetizzare queste informazioni in misure aggregate. La più diffusa è l'*errore quadratico medio*, indicato con σ_d^2, calcolato come

$$\sigma_d^2 = \frac{1}{N} \sum_{n=1}^{N} (x_n - y_n)^2.$$

Altre metriche spesso utilizzate sono il *signal-to-noise ratio* (SNR), che confronta la potenza del segnale con quella del rumore, e il *peak signal-to-noise ratio* (PSNR), espresso in decibel, che mette in relazione il quadrato del segnale massimo x_{peak} con l'errore quadratico medio:

$$\text{SNR} = \frac{\sigma_x^2}{\sigma_d^2}, \qquad \text{PSNR} = \frac{x_{\text{peak}}^2}{\sigma_d^2},$$

dove σ_x^2 rappresenta la media al quadrato della sequenza originale $\{x_n\}$. Infine, in alcune applicazioni è utile considerare il massimo errore assoluto

$$d_\infty = \max_n |x_n - y_n|,$$

particolarmente quando la distorsione dei dati è accettabile finché non supera una soglia critica. Nei capitoli successivi verranno introdotte ulteriori metriche specifiche per valutare le prestazioni dei compressori nell'ambito delle implementazioni su GPU.

1.3 Quantizzazione degli scalari

Il processo di rappresentazione di un grande insieme, possibilmente infinito, di valori con un numero finito di rappresentati è definito come *quantizzazione*. Si consideri ad esempio una sorgente che genera numeri tra -10.0 e 10.0. Un semplice schema di quantizzazione potrebbe essere quello di trasformare ogni output della fonte nell'intero più vicino. Questo approccio riduce la dimensione dell'alfabeto per rappresentare l'output della sorgente; l'infinito numero di valori tra 10.0 e -10.0 viene ridotto ad un insieme contenente 21 valori. Nella pratica un *quantizzatore* è composto da due funzioni: la funzione dell'encoder e quella del decoder. L'encoder divide il range di valori che la sorgente può generare in un certo numero di intervalli. Ogni intervallo è rappresentato da una *codeword*, nel caso dell'esempio precedente la codeword è un intero in $[-10, 10]$. L'encoder rappresenta gli output della sorgente che cadono in uno di questi intervalli con la codeword assegnata a quell'intervallo. Dato che tra due interi esistono infiniti numeri razionali, non avremo più modo di recuperare il valore originale una volta quantizzato, per questo la funzione di encoding è **irreversibile**. Per ogni codeword generata dall'encoder, il decoder genera un valore di ricostruzione. Poiché una codeword rappresenta un intero intervallo e non c'è modo di sapere quale valore nell'intervallo è stato effettivamente generato dalla sorgente, il decoder sceglie un valore che, in un certo senso, rappresenta al meglio tutti i valori dell'intervallo. Ad esempio si può scegliere la metà dell'intervallo come rappresentante, quindi se Q è la funzione dell'encoder e Q' è la funzione del decoder, $Q'(Q(0.6)) = Q'(110) = 0.5$. La parte di costruzione degli intervalli è parte dell'encoder, mentre la ricostruzione e la scelta dei valori rappresentati fa parte del design del decoder. La coppia *encoder* e *decoder* prende il nome di **quantizzatore**.

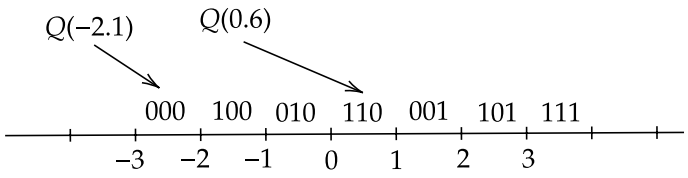


Figura 1.1. Quantizzazione scalare con funzione Q che mappa ogni razionale ad una codeword che rappresenta l'intervallo dei due interi tra cui è compreso.

In generale se vogliamo quantizzare una sorgente con M intervalli, allora sceglieremo $M + 1$ punti agli estremi degli intervalli ed M rappresentanti, uno per ogni intervallo. I punti agli estremi vengono chiamati *decision boundaries*, mentre i valori rappresentanti sono chiamati *recostruction levels*.
Siano $\{b_i\}_{i=0}^{M}$ i *decision boundaries*, $\{y_i\}_{i=1}^{M}$ i *recostruction levels* e sia $Q(\cdot)$ la funzione di quantizzazione. Allora

$$Q(x) = y_i \text{ se e solo se } b_{i-1} \leq y_i < b_i \tag{1.1}$$

La differenza tra x e $Q(x)$ viene chiamata errore di quantizzazione o *distorsione di quantizazzione.* Nella **Figura 2.1** ciascun intervallo viene rappresentato tramite una codifica binaria a lunghezza fissa (**fixed-length encoding**) quindi la dimensione di una codeword è data da $\lceil \log_2 M \rceil$ bit.

Quantizzazione dei float :

Si consideri $f \in$ `float`, ed un limite sull'errore $e \in$ `float`. Si può ottenere un $r \in$ `int`, che è una quantizzazione di f che rispetta il limite sull'errore e :

$$r = \left\lfloor \frac{f}{2e} \right\rfloor \tag{1.2}$$

garantendo che $|r \times 2e - f| \leq e$. Nella fase di decoding si applica la trasformazione inversa, ovvero $f' = r \times 2e$.

Ad esempio, preso $f =$ `42.683` ed $e =$ `0.01`, allora $r = \lfloor$`42.683/0.02`$\rfloor =$ `2134`, quindi $f' =$ `2134` $\times$ `0.02` $=$ `42.68` e $f' -$ `42.683` $=$ `0.003` < 0.01. Data una sequenza di `float` $\{f_1, f_2, \ldots, f_m\}$, si può convertire in una sequenza di interi tramite la procedura descritta sopra e poi trasformare facilmente ogni intero nella sua codifica binaria.

1.4 Linear prediction

Solitamente all'interno dei dataset scientifici i valori dei *data point* sono molto simili quando questi ultimi sono adiacenti nella struttura dati in cui sono memorizzati. Una semplice sottrazione tra dati adiacenti produce una **piccola differenza**, che è molto meno costosa da memorizzare rispetto ai dati originali. In generale si può assumere che un dato sia correlato a p dei suoi predecessori e quindi il valore predetto, indicato con $\hat{s}_k$, sarà

$$\hat{s}_k = \sum_{i=1}^{p} a_i s_{k-i}. \tag{1.3}$$

Bisogna notare che solo i dati che precedono s_k possono contribuire ad $\hat{s}_k$ e non i suoi successori. Questo perché in un processo di decompressione non si conoscono ancora i valori originali dei successori. Se la procedura di *linear prediction* (**eq 2.7**) è eseguita bene, quindi con il giusto numero di predecessori che contribuiscono alla somma, allora si otterrà una differenza $e_k = s_k - \hat{s}_k$ quasi sempre piccola. Lo *zero-predittore* semplicemente predice ogni s_k come zero. Un predittore del primo ordine prevede ogni s_k come uguale al suo predecessore s_{k-1}, quindi

$$\hat{s}_k = s_k - s_{k-1}. \tag{1.4}$$

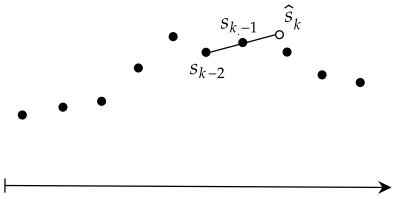


Figura 1.2. Predittore del secondo ordine.

Un predittore del secondo ordine calcola il segmento (polinomio di grado 1) passante per s_{k-1}, s_{k-2} e lo continua per prevedere $\hat{s}_k$ (*estrapolazione*). Quello di terzo grado calcola una sezione di conica (polinomio di secondo grado) passante per tre predecessori. In generale quando si costruisce un n-predittore si calcola il polinomio di grado $n-1$ passante per gli n punti tra s_{k-1} e s_{k-n}, e si fa un'estrapolazione per prevedere il nuovo valore.

Capitolo 2

MPI: Message Passing Interface

> «MPI è una specifica di **interfaccia** di libreria di ***message-passing***. MPI si occupa principalmente del modello di programmazione parallela message-passing, in cui i dati vengono spostati dallo spazio degli indirizzi di un processo a quello di un altro processo attraverso operazioni cooperative su ciascun processo. Le estensioni al modello di message-passing "classico" sono fornite nelle operazioni collettive, nelle operazioni di accesso alla memoria remota, nella creazione dinamica di processi e nell'I/O parallelo. MPI è una specifica, non un'implementazione; esistono diverse implementazioni di MPI. Questa specifica riguarda un'interfaccia di libreria; MPI non è un linguaggio e tutte le operazioni MPI sono espresse come funzioni, subroutine o metodi, secondo i binding linguistici appropriati che, per C e Fortran, fanno parte dello standard MPI.»
>
> –Standard MPI-4.1 (November 2, 2023)

Nei programmi di *message-passing*, un programma che esegue su una coppia *cpu-memoria* viene chiamato **processo**, e due processi possono comunicare chiamando funzioni: un processo chiama una funzione di invio (*send*) e l'altro una funzione di ricezione (*receive*). Lo scambio di messaggi descritto sopra viene chiamata comunicazione *point-to-point*. Ma esistono anche funzioni di comunicazione globale che possono coinvolgere più di un processo. Queste funzioni vengono chiamate *comunicazioni collettive*.

2.1 Memoria distribuita e condivisa

Michael Flynn introdusse la *tassonomia* delle architetture dei computer nel 1966, dove i calcolatori erano classificati in base a quanti dati potevano processare in modo concorrente e quante istruzioni differenti potevano eseguire allo stesso tempo. Col passare degli anni questa classificazione è stata raffinata aggiungendo sottocategorie, specialmente alle macchine **MIMD** (multiple instruction, multiple data), che sono state divise in due categorie :

- **Shared-memory MIMD**: architettura di macchina con uno spazio di memoria condivisa universalmente accessibile. La memoria condivisa semplifica tutte le transazioni che devono avvenire tra le CPU con una quantità minima di overhead, ma costituisce anche un collo di bottiglia che limita la scalabilità del sistema. Una soluzione per questo problema è la partizione della memoria tra le CPU, in modo che ogni CPU "possieda" una parte della memoria. In

questo modo una CPU ottiene un accesso più veloce alla sua memoria locale, ma può comunque accedere, anche se più lentamente, alla memoria non locale appartenente ad altre CPU. Il partizionamento non influisce sullo schema di indirizzamento, che è universale. Questa tecnica di partizionamento è nota come accesso non uniforme alla memoria (NUMA) e consente alle macchine a memoria condivisa di scalare fino a poche decine di CPU.

- **Distributed-memory MIMD**: una macchina composta da processori che comunicano scambiandosi messaggi. Il costo della comunicazione è elevato, ma poiché non c'è un singolo mezzo di comunicazione da contendere, tali macchine possono scalare senza alcun limite pratico, a parte i limiti di spazio e di energia.

La maggior parte dei sistemi MIMD di grandi dimensioni sono sistemi ibridi in cui un numero di sistemi a memoria condivisa relativamente piccolo è collegato da una rete di interconnessione. In questi sistemi, i singoli sistemi a memoria condivisa sono chiamati **nodi**. Alcuni sistemi MIMD sono sistemi **eterogenei**, in cui i processori hanno capacità diverse, ad esempio un sistema con una CPU e una GPU è un sistema eterogeneo.

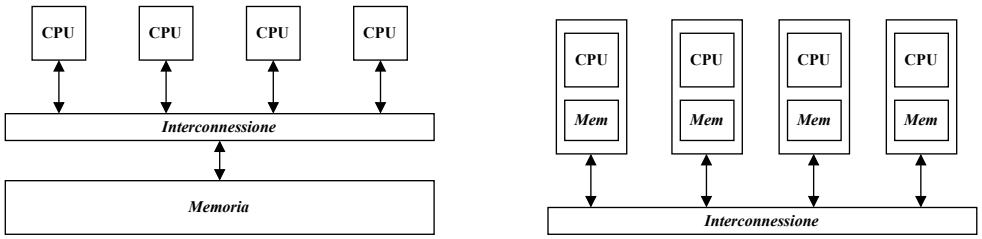


Figura 2.1. Rappresentazione schematica di un sistema a memoria condivisa (sinistra) e a memoria distribuita (destra).

2.2 Concetti chiave di MPI

Nei sistemi a memoria distribuita non è possibile comunicare tramite la memoria condivisa. Pertanto, i messaggi vengono utilizzati per coordinare attività parallele che eventualmente vengono eseguite su processori distribuiti ma interconnessi. L'astrazione che MPI presenta è proprio quella di processi che possono scambiarsi messaggi l'un l'altro, semplicemente specificando l'altro capo dell'operazione di comunicazione. I processi possono essere distribuiti con la stessa facilità su una singola macchina o su una rete di macchine eterogenee senza dover modificare o ricompilare il codice. La libreria MPI e l'ambiente di runtime sono responsabili per:

- **Identificazione dei processi**: ad ogni processo viene assegnato univocamente un numero intero non negativo chiamato ***rank***. Questo numero di identificazione è assegnato nel contesto di un gruppo di processi che prende il nome di **comunicatore**. Per impostazione predefinita, tutti i processi di un programma MPI appartengono al comunicatore globale identificato dalla costante simbolica `MPI_COMM_WORLD`. Quando si parla di dimensione del comunicatore ci si riferisce al numero di processi che appartengono al comunicatore.

- **Instradamento dei messaggi**: MPI si occupa di consegnare in modo efficiente i messaggi alle loro destinazioni utilizzando i socket o RDMA (per

le comunicazioni intermacchina) o i buffer condivisi (per le comunicazioni intramacchina).

- **Buffering dei messaggi**: Una volta che un messaggio è stato spedito dal mittente, viene tipicamente bufferizzato da MPI durante il tragitto verso la sua destinazione. Più messaggi possono attendere la raccolta nello spazio buffer MPI di un destinatario. MPI fornisce meccanismi per eliminare i compiti di buffering all'applicazione. Questo può essere utile per gestire messaggi molto grandi.
- **Marshaling dei dati**: MPI permette lo scambio di messaggi anche su architetture eterogenee (ad esempio, big-endian e small-endian) convertendo le rappresentazioni dei dati per adattarle alla destinazione del messaggio. Questo è il motivo per cui le chiamate MPI richiedono la dichiarazione esplicita dei tipi di dati da comunicare.

2.3 Comunicazioni point-to-point

MPI_SEND

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
        int dest, int tag, MPI_Comm comm)
```

L'operazione invia il contenuto di *buf*, di cui è stata specificata la dimensione (*count*) e il tipo (*dtype*), al processo con rank *dest* all'interno dello stesso comunicatore (*comm*). Sia *sender* il rank del processo del mittente, quindi il messaggio viaggerà da *sender* a *dest*. Il *tag* potrebbe essere utile al destinatario per selezionare solo i messaggi a cui è interessato. Il valore di ritorno è `MPI_Success` se l'operazione di invio si è conclusa con successo.

MPI_RECV

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source,int tag,
             MPI_Comm comm, MPI_Status *status)
```

In questo caso il destinatario riceverà il messaggio in *buf* dal processo con rank *source*. L'ultimo parametro, lo *status* è utilizzato per ritornare delle informazioni sul messaggio appena ricevuto (e.g. dimensione effettiva del messaggio ricevuto). Ovviamente per ricevere correttamente i messaggi il destinatario dovrà scegliere un valore

$$\text{count}_{\text{recv}} \geq \text{count}_{\text{send}} \quad \text{e} \quad \text{dtype}_{\text{recv}} = \text{dtype}_{\text{send}}.$$

Il parametro *source* può essere un numero intero non negativo, nel qual caso identifica il processo, oppure può essere impostato sulla costante simbolica `MPI_ANY_SOURCE` (una rapida ricerca nell'header mpi.h rivela che è uguale a -1). In quest'ultimo caso qualsiasi processo del comunicatore specificato può essere l'origine del messaggio.

Altre varianti di Send e Receive

Nello standard MPI, la `MPI_Send` è chiamata operazione di **send bloccante**, ovvero che blocca il mittente finché l'operazione di invio non è completata. Tuttavia, questo è fuorviante, perché la funzione può ritornare il controllo prima che il

messaggio venga consegnato. Questo succede perché in realtà la `MPI_Send` esegue la cosiddetta ***standard communication mode***. Ciò che accade è che MPI decide, in base alle dimensioni del messaggio, se bloccare la chiamata finché il processo di destinazione non la riceve o se ritornare prima che venga fornita una ricezione. Quest'ultima opzione viene scelta se il messaggio è abbastanza piccolo, rendendo `MPI_Send` localmente bloccante, cioè la funzione ritorna non appena il messaggio viene copiato in un buffer MPI locale, aumentando l'utilizzo della CPU. La copia è necessaria per liberare il buffer utilizzato dal processo sorgente per le operazioni successive, poiché con questa forma di invio non c'è modo per il processo mittente di sapere quando il messaggio viene consegnato.

Esistono tre ulteriori modalità di comunicazione:

- **Bufferizzata**: l'operazione di send è *localmente bloccante*, ovvero ritorna il controllo appena il messaggio è copiato sul buffer, ma il buffer in questo caso è fornito dall'utente.
- **Sincrona**: in modalità sincrona, l'operazione di invio ritorna solo dopo che il processo destinatario ha iniziato (ma non per forza completato) la ricezione.
- **Ready**: L'operazione di invio ha successo solo se esiste una operazione di ricezione già iniziate che gli corrisponde. In caso contrario la funzione ritorna una codice di errore.

Comunicazioni non-bloccanti

Solitamente le send bufferizzate non sono utilizzate nella pratica perché necessitano di una copia aggiuntiva della memoria. Le performance possono essere migliorate se si evitano le copie o se dopo la chiamata di una send l'esecuzione continua immediatamente senza tenere conto dello stato della comunicazione. Le funzioni che implementano questo comportamento sono chiamate *immediate* o non bloccanti, e risultano utili nei casi in cui si può rafforzare la concorrenza sovrapponendo fasi comunicazione e computazione.

Le funzioni non bloccanti possono essere utilizzate sia lato mittente (`MPI_Isend`), sia lato destinatario (`MPI_Irecv`). Ovviamente, una volta effettuate le due chiamate, essendo che l'esecuzione non viene bloccata fino al completamento dell'operazione, né il mittente né il destinatario conoscono lo stato della comunicazione.

Per aggirare il problema, entrambi i processi devono interrogare l'ambiente MPI riguardo lo stato dell'azione che hanno inizializzato. A questo fine le due chiamate di libreria hanno un parametro aggiuntivo, `MPI_Request *req`, che a sua volta viene utilizzato dalle funzioni `MPI_Wait` (bloccante) e `MPI_Test` (non-bloccante) per controllare lo stato della comunicazione.

2.4 Comunicazioni collettive

Con il termine comunicazioni collettive si fa riferimento alle operazioni che coinvolgono più di due nodi. Uno degli argomenti chiave di una chiamata a una comunicazione collettiva è un comunicatore che definisce il gruppo o i gruppi di processi MPI partecipanti e fornisce un contesto per l'operazione. Molte delle chiamate a comunicazioni collettive necessitano di un solo nodo che trasmette, ad esempio una broadcast, o di un solo nodo che riceve da tutti gli altri. Questo tipo di processo MPI è chiamato ***root***. Bisogna tenere a mente che le comunicazioni collettive si differenziano in diversi modi da quelle point-to-point:

- Tutti i processi nello stesso comunicatore devono chiamare la stessa funzione collettiva.
- Gli argomenti passati alla funzione devono essere compatibili. Ovvero il tipo di dato, il tipo di operazione e, per quelle che lo richiedono, il processo *root* devono essere inseriti in modo consistente tra tutti i processi che partecipano alla collettiva.
- Le chiamate non vengono associate in base ai tag, ma viene utilizzato esclusivamente l'ordine di chiamata e il comunicatore in cui vengono inizializzate.
- Anche se in alcune collettive un processo non riceve nulla (o non trasmette nulla), è necessario che venga comunque fornito il puntatore ad un buffer di output (input), che potrebbe essere `NULL`.

MPI fornisce una grande varietà di comunicazioni collettive che possono essere distinte in tre gruppi :

- **one-to-all**: `MPI_Bcast`, `MPI_Scatter`, `MPI_Scatterv`,
- **all-to-one**: `MPI_Gather`, `MPI_Gatherv` , `MPI_Reduce`,
- **all-to-all**: `MPI_Allreduce`, `MPI_Alltoall`, `MPI_Allgather`, `MPI_Barrier`.

MPI_SCATTER

```
int MPI_Scatter(const void *sendbuf, int sendcount,
     MPI_Datatype sendtype, void *recvbuf, int recvcount,
     MPI_Datatype recvtype, int root, MPI_Comm comm)
```

La *scatter* divide il dataset presente nel *sendbuf* del processo *root* in parti uguali (*sendcount*) e lo distribuisce a tutti gli altri processi del comunicatore. Quindi i destinatari riceveranno la loro porzione in *recvbuf*. `MPI_Scatterv` è una versione alternativa dello scatter che permette di specificare quanti dati inviare a ogni processo.

MPI_REDUCE

```
int MPI_Reduce(const void *sendbuf, void *recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm)
```

`MPI_REDUCE` combina gli elementi forniti in buffer di ingresso di ogni processo MPI del gruppo, utilizzando l'operazione *op*, e restituisce il valore combinato nel buffer di uscita del processo MPI con rank root. Il buffer di ingresso è definito dagli argomenti *sendbuf, count* e *datatype*; il buffer di uscita è definito da argomenti *recvbuf, count* e *datatype*; entrambi hanno lo stesso numero di elementi, con lo stesso tipo. La ***reduce***,insieme alla sua versione *all-to-all* (la *allreduce*), è un' operazione che viene spesso utilizzata per aggregare i risultati nelle iterazioni delle applicazioni che processano grandi quantità di dati.

MPI_REDUCE_SCATTER

```
int MPI_Reduce_scatter_block(const void *sendbuf, void
*recvbuf, int recvcount,MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm)
```

`MPI_Reduce_scatter_block` esegue prima una riduzione globale, element-wise, su vettori di elementi *count = n*recvcount* nei buffer di invio definiti da *sendbuf*, *count* e *datatype*, usando l'operazione *op*, dove n è il numero di processi MPI nel gruppo di comm. Il vettore risultante viene trattato come n blocchi consecutivi di elementi *recvcount* che vengono sparsi ai processi MPI del gruppo. Il blocco i-esimo viene inviato al processo MPI i e memorizzato nel buffer di ricezione definito da *recvbuf*. La chiamata `MPI_Reduce_Scatter` estende la funzionalità di `MPI_Reduce_Scatter_Block` in modo che i blocchi sparsi possano variare di dimensione.

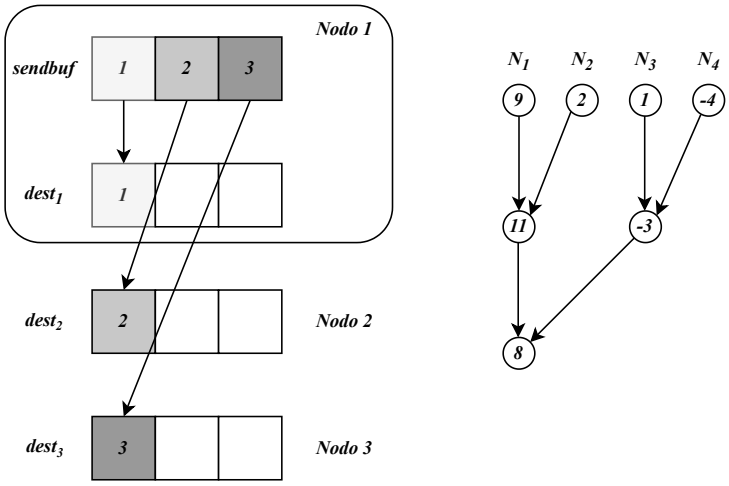


Figura 2.2. Una rappresentazione di come funziona MPI_Scatter, e MPI_Reduce con somma globale ad albero e root N_1.

MPI_GATHER

```
int MPI_Gather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype,void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

L'operazione di *gathering* è l'esatto opposto dell'operazione di *scattering*: i dati (*sendbuf*) di tutti i processi del comunicatore vengono raccolti nel *recvbuf* da parte del *root*. La radice riceve i messaggi e li memorizza in ordine. È come se gli n messaggi inviati dai processi del gruppo (compresa la radice stessa), fossero concatenati in ordine di rank e il messaggio risultante fosse ricevuto dal processo root come se fosse una chiamata a

`MPI_Recv(recvbuf, recvcount*n, recvtype, ...)`.

MPI_ALLGATHER

```
int MPI_Allgather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm)
```

La `MPI_Allgather` può essere pensata come una *gather* in cui tutti i processi ricevono il risultato, non solo il processo root. Il blocco di dati inviato dal j-esimo processo MPI viene ricevuto da ogni processo MPI e collocato nel j-esimo blocco del buffer *recvbuf*.

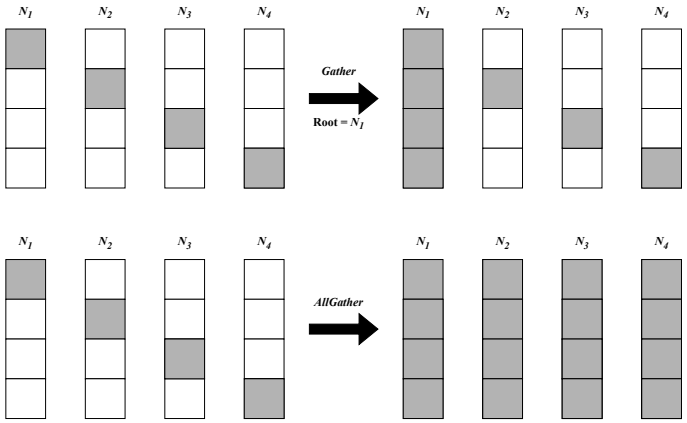


Figura 2.3. Rappresentazione di una MPI_Gather e MPI_Allgather con 4 processi.

MPI_ALLREDUCE

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf,
                  int count,MPI_Datatype datatype, MPI_Op op,
                  MPI_Comm comm)
```

La `MPI_Allreduce` si comporta proprio come una `MPI_Reduce`, ma il risultato viene ricevuto completo da tutti i processi che partecipano alla chiamata.

2.5 Algoritmi di Allreduce

La collettiva di *allreduce* viene ampiamente utilizzata nelle applicazioni reali di simulazioni scientifiche e ML, è quindi importante che le prestazioni siano elevate. Inoltre, nei capitoli successivi verranno forniti dei dettagli su alcune implementazioni della collettiva, integrata con tecniche di compressione viste in precedenza. Per questo motivo è necessario analizzare più nel dettaglio come funziona e quali algoritmi vengono utilizzati nelle implementazioni. La *allreduce* in realtà è l'unione di due altre collettive viste sopra : *reduce-scatter* e *allgather*. Da questa osservazione nasce una possibile idea algoritmica di utilizzare inizialmente una *reduce-scatter*, in modo da distribuire a ciascun processo una porzione dell'array contenente la somma globale. Successivamente, mediante una *allgather*, ogni processo condividerà il proprio blocco

con gli altri, permettendo così a tutti di ricostruire localmente l'intero array con la somma globale. Un'altra idea è quella di comporre una *reduce* con una *broadcast*. Questa idea può essere efficiente se i messaggi inviati sono di piccole dimensioni, ma soffre di un collo di bottiglia nei processi *root*.

Per stimare il costo degli algoritmi di comunicazione collettiva e confrontarli tra loro utilizziamo un semplice modello [27] basato su **latenza** e utilizzo della **larghezza di banda**. Assumiamo che il tempo necessario per inviare un messaggio tra due nodi può essere modellato come $\alpha + n\beta$, dove α è la latenza di ciascun messaggio, indipendente dalla sua dimensione, β è il tempo di trasferimento per byte, e n è il numero di byte trasferiti. Si assume inoltre che il tempo impiegato sia indipendente dal numero di coppie di processi che comunicano tra loro, indipendente dalla distanza tra i nodi comunicanti e che i collegamenti di comunicazione siano bidirezionali. Si ipotizza che l'interfaccia di rete del nodo sia a porta singola; cioè, al massimo un messaggio può essere inviato e un messaggio può essere ricevuto contemporaneamente. Nel caso delle operazioni di riduzione, si assume che γ sia il costo di calcolo per byte per l'esecuzione dell'operazione di riduzione a livello locale su un qualsiasi processo. Nella seguente analisi assumeremo ci siano p processi che partecipano alla collettiva.

2.5.1 Ring

Nell'algortimo ad anello i dati sono trasmessi lungo un anello virtuale di processi. Supponiamo che i processi siano identificati da rank che vanno da 0 a $p-1$ e, nel caso di una ***allgather***, che ogni processo possieda $\frac{n}{p}$ byte. Al primo step dell'algoritmo il processo con rank i invia i suoi $\frac{n}{p}$ byte al rank $i+1$. Allo stesso tempo il rank i riceve $\frac{n}{p}$ byte dal processo $i-1$. Quindi allo step successivo i copia i dati ricevuti da $i-1$ allo step precedente e li inoltra al vicino ovvero il rank $i+1$. E come allo step prima riceve altri $\frac{n}{p}$ byte dal rank $i-1$, che a sua volta li aveva ricevuti dal vicino a sinistra $(i-2)$ al primo step. Si continua con questo processo di scambio finché tutti non hanno una copia locale di tutti i byte di tutti gli altri processi, ovvero n byte. Considerando la latenza α e la bandwidth β, il tempo richiesto dall'algoritmo è

$$t_{\text{ring}} = (p-1)\alpha + \frac{p-1}{p} n\beta \tag{2.1}$$

Nel caso di un' operazione collettiva come la reduce scatter bisogna tenere conto anche del tempo di computazione γ, quindi $t'_{\text{ring}} = (p-1)\alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$. Il tempo totale per una collettiva di *allreduce* è dato dalla somma dell'operazione di *reduce-scatter* e quella di *allgather*:

$$T_{\text{ring}} = t'_{\text{ring}} + t_{\text{ring}} = 2 \times \left\{ (p-1)\alpha + \frac{p-1}{p} n\beta \right\} + \frac{p-1}{p} n\gamma \tag{2.2}$$

La possibilità di sovrapporre le fasi di calcolo a quelle di comunicazione, unitamente al fatto che il costo associato a γ risulta spesso inferiore rispetto a quelli di α e β, consente, in determinate circostanze, di considerarne trascurabile l'impatto.

2.5.2 Recursive doubling

In MPICH, una delle principali implementazioni di MPI che viene utilizzato come base di molte applicazioni commerciali, il *recursive doubling* con scambio a coppie è

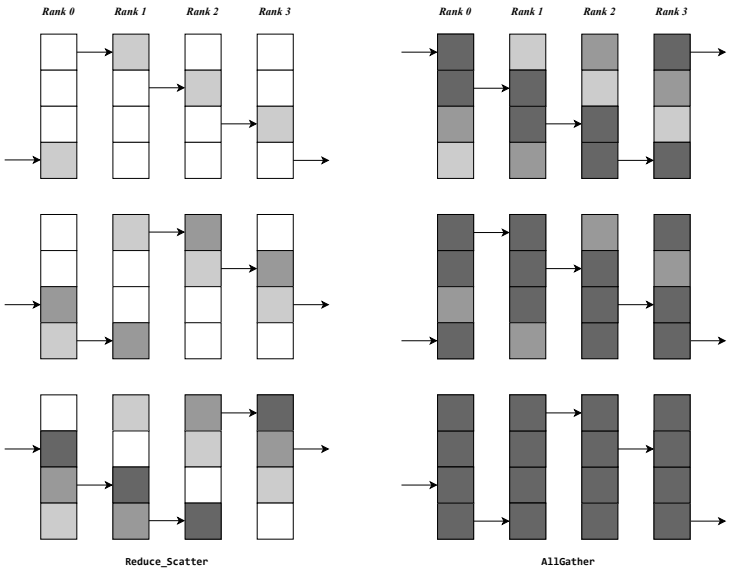


Figura 2.4. Algoritmo di ring *allreduce* implementato tramite una *reduce-scatter* seguita da una *allgather*.

utilizzato per la *allreduce* quando è soddisfatta una delle seguenti condizioni: se i messaggi sono di piccole dimensioni, inferiori a 2KB; se viene utilizzata un'operazione definita dall'utente per ridurre il vettore durante ogni fase, in modo che gli operatori non associativi non vengano erroneamente ottimizzati con altri algoritmi; se il numero di elementi del vettore è minore della potenza di due inferiore che più si avvicina al numero di processi [1]. Ogni passo dell'algoritmo di *recursive doubling* consiste nello scambio di messaggi tra coppie di processi, con una conseguente combinazione locale dei risultati. Il recursive doubling richiede che il numero di processi che partecipano alla chiamata sia una potenza di due, poiché ogni fase suddivide il gruppo per due. Con un grande numero di processi , le potenze di due sono scarse, quindi questa è una forte limitazione all'utilizzabilità del metodo di base. MPICH risolve questo problema, aggiungendo uno stadio iniziale, in cui il gruppo di processi viene ristretto ad una potenza di due, e uno stadio finale in cui il gruppo viene riespanso. Per prima cosa viene trovata la potenza di due inferiore più vicina al numero di processi p, ovvero $q = 2^{\lfloor \log_2 p \rfloor}$, e sia $r = p - q$ il resto dei processi. Il valore q è il numero di processi che verranno utilizzati nel *recursive doubling*. Tutti i rank i, dove i è pari e $i < 2r$, inviano al loro vicino di rank $i + 1$. Successivamente solo i processi di rank dispari parteciperanno alla collettiva, i pari rimarranno in attesa. Su i processi attivi rimanenti verrà eseguito uno scambio a coppie, in cui il rank i, alla j-esima iterazione, scambia con il rank $i + 2^{j-1}$ e in seguito fa localmente una combinazione dei risultati. Infine i rank che erano rimasti inattivi durante il recursive doubling ricevono dai loro diretti vicini i dati aggiornati globalmente. Utilizzando lo stesso modello dell'algoritmo della *ring allreduce* si ottiene

$$T_{\text{rec\_doub}} = (\alpha + n\beta + n\gamma) \times \log_2 p \quad \text{con } p = 2^k \text{ per qualche } k \in N. \tag{2.3}$$

Se p non è una potenza di due, bisogna solo aggiungere al termine logaritmico le due fasi di restrizione ed espansione del gruppo .

[1] Stiamo semplicemente considerando il caso in cui $n < 2^{\lceil \log_2 p \rceil}$.

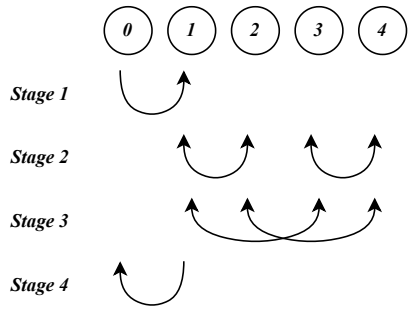


Figura 2.5. Allreduce utilizzando il *recursive doubling* di MPICH per le non-potenze di due.

2.5.3 Altre implementazioni della Allreduce

Per comprendere i successivi capitoli sarà sufficiente conoscere i due algoritmi descritti in precedenza, che coprono sia l'esigenza di inviare messaggi brevi (*recursive doubling*), sia quella di inviare messaggi più grandi (*ring*). Nonostante questo, esistono algoritmi più efficienti, in certi casi, dei due descritti in precedenza. L'algoritmo ideato da Rabenseifner [21], ad esempio, esegue in $\log_2 p$ passi e allo stesso tempo è ideato per messaggi di grandi dimensioni. I principi alla base dell'algoritmo sono :

- **Recursive vector halving/doubling**: per la riduzione dei vettori lunghi, il vettore può essere diviso in due parti e una metà viene ridotta dal processo stesso, mentre l'altra metà viene inviata a un processo vicino per la riduzione. Nel passaggio successivo, ancora una volta i buffer vengono dimezzati, e così via. Nel caso del *vector doubling* il funzionamento è opposto, quindi il vettore locale e quello ricevuto vengono uniti e inviati all'iterazione successiva.

- **Recursive distance halving/doubling**: alla j-esima iterazione il processo con rank i scambia dati con il processo con rank $i + 2^{j-1}$, con $2^{j-1} \leq p/2$, nel caso di *distance doubling*, e rank $i + p/2^j$ fino a quando $p/2^j = 1$.

Come per il *recursive doubling*, è richiesto un numero di processi che sia una potenza di due, altrimenti sono necessari degli scambi di messaggi aggiuntivi.
Correzione del numero di processi (se $p \neq 2^k$): si ricavano $q = 2^{\lfloor \log_2 p \rfloor}$ ed $r = p - q$; i processi con rank $\leq 2r$ dividono a metà il vettore locale. Quelli con rank pari inviano la seconda metà ai processi con rank dispari, quelli con rank dispari inviano la prima metà ai pari. Entrambi fanno una *reduce* locale dei dati ricevuti, successivamente i processi dispari inviano la seconda metà a quelli pari e restano in attesa fino alla fine della collettiva. Quindi i primi r processi pari e gli ultimi $p - 2r$ vengono rinumerati da 0 a $q - 1$.
Reduce-scatter: Nella prima fase di *recursive vector halving* e *distance doubling* , i processi con rank i' pari(dispari) inviano la seconda(prima) metà del loro buffer al processo con rank $i' + 1(i' - 1)$. Quindi si calcola la riduzione tra il buffer locale e il buffer ricevuto. Nei successivi $\log_2 q - 1$ passi, i buffer vengono ricorsivamente dimezzati e la distanza raddoppiata. Ora, ciascuno dei q processi possiede $1/q$ del vettore totale dei risultati della riduzione. Nel complesso questa sezione ha costo $(\log q)\alpha + \frac{q-1}{q} \times (n\beta + n\gamma)$.
Allgather: Nella fase di raccolta dei risultati bisognerà utilizzare il protocollo

inverso, quindi *recursive vector doubling* e *distance halving*. Al primo passo le coppie di processi a distanza $q/2$ si scambiano una parte del buffer pari a $1/q$ per ottenere $2/q$ del vettore risultato e al passo successivo lo scambio a distanza $q/4$ con $2/q$ del vettore risultato per ottenere i $4/q$. Dopo ogni fase di scambio di comunicazioni, il buffer dei risultati viene raddoppiato e dopo $\log_2 q$ passi, i q processi hanno ricevuto il risultato totale della riduzione. Questa parte di raccolta costa $(\log_2 q)\alpha + \frac{q-1}{q} \times n\beta$. Se $p \neq q$ allora il vettore totale deve essere inviato ai r processi rimasti idle, richiedendo un costo aggiuntivo di $\alpha + n\beta$.

Capitolo 3

Calcolo parallelo su GPU con CUDA

Nell'ultimo decennio, il calcolo ad alte prestazioni si è evoluto in modo significativo, in particolare grazie all'emergere di architetture eterogenee GPU-CPU, che hanno portato a un fondamentale cambio di paradigma nella programmazione parallela. La CPU è stata pensata e ottimizzata per eseguire programmi sequenziali. Utilizza unità di controllo sofisticate per consentire alle istruzioni di un singolo thread di essere eseguite in parallelo o addirittura al di fuori del loro ordine sequenziale, mantenendo l'aspetto dell'esecuzione sequenziale. Ancora più importante è la presenza di grandi memorie cache, per ridurre le latenze di accesso alle istruzioni e ai dati di applicazioni complesse di grandi dimensioni. Al contrario, le GPU dedicano gran parte dello spazio su chip per le unità di calcolo in modo da massimizzare il throughput di operazioni in floating-point. Quindi, programmi inerentemente sequenziali non risultano adatti all'esecuzione su GPU. Per questo motivo l'utilizzo di CPU e GPU in modo coordinato permette ai sistemi di calcolo distribuito di eccellere in ogni possibile scenario. **CUDA** è una piattaforma di elaborazione in parallelo e un modello di programmazione sviluppato da NVIDIA per l'elaborazione generale su GPU. Con CUDA, gli sviluppatori sono in grado di accelerare notevolmente le applicazioni di calcolo sfruttando la potenza delle GPU.

3.1 Modello di esecuzione

Nella trattazione che segue con il termine **Host** indichiamo la CPU e la sua memoria, e con **Device** la GPU e la sua memoria e faremo riferimento esclusivamente alla runtime API di CUDA.

Dato che la GPU possiede una grande quantità di unità di calcolo, può essere utilizzata per accelerare alcune parti di programma. Per fare un uso intelligente della GPU, possiamo decomporre il nostro problema in un grande numero di *task* che possono essere eseguiti in modo concorrente. Tramite CUDA vengono creati dei gruppi di thread, gestiti dagli scheduler della GPU in modo da eseguire i task. I thread creati sono organizzati in una struttura a 6 dimensioni (ma può avere anche dimensioni minori). Questa struttura ha una forma gerarchica, ed ogni thread è in grado di ricavare la propria posizione al suo interno. La componente al livello più basso è costituita dai **blocchi** che possono essere di una, due o tre dimensioni e contengono al loro interno i thread. A loro volta i blocchi sono organizzati in una **griglia** che può variare anch'essa tra una, due e tre dimensioni. Questa scelta nell'organizzazione della struttura è dovuta in parte all'hardware su cui

viene implementata e d'altra parte si rivela un'astrazione che può essere utile nella risoluzione di determinati problemi.

Nvidia utilizza la **compute capability** (CC) per codificare le caratteristiche e le capacità dell'hardware. Ad esempio, la CC di un'architettura ci dice quanto può essere grande un blocco o griglia nelle sue varie dimensioni.

Dopo aver determinato la struttura, ci sarà una funzione che verrà eseguita da tutti i thread, che nella terminologia di CUDA è chiamata **kernel**. Uno dei parametri della funzione è proprio la configurazione di esecuzione dei thread (la struttura formata da blocchi organizzati in una griglia). L'esecuzione del kernel è asincrona, ovvero: la funzione viene eseguita sul *device*, quindi una volta chiamata dall'host gli ritorna direttamente il controllo dell'esecuzione. Eventualmente sarà possibile sincronizzare device e host tramite apposite funzioni.

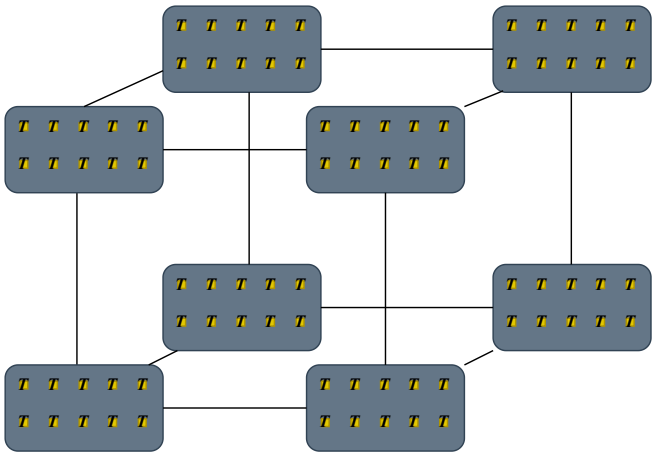


Figura 3.1. Configurazione di esecuzione dei thread con griglia in 3 dimensioni e blocchi in due dimesioni.

Quando un kernel esegue su una GPU, in realtà sta eseguendo istruzioni in modo sincrono su tanti **Streaming Processor** (SP). Un insieme di SP che eseguono sotto il controllo di una stessa control unit vengono chiamati **Streaming Multiprocessor (SM)**. Una GPU contiene diversi SM, ognuno dei quali esegue il proprio kernel. Questo modello di esecuzione è chiamato da Nvidia *Single Instruction Multiple Threads* (SIMT) ed è simile al modello SIMD, ma c'è una sottile ma importante differenza tra i due. In un'architettura SIMD, ogni istruzione applica la stessa operazione in parallelo su molti elementi. La SIMD è tipicamente implementata utilizzando processori con registri vettoriali e unità di esecuzione. In un'architettura SIMT, invece di un singolo thread che emette istruzioni vettoriali applicate a vettori di dati, abbiamo più thread che emettono istruzioni comuni a dati arbitrari.

I thread vengono schedulati ed eseguiti su una SM come dei blocchi, ma non tutti i thread di un blocco eseguono in modo concorrente. Ogni blocco di thread è eseguito in dei gruppi, detti **warp**. Attualmente la dimensione di un warp è 32 ma potrebbe cambiare con le successive generazioni di GPU.

Una SM può fare context switch molto velocemente da un thread all'altro perché, al contrario della CPU, ogni thread ha il proprio contesto di esecuzione e i registri memorizzati sul chip. Inoltre possono esistere più scheduler per ogni SM, permettendo a diversi warp di trasmettere sequenze di istruzioni indipendenti contemporaneamente. Ogni scheduler può emettere fino a due istruzioni contemporaneamente se queste

sono indipendenti, i.e. l'esito di una non dipende da quello dell'altra (**Instruction level parallelism**).

I thread all'interno di un warp eseguono le stesse istruzioni, ma operano su dati differenti. Di conseguenza, l'esito di un'operazione condizionale può portare alcuni thread a seguire percorsi diversi (ad esempio, alcuni il ramo then, altri l'else). In tali casi, ogni ramo viene valutato sequenzialmente, ma solo se almeno un thread lo percorre. I thread non coinvolti nell'esecuzione del ramo corrente restano in attesa. Questo meccanismo può causare significativi cali di performance, motivo per cui è importante conoscere il modello di esecuzione dei thread.

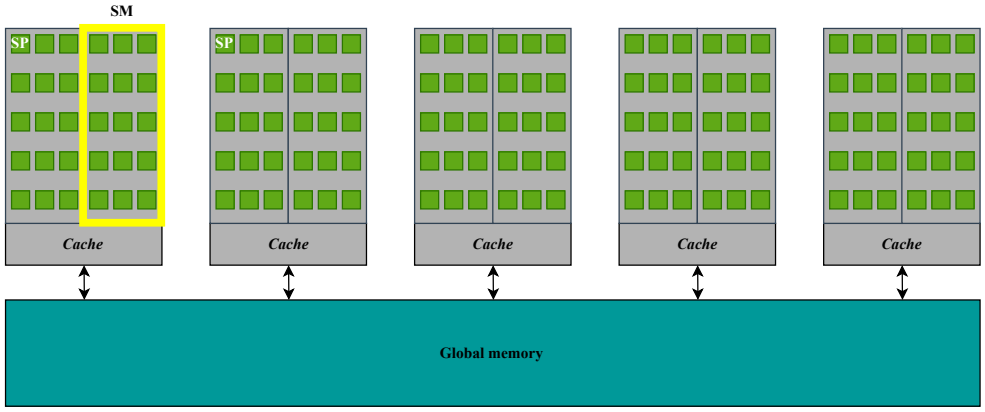


Figura 3.2. Architettura di una GPU compatibile con CUDA, in particolare sono rappresentati gli SP raggruppati in SM.

3.2 Gerarchia della memoria

Sebbene la GPU possa accelerare significativamente l'esecuzione di alcuni programmi, è fondamentale comprenderne a fondo l'architettura per sfruttare appieno il grande numero di processori disponibili. Solo così è possibile evitare colli di bottiglia legati a operazioni di lettura/scrittura in memoria e ai trasferimenti di dati tra host e dispositivo.

Solitamente memoria host e memoria device sono disgiunte, quindi all'interno del kernel non sarà possibile accedere alla memoria dell'host. Viceversa dall'host non è possibile accedere direttamente alla memoria sul device, ma tramite una `cudaMalloc` è possibile allocare memoria sulla GPU direttamente dall'host e successivamente, magari dopo la fine di un kernel, ricopiarla sull'host con una `cudaMemcpy`. Una copia può essere eseguita *host/host*, *host/device*, *device/host* o *device/device*. Quando trasferimenti e allocazioni vengono fatte tra GPU e CPU queste hanno sempre luogo al livello della **global memory**, che è una memoria ad alta capacità, ma ha una velocità relativamente bassa. Questo perché la GPU è pensata per filtrare e trasformare grandi stream di dati su ciascuno dei quali "operare" una sola volta, senza richiedere la memorizzazione sul chip per elaborazioni successive.

Esistono poi altri tipi di memorie più veloci che hanno funzioni e caratteristiche differenti:

- **local memory e registri**: ogni SM ottiene un insieme di registri e li suddivide tra i thread. I registri vengono utilizzati per memorizzare le variabili

automatiche e la quantità di registri per thread è determinta dalla compute capability dell'hardware, per esempio una Nvidia A100 possiede 64K registri per SM. Se le variabili locali memorizzate durante l'esecuzione eccedono il numero di registri, queste vengono memorizzate fuori dal chip, nella cosiddetta **local memory**. L'accesso alla local memory è molto più lento, perché risiede a tutti gli effetti nella global memory. Il numero di registri utilizzati per thread influenza l'**occupancy**, ovvero il rapporto tra warp residenti e warp massimi che una SM può ospitare.

- **Shared memory e cache L1**: la cache L1 e la shared memory fanno parte dello stesso blocco di memoria su chip e sono entrambe specifiche per SM. A differenza della L1, la gestione della shared memory è lasciata all'utente. Essendo sia L1 che shared memory parte dello stesso blocco, la quantità di memoria tra le due può essere partizionata in modi diversi, e.g. con CC $\geq$ 8.0 ogni SM ha 192 KiB di memoria, con l'opzione di dedicare 164 KiB alla shared memory e i restanti alla L1. La shared memory può essere utilizzata per mantenere dati acceduti con frequenza che altrimenti richiederebbero scritture o letture sulla global memory. La cache L2 invece, è condivisa tra le SM.

- Constant memory : memoria fuori dal chip, di sola lettura, che può però può essere copiata in cache per garantire accessi rapidi.

- Texture e Surface memory: memoria fuori dal chip che permette implementazioni veloci di particolari operazioni di filtraggio.

In particolare CUDA assume che il device abbia un modello di memoria **weakly-ordered**, i.e. l'ordine di scrittura su memoria (qualunque essa sia) da parte di un thread non è lo stesso ordine di scrittura osservato dagli altri thread o dall'host. Perciò è undefined behavior per due thread scrivere o leggere la stessa zona di memoria senza sincronizzazione. Per questo esistono delle funzioni chiamate **memory fence**, che permettono di sincronizzare le scritture e le letture in memoria da parte del thread chiamante al livello di singolo device, con `__threadfence`, o di device, host e **peer device**[1] con `__threadfence_system` .

3.3 Ottimizzazioni

Accessi coalescenti

Anche se i registri offrono un accesso estremamente veloce ai dati, in alcuni momenti dell'esecuzione i thread dovranno comunque accedere alla global memory, che nei dispositivi CUDA è implementata tramite DRAM e presenta latenze significativamente superiori. I bit sono immagazzinati in dei piccoli condensatori dove la presenza o l'assenza di carica determina il bit 1 o 0. Leggere da una cella significa che quel condensatore dovrà usare la sua piccola carica per attivare un sensore che deciderà se la quantità di carica è sufficiente per valutare il bit come "1". Questo processo impiega decine di nanosecondi, andando in contrasto con la velocità dei cicli di clock sotto il nanosecondo degli hardware moderni.

Per ammortizzare questa latenza le DRAM attualmente **parallelizzano** gli accessi: ogni volta che si fa l'accesso ad una cella, una serie di celle consecutive, oltre quella a cui si voleva accedere, vengono caricate (DRAM *burst*). Quindi se si

[1] Device collegati (tramite NVLink o PCIe) che permettono il trasferimento e l'accesso ai dati senza passare per la memoria host.

fa un accesso sistematico alla global memory tramite dei *burst* si ottengono i dati a velocità superiori rispetto ad una sequenza di accessi a zone randomiche della memoria.

I device CUDA impiegano una tecnica che consente ai programmatori di ottenere un'elevata efficienza di accesso alla memoria globale organizzando gli accessi alla memoria da parte dei thread in schemi favorevoli. Questa tecnica sfrutta il fatto che i thread in un warp eseguono la stessa istruzione in qualsiasi momento. Quando tutti i thread in un warp eseguono un'istruzione di caricamento, l'hardware rileva se accedono a posizioni di memoria globale consecutive. In questo caso, l'hardware combina tutti questi accessi in un unico accesso a locazioni consecutive sulla DRAM.

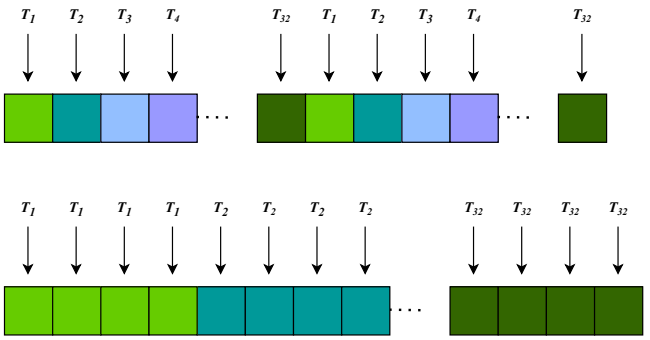


Figura 3.3. Pattern di accesso coalescente (sopra) e non coalescente (sotto) alla memoria da parte di un warp.

Stream CUDA

Uno **stream** è una sequenza di comandi (comprese le copie di memoria e le invocazioni al kernel) che vengono eseguiti in ordine. In un programma CUDA si può controllare la concorrenza a livello di device gestendo gli stream. Ogni stream esegue in modo sequenziale, ma istruzioni da stream differenti possono essere eseguite in maniera concorrente. Se un comando non viene associato esplicitamente a nessuno stream allora sarà assegnato allo stream di default. Uno stream può essere creato tramite una chiamata alla funzione `cudaStreamCreate` e passato come parametro ai kernel o a tutte quelle chiamate che hanno come suffisso `Async`. Successivamente potrà essere distrutto con `cudaStreamDestroy`, che è una funzione non bloccante e permette il rilascio delle risorse associate allo stream non appena tutte le operazioni in attesa vengono completate. Bisogna tenere conto di una serie di operazioni che bloccano l'esecuzione concorrente di ogni stream: allocazioni **page-locked**, allocazioni su device (`cudaMalloc`), settaggio della memoria device con `cudaMemset`, qualunque comando che sia aggiunto allo stream di default e modifiche nella configurazione della shared memory o cache L1.

Primitive di warp

Le funzioni di warp (o primitive di warp) sono delle funzioni che permettono la comunicazione, scambio di dati e semplici operazioni fra warp senza l'utilizzo della shared memory, ma solo attraverso i registri. Nel contesto delle primitive di warp, i thread che costituiscono un warp sono chiamati *lanes*. Ogni lane ha un identificatore univoco che è un intero nell'intervallo $[0, 31]$. Ogni primitiva di warp

prende come parametro una maschera di 32 bit, che identifica quale lane parteciperà all'operazione assegnando a ciascuna di esse un bit, e.g. se tutti i thread del warp partecipano la maschera assumerà il valore `0xFFFFFFFF`.

Le primitive di warp di **shuffling** vengono utilizzate per lo scambio di dati e per implementare operazioni collettive a livello di warp (e.g. broadcast e reduce come in Figura 4.4). La funzione `__shfl_down_sync(mask, var, delta, width)`, chiamata dal thread con *lane* = *i*, prende come parametri la maschera di thread che partecipano allo scambio, e legge il valore *var* dal registro del thread *i* + *delta* se *i* + *delta* ≤ *width*, altrimenti ritorna il valore *var* della *lane* stessa. Esiste anche una versione *"up"* per lo shuffle che legge *var* dal thread con *lane* = *i* − *delta* e una versione in cui non viene specificato il *delta*, bensì la lane *source* da cui tutti leggono *var*. Ci sono poi le funzioni di *match*, che servono a confrontare i valori o predicati passati in input fra le lane selezionate tramite la maschera, funzioni come `__activemask` che ritornano la maschera corrispondente ai thread che hanno eseguito la chiamata (utile nei branch) e primitive di sincronizzazione sia al livello di warp (`__syncwarp`) che di thread block (`__syncthreads`), garantendo che tutte le istruzioni precedenti siano completate da tutti i thread del warp/blocco prima di procedere.

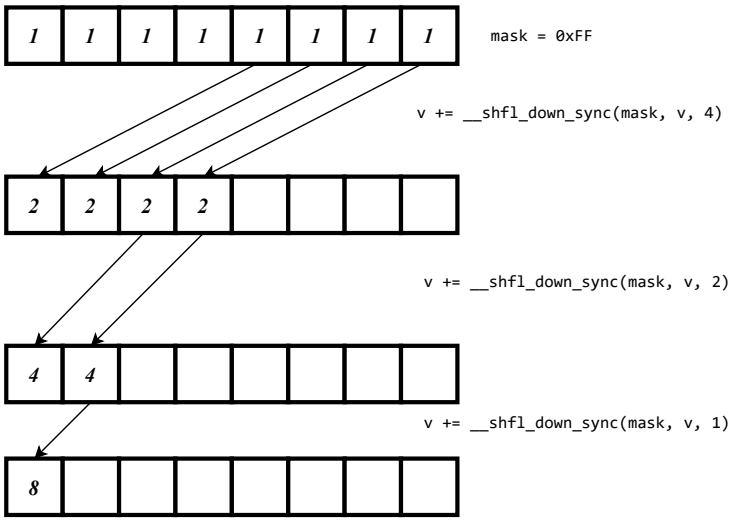


Figura 3.4. Reduce ad albero al livello di warp implementata tramite shuffle.

3.4 Roofline model

Il *Roofline model* è un modello di performance che descrive i limiti imposti dall'hardware alle prestazioni di un algoritmo, mettendo in relazione la potenza di calcolo e la larghezza di banda della memoria. Il modello proposto da Samuel Williams, Andrew Waterman e David Patterson nel 2009 [31] lega insieme le performance computazionali e quelle della memoria in un grafico a due dimensioni. Più precisamente viene definita l'*intensità operazionale (operational intensity)* che misura le operazioni effettuate su ciascun byte proveniente dalla DRAM. Questa è una misura più generale dell'*intensità aritmentica* (IA), che invece misura il numero

di operazioni in virgola mobile (Flops) per byte caricato dalla memoria (che sia DRAM , cache o registri).

Sull'asse X del grafico viene rappresentata l'intensità operazionale (o aritmetica), mentre sull'asse Y si misura il numero di operazioni al secondo raggiungibili. Nel caso delle operazioni in virgola mobile avremo che

$$\text{GFlops/s raggiungibili} = \min\{Y_{peak},\ M_{peak} \times \text{IA}\} \tag{3.1}$$

Dove Y_{peak} è il numero massimo di operazioni in virgola mobile al secondo che può fornire l'hardware in Flop/s, mentre M_{peak} è la velocità massima a cui un processore può caricare dati dalla memoria espressa in byte/s.

Per rappresentare esplicitamente la memory bandwidth in questo modello è sufficiente fare un'osservazione: dato che sull'asse X si misura in GFlops/byte e sull'asse Y si misura in GFlops/s, la memory bandwidth, che si misura in byte/s ovvero $\frac{\text{GFlops/s}}{\text{GFlops/byte}}$, è rappresentata da una linea che parte dall'origine ed è inclinata di 45° rispetto agli assi. Y_{peak} invece è una linea orizzontale perché è stabilita da un limite fisico della macchina. Le due linee si intersecano nel punto di massima performance sia di memory bandwidth che di operazioni al secondo.

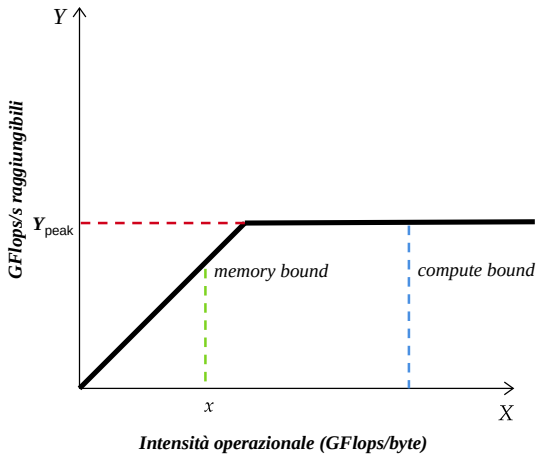


Figura 3.5

Dato un kernel possiamo trovare la sua intensità aritmetica x sull'asse X e i GFlops/s raggiungibili si trovano in qualche punto della retta verticale che parte da x e rimane sotto alle due linee descritte in precedenza (vedi la linea verde della Figura 4.5). La linea orizzontale unita a quella diagonale, alle quali faremo riferimento con il nome unico di *Roofline*, determinano un limite superiore alle performance del kernel che dipende dalla sua intensità operazionale. Se pensiamo all'intensità operazionale come a una colonna che arriva fino alla *Roofline*, questa o colpirà la parte spiovente, e quindi la performance si dice essere **memory bound**, o colpirà la parte piatta, in quel caso il kernel sarà **compute bound**.

Se il punto di cresta della *roofline* è spostato molto a destra significa che il kernel dovrà avere un'alta intensità operazionale per raggiungere le prestazioni di picco. In modo opposto, se questo punto si trova spostato verso sinistra, allora quasi tutti i kernel possono raggiungere la performance massima.

3.5 MPI GPU-aware e comunicazioni intra-nodo e inter-nodo

Dato che MPI è lo standard di comunicazione più utilizzato in HPC, molti sforzi sono stati dedicati a rendere MPI utilizzabile anche dai modelli di programmazione GPU. Con MPI GPU-aware si intende un'implementazione di MPI che può fare distinzione tra device e host buffer. Prima della creazione di questo tipo di implementazioni, le comunicazioni che avvenivano tra più GPU necessitavano di una copia dei buffer **device→host** da parte del mittente e poi una copia **host→device** del destinatario.

Con *MPI GPU-aware* è possibile fornire alle chiamate MPI direttamente i buffer su device, permettendo alla comunicazione di avvenire sul **data path** determinato da alcuni meccanismi che evitano copie ridondanti tra host e device. L'obiettivo dell'implementazione GPU-aware è quello di minimizzare i trasferimenti tra host e device rendendo le applicazioni GPU-*centriche*, in modo da beneficiare delle potenti capacità computazionali offerte dall'hardware e riducendo i colli di bottiglia dovuti alle comunicazioni.

Assumiamo il caso concreto di un sistema multi-nodo, in cui in ogni nodo risiede un processo/thread che ha il controllo di due o più GPU. Allora all'interno del sistema possiamo fare distinzione tra due tipi di comunicazioni multi-GPU: comunicazioni **inter-nodo** e comunicazioni **intra-nodo**.

Le comunicazioni **intra-nodo** possono essere gestite lato host con due copie (come descritto in precedenza), oppure tramite API host-side come NCCL o MPI GPU-aware, oppure ancora tramite API o meccanismi lato device previsti dal tipo di GPU, che permettono un accesso trasparente alla memoria degli altri device. Ad esempio, a partire da CUDA 4.0 è supportato un sistema di indirizzamento virtuale unificato tra host e device, chiamato Unified Virtual Addressing (**UVA**). Grazie a quest'ultimo sono state introdotte le comunicazioni peer-to-peer tra GPU che condividono un singolo nodo root PCIe e una serie di switch PCIe che si diramano dal root e collegano le GPU (**GPUDirect 2.0 o P2P**). Quindi, invece di trasferire i dati all'host, le GPU possono accedere alla memoria del peer tramite i PCIe.

Le comunicazioni **inter-nodo** sono un po' più complesse rispetto a quelle intra-nodo perché richiedono interazioni con la NIC. Con le versioni di CUDA 3.1, la **GPUDirect 1.0**[2] permetteva alle GPU e alle NIC di condividere la stessa regione di **pinned memory**[3]. Precedentemente le pinned memory di GPU e NIC erano disgiunte e quindi, per comunicare i dati di una GPU agli altri nodi, era necessaria una copia sulla sua pinned memory. Successivamente, la CPU copiava i dati sulla memoria della NIC, che in questo modo poteva accedervi e distribuirli alla rete. La copia intermedia inizializzata su CPU tra le regioni di memoria pinnate di GPU e NIC introduce un overhead per la CPU e una latenza non trascurabile per le comunicazioni GPU. GPUDirect 1.0 ha introdotto una pinned memory condivisa GPU-NIC evitando la copia intermedia su CPU.

Con l'introduzione di **GPUDirect RDMA** (Remote Direct Memory Access) in **CUDA 5.0**, le comunicazioni dirette tra GPU Nvidia sono diventate significa-

[2]Con GPUDirect ci si riferisce a una famiglia di tecnologie che migliora i trasferimenti di dati tra le GPU. Nel contesto di MPI, le tecnologie GPUDirect coprono tutti i tipi di comunicazione inter-rank: intra-nodo e inter-nodo.

[3]La pinned memory è utilizzata per inizializzare i trasferimenti da host a GPU, infatti la memoria allocata tramite cudaMalloc dall'host è *pageable* e quindi non è accessibile dal device. In un trasferimento da host a device è necessario che la memoria pageable venga copiata temporaneamente su un buffer page-locked per poi essere copiata sulla GPU.

tivamente più efficienti. I trasferimenti RDMA utilizzano la *pinned memory*, non swappabile dal sistema operativo, che consente operazioni indipendenti dalla CPU e dall'OS stesso.

In pratica, questo avviene esponendo una porzione della memoria della GPU nello **spazio indirizzabile PCIe**, attraverso una regione chiamata **Base Address Register (BAR)**. Ciò permette alle NIC di accedere direttamente alla memoria della GPU, effettuando letture e scritture senza passaggi intermedi. GPUDirect RDMA fornisce diverse ottimizzazioni al data path, in particolare eliminando copie aggiuntive nella memoria host , riducendo le latenze intrinseche derivanti dall'interazione GPU-NIC, aumentando la larghezza di banda e riducendo l'overhead della CPU. Il supporto per GPUDirect RDMA è stato integrato in diverse librerie di comunicazione leader del settore, tra cui le implementazioni MPI GPU-aware.

Parte II

Analisi dello stato dell'arte

Capitolo 4

Compressione accelerata su GPU

Come già detto nell'**introduzione** si prevede che i futuri sistemi exa-scale mostreranno una tendenza per cui il movimento dei dati tra le gerarchie di memoria e il trasferimento dei dati tra i nodi diventerà relativamente costoso in termini di tempo ed energia, rispetto alla potenza di calcolo in costante aumento. Per questo motivo applicare la compressione dei dati, e quindi ridurli nella loro dimensione, può portare un vantaggio significativo alle applicazioni scientifiche che vengono eseguite su questi sistemi.

Visto che i dataset scientifici sono in maggior parte composti da numeri in virgola mobile (in singola o doppia precisione), un algoritmo di compressione standard può portare a una bassa riduzione in dimensione dei dati e a un grande overhead dovuto alla procedura di compressione.

Per incrementare il *compression factor* e diminuire l'overhead si è iniziato ad esplorare più approfonditamente le caratteristiche dei dati utilizzati in queste simulazioni, in modo da costruire dei compressori ad-hoc che siano più performanti su specifiche applicazioni. In questi algoritmi si sfruttano dei pattern ripetivi nei dati spazialmente o temporalmente adiacenti (ad esempio tra due iterazioni successive).

La compressione lossy, come spiegato nel Cap. 1, produce un *compression factor* più alto con l'introduzione di un errore che spesso, nelle simulazioni su larga scala, può impattare sull'esito della simulazione, rendendo i dati compressi inutilizzabili.

Alcuni compressori come ZFP [14] permettono all'utente di regolare la qualità della ricostruzione e il compressione rate lasciando la scelta tra due modalità di compressione:

- **Fixed-rate**: si fissa il compression rate, senza tenere conto della distorsione.
- **Error-bounded**: si fissa un *bound* sull'errore e la compressione viene effettuata in modo da non superare quel limite.

La compressione lossless invece permette di ricostruire i dati originali senza errore, ma spesso richiede algoritmi più complessi con compression factor nettamente inferiore a quello della compressione lossy.

Nel seguito vengono analizzate alcune idee algoritmiche alla base di compressori efficienti e, più nello specifico, dei compressori accelerati tramite GPU.

4.1 Meccanismi e struttura della compressione su GPU

Trasformazioni sul dataset

Per ottenere fattori di compressione elevati, un algoritmo deve cercare la ridondanza nella codifica dei dati ed eseguire una complessa trasformazione per una loro rappresentazione compatta. Di conseguenza, sono tipici gli alti costi di CPU/GPU e/o di memoria sia per la compressione che per la decompressione. Mentre molti schemi di compressione sfruttano lunghe sequenze di byte ripetuti, i dataset scientifici sono spesso composti da numeri in virgola mobile che rappresentano informazioni ad alta entropia. Nel corso del tempo sono state proposte alcune procedure che sfruttano la somiglianza locale sia spaziale che temporale dei dati.

Prendiamo come esempio la procedura proposta da *Bicer et al.* [2] per dataset climatici, dove la temperatura di una regione dipende fortemente dalla temperatura di quella regione allo step temporale precedente, ma anche dalla temperatura delle regioni vicine. Questa dipendenza è tipicamente espressa da una differenza molto piccola tra i loro valori numerici. Se consideriamo due celle X e Y vicine spazialmente o temporalmente, allora possiamo memorizzare solo X e la differenza $\delta = Y - X$ (ovvero una *linear prediction del primo ordine*) invece di memorizzare sia X che Y. In particolare si possono codificare tutte le temperature in binario e fare uno XOR dei dati adiacenti. Se le differenze sono piccole avremo una sequenza di z zeri prima del valore effettivo di $\delta = X \oplus Y$. Quindi per codificare la versione compressa di Y basterà memorizzare le cifre significative di δ e il valore di z in binario. Utilizzando un approccio lossy si potrebbe considerare di tralasciare alcune delle cifre meno significative di δ.

Altre tecniche di compressione utilizzate dividono ciascun dato in delle parti considerate compressibili e incompressibili, oppure, immaginando i dati incolonnati, applicano trasformazioni alle colonne cercando di trovare segmenti di byte da comprimere. Oppure si utilizzano tecniche di compressione lossy come la quantizzazione con bound sull'errore in modo da ridurre la dimensione dei dati anche quando la loro distribuzione non è uniforme come nel caso precedente.

Architettura del kernel CUDA

Progettare uno schema di compressione efficiente per l'esecuzione su GPU non è un compito semplice. A differenza delle controparti ottimizzate per CPU, è necessario considerare diversi fattori tra quelli elencati nel Cap. 3 specifici dell'architettura GPU (in particolare delle architetture compatibili con CUDA). In generale, quando si vuole sfruttare una macchina a più processori per comprimere, essendo le procedure utilizzate quasi sempre sequenziali, quello che si può fare è dividere lo stream di dati da comprimere in dei blocchi e distribuirli a ogni processo (o thread). Quest'ultimo elaborerà in modo sequenziale la sua porzione di dati e, alla fine, i blocchi compressi verranno aggregati insieme ai *metadati* utili alla decompressione (e.g. offset di blocco, encoding utilizzato).

Nello specifico, se si vuole costruire un compressore efficiente su GPU, bisogna considerare colli di bottiglia come gli accessi alla global memory e la possibilità che certi schemi di compressione portino a un basso utilizzo degli SM. Idealmente un buon compressore dovrà :

1. **dividere il dataset in blocchi**, in modo da parallelizzare le operazioni tra i vari processori;

2. **minimizzare gli accessi alla global memory**;

3. avere un elevato **compression/decompression throughput** e **compression rate**;

Nella pratica non verranno utilizzati algoritmi di compressione che necessitano della creazione di dizionari o strutture dati più complesse (e.g. *Huffman trees*), ma si cercherà di costruire una funzione che trasforma i dati sfruttando pattern ripetitivi locali al blocco. Dopo la fase di codifica l'algoritmo prevede una fase in cui si salvano i *metadati*: questa è forse una delle fasi più delicate perché richiede la cooperazione e coordinazione tra i thread che dovranno scambiare dati e fare diversi accessi a regioni di memoria condivisa.

La fase di decompressione è una trasformazione inversa e generalmente più veloce rispetto alla compressione. La divisione in blocchi avviene come per la compressione sfruttando la dimensione **originale** (ovvero prima della compressione) del dataset. Nella fase successiva si utilizzano i *metadati* per estrarre tramite alcune operazioni i dati originali o, nel caso della compressione lossy, una loro approssimazione.

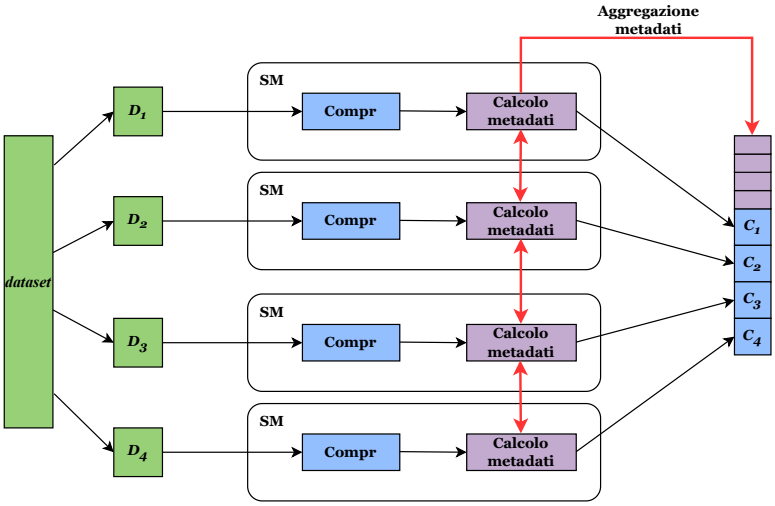


Figura 4.1. Architettura di un compressore su GPU a livello di Streaming Multiprocessor (SM): anche all'interno di ciascun SM si adotta uno schema analogo a quello descritto in precedenza. Ogni SM suddivide il proprio blocco in sottoblocchi, che vengono elaborati dai singoli thread.

4.2 cuSZp

cuSZp è un compressore lossy su GPU ideato da *Yafan Huang et al.* [11] che aumenta in modo significativo il throughput end-to-end[1]. Altri compressori, come cuSZ [28] o cuZFP, offrono un'elevata velocità di esecuzione dei kernel, ma presentano alcune limitazioni significative. Ad esempio, cuSZ esegue parte del programma sull'host e suddivide le fasi di compressione e decompressione in più kernel,

[1]Con throughput end-to-end si intende il periodo di tempo dal momento in cui i dati popolano la global memory, al momento in cui i dati compressi vengo completamente salvati in global memory. Questo viene specificato perché le performance di molti compressori vengono valutate esclusivamente sulla base del kernel throughput. Nel caso di compressori con kernel multipli, nelle valutazione venivano tralasciati i costosi movimenti di dati tra device e host.
Di fatto con un singolo kernel, *kernel throughput = end-to-end throughput.*

introducendo un'inefficienza complessiva. cuZFP, invece, evita questi svantaggi grazie a un'implementazione compatta in un singolo kernel GPU; tuttavia, soffre di una scarsa qualità nella ricostruzione dei dati, poiché supporta unicamente la modalità fixed-rate.

cuSZp implementa la compressione in un unico kernel, bilanciando tra elevato throughput di compressione (decompressione) e compression-rate, mantenendo l'errore limitato sui dati ricostruiti. cuSZp riesce ad ottenere un throughput end-to-end di 93.63 GB/s e 120.04 GB/s in media, rispettivamente per compressione e decompressione, circa 93.53× più veloce di cuSZ. Inoltre, rispetto agli altri compressori, garantisce un compression-factor più alto mantenendo lo stesso errore sui dati.

A questo proposito, l'errore può essere misurato ponendo un limite sull'**errore assoluto (ABS)** oppure calcolando un **errore relativo** al range di valori che assumono i dati in input **(REL)**. Il limite sull'errore ABS δ è un valore costante che viene deciso prima della compressione. Il limite sull'errore REL invece, viene rappresentato da λr, dove $\lambda \in (0, 1)$ denota il *rapporto relativo* scelto ed r esprime l'intervallo di valori presenti nel dataset (i.e. *valore massimo - valore minimo*).

Panoramica sul compressore

Preso un dataset in input, cuSZp tratta questo dataset come un array 1D e lo suddivide in blocchi di uguale lunghezza. Se un blocco non è un **blocco di zeri**, ovvero un blocco con soli "0", allora esegue una quantizzazione, poi una linear prediction ed infine una fixed length encoding generando un offset per il blocco. Nel caso si presenti un blocco di zeri, cuSZp lo bypassa limitandosi a registrare che si tratta appunto di un blocco nullo. Successivamente, ogni offset viene scritto nell'array degli offset, con lunghezza uguale al numero di blocchi in cui si è diviso il dataset. Effettua una sincronizzazione globale degli offset. Infine esegue un **block bit-shuffle** e scrive i dati compressi nelle posizioni calcolate.

In modo simile alla compressione, cuSZp esegue una sincronizzazione globale degli indici per trovare la posizione dei blocchi compressi. Trovate le posizioni, si applica un block-bit shuffle inverso, una fixed-length encoding inversa e si inverte il processo di *prediction* e quantizzazione.

Nelle sezioni che seguono analizzeremo più nel dettaglio tutti i passaggi della procedura di compressione .

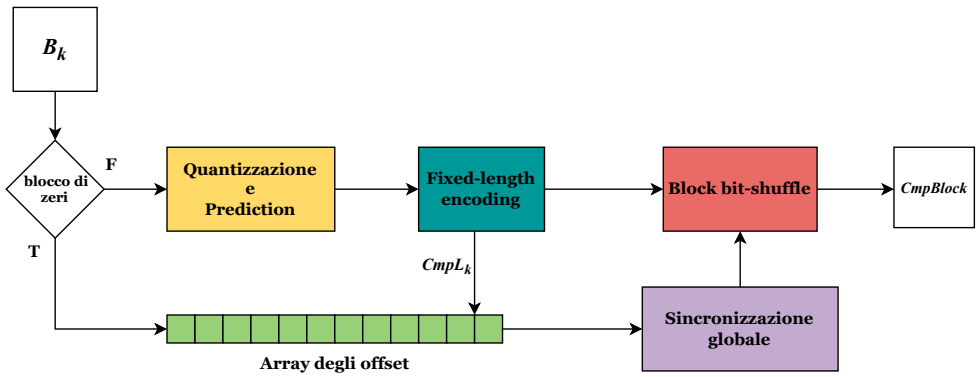


Figura 4.2. Schema di compressione introdotto da cuSZp.

4.2.1 Quantizzazione e *prediction*

L'obiettivo della fase di quantizzazione è convertire i numeri in virgola mobile in numeri interi, riducendo così la variabilità dei loro bit e preparando i dati per ulteriori compressioni nelle fasi successive.

Dato un blocco $B = \{d_1, d_2, ..., d_L\}$, dove d_i rappresenta l'i-esimo float e L la lunghezza del blocco, la fase di quantizzazione genera una sequenza $\{r_1, r_2, ...r_L\}$, dove ogni intero r_i può essere calcolato prendendo la parte intera di $(d_i/2eb)$ garantendo che $|r_i \times 2eb - d_i| \leq eb$ (Cap. 1). La ricostruzione del float si ottiene con $d'_i = r'_i \times 2eb$. Un limite sull'errore molto piccolo può portare alla creazione di interi molto grandi, per questa ragione nella fase successiva andremo ad eliminare eventuali bit pattern ripetitivi. Nella fase di *prediction* (Figura 4.2) si prende la sequenza di interi generata dalla quantizzazione e si memorizzano le differenze dei dati adiacenti quindi, presi due interi adiacenti r_i e r_{i-1}, memorizziamo il valore $l_i = r_i - r_{i-1}$, dove $r_0 = 0$.

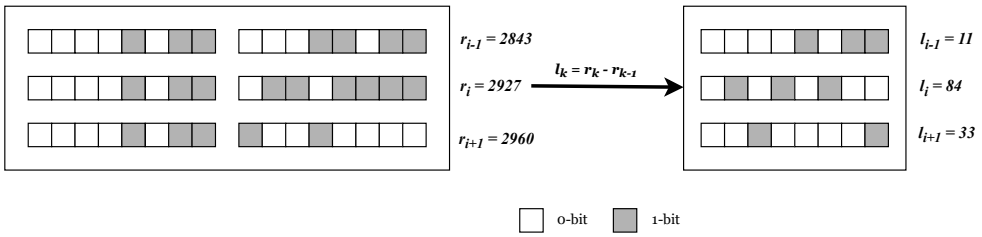


Figura 4.3. Esempio concreto di eliminazione dei pattern ripetitivi di bit dopo la quantizzazione. In questo caso supponiamo che r_{i-1} non sia il primo elemento del blocco (ovvero $i > 2$) e quindi il risultato l_{i-1} è dato da $r_{i-1} - r_{i-2} = 2843 - 2832 = 11$, in particolare l'unico valore che sarà memorizzato per intero sarà r_1.

4.2.2 Fixed-length Encoding

A partire dalla sequenza $\{l_i\}_{i=1}^{L}$ di interi, ottenuta tramite *linear prediction*, si memorizza per ciascun elemento il segno utilizzando un singolo bit: 1 se negativo, 0 se positivo. Tutti questi bit vengono raccolti in una mappa dei segni. Quindi per un blocco di lunghezza L, cuSZp richiede dello spazio extra pari a $L/8$ *unsigned char* per memorizzare i segni. Successivamente, si calcola il valore assoluto di ciascun l_i e iterando sui valori del blocco si trova quello massimo. Quest'ultimo stabilisce la lunghezza della codifica di ciascun elemento del blocco. Ad esempio in un blocco con valori $\{1, 4, 0, 12, 4, 3, 1, 1\}$, il massimo è "12" e stabilisce che il blocco deve essere codificato con lunghezza fissa pari a 4 bit. Sapendo la lunghezza di ciascuna codeword si ricava la lunghezza del blocco compresso B_k:

$$\mathrm{Cmp}L_k = \frac{F_k \times L}{8} + \frac{L}{8} \tag{4.1}$$

dove F_k è la lunghezza fissata per la codifica e il termine $L/8$ rappresenta la dimensione della mappa dei segni. La codifica a lunghezza fissa (fixed-length encoding) è stata preferita rispetto a strategie più efficienti in termini di compression rate, come la codifica di Huffman, che assegna meno bit alle codeword più frequenti. Tuttavia, nei blocchi generati da cuSZp è raro trovare dati ad alta frequenza. Inoltre, la codifica a lunghezza fissa è estremamente semplice: non introduce costi computazionali aggiuntivi né richiede spazio extra per memorizzare strutture dati ausiliarie per la codifica.

4.2.3 Sincronizzazione globale

Tramite la sincronizzazione globale (*global synchronization*) in cuSZp vengono generati gli indici di ogni blocco compresso per l'intero dataset. Questo passaggio è cruciale e deve essere eseguito con la massima precisione per non compromettere il compression rate. Al termine della fase di Fixed-length encoding, infatti, i blocchi compressi non hanno tutti la stessa lunghezza: per individuare l'inizio dell'i-esimo blocco è quindi necessario conoscere le lunghezze di tutti i blocchi precedenti. Ciò richiede un accumulo globale e sincronizzato di tutti i valori $\mathrm{Cmp}L_k$ calcolati dai singoli thread. Nei compressori precedenti, la sincronizzazione globale veniva tipicamente eseguita sulla CPU, causando costosi trasferimenti di dati tra CPU e GPU. In cuSZp, invece, l'intera sincronizzazione avviene direttamente sulla GPU ed è formulata come un problema di **prefix-sum esclusiva**[2] , garantendo maggiore efficienza ed evitando overhead inutili.

Come descritto nella sezione precedente, ogni blocco B_k dopo la compressione ha dimensione $\mathrm{Cmp}L_k$. Quindi B_0 all'interno del blocco finale compresso, occuperà gli indici da 0 a $\mathrm{Cmp}L_0$, il blocco B_1 sarà posizionato nell'intervallo di indici tra $\mathrm{Cmp}L_0$ e $\mathrm{Cmp}L_0 + \mathrm{Cmp}L_1$. In generale, il blocco B_i avrà indici tra $\sum_{k=0}^{i-1} \mathrm{Cmp}L_k$ e $\sum_{k=0}^{i} \mathrm{Cmp}L_k$. Quindi l'output della fase di sincronizzazione globale sarà un array di lunghezza N (i.e. il numero di blocchi in cui è stato diviso il dataset) del tipo:

$$\mathrm{GlobalSync} = \left[0, \sum_{k=0}^{0} \mathrm{Cmp}L_k, \ldots, \sum_{k=0}^{N-1} \mathrm{Cmp}L_k, \sum_{k=0}^{N} \mathrm{Cmp}L_k\right] \tag{4.2}$$

In cuSZp la sincronizzazione globale viene implementata in modo gerarchico sfruttando a tutti i livelli il parallelismo offerto dal modello computazionale delle GPU CUDA compatibili. Dato in input l'array

$$\mathrm{BlockOffs} = [\mathrm{Cmp}L_0, \mathrm{Cmp}L_1, \ldots, \mathrm{Cmp}L_N] \tag{4.3}$$

in primo luogo viene effettuata una prefix-sum al livello di thread, dove ogni thread somma gli offset di blocco su cui ha lavorato. Poi viene svolta una prefix-sum al livello dei warp, tramite lo **shuffle** (Cap 3.3), generando l'offset di warp. Dato che il kernel viene eseguito con un solo warp per threadblock, questa fase completa anche la prefix-sum al livello di threadblock. Infine, viene eseguita una **prefix-sum globale**, implementata mediante una *chained-scan parallelization*, dalla quale si ricavano sia gli offset dei blocchi compressi sia la dimensione finale del dataset compresso.

4.2.4 Block Bit-shuffle

La fase di Block Bit-shuffle riorganizza i dati compressi e codificati in binario prima di scriverli nello spazio di memoria, i cui limiti sono stati calcolati tramite la sincronizzazione globale. L'obiettivo del block bit-shuffle è quello di trasformare gli interi codificati in un insieme allineato di byte facile da memorizzare.In questo modo si evita uno shift irregolare dei bit quando la lunghezza fissa non è divisibile per 8, ottimizzando il flusso di controllo del programma e rendendo il processo particolarmente adatto alla parallelizzazione su GPU. Consideriamo ad esempio un blocco B di lunghezza $L = 8$ con al suo interno gli interi codificati $\{l_1, l_2, ...l_8\}$. Con il block bit-shuffle, invece di memorizzare direttamente gli interi codificati, il

[2] Una prefix-sum esclusiva prende in input un array di n elementi e restituisce in output un array
$$[0, x_0, x_0 + x_1, \ldots, (x_0 + x_1 + \cdots + x_{n-2})]$$
in cui l'ultimo elemento contiene il contributo di tutte le x_i fino a x_{n-2}, escluso quello di x_{n-1}.

k-esimo bit di ciascun l_i viene memorizzato nel k-esimo byte. In altre parole, invece di memorizzare ciascun l_i in modo contiguo in memoria, memorizziamo in modo contiguo il k-esimo bit di ogni l_i, quindi nel caso caso specifico in cui $|B| = 8$, il primo byte sarà occupato dal primo bit di ogni intero in B, il secondo byte contiene il secondo bit di ciascun intero in B e così via per tutti gli altri bit.

4.3 cuSZp2

L'algoritmo di compressione (e decompressione) descritto in precedenza, nonostante sia molto più efficiente dei suoi predecessori, rimane ancora limitato da uno scarso utilizzo della memory bandwidth. cuSZp2 [10] migliora alcune delle parti di cui si componeva cuSZp, aumentando l'utilizzo della banda di memoria, throughput e compression rate.

Più nello specifico, viene proposta una nuova modalità di fixed-length encoding per migliorare il compression rate, si utilizzano accessi vettorializzati alla memoria per aumentare l'utilizzo della memory bandwidth e viene modificata la prefix-sum utilizzata nella sezione di sincronizzazione globale.

4.3.1 Outlier Fixed-Length encoding (Outlier-FLE)

La fase di quantizzazione e linear prediction del primo ordine rimane la stessa anche in cuSZp2. Come detto in precedenza, la linear prediction del primo ordine risulta efficace nei dataset che riscontrano somiglianza nei valori spazialmente vicini. Spesso però accade che il primo elemento del blocco su cui si applica questa trasformazione rimanga di grandi dimensioni, compromettendo la codifica a lunghezza fissa del resto del blocco.

La soluzione proposta è quella di memorizzare delle informazioni aggiuntive per la codifica dell'outlier (il valore più grande del blocco[3]) all'interno dell'offset di blocco. Infatti, nell'offset di blocco si memorizza in un byte un intero compreso tra 0 e 31, che indica la lunghezza della codifica per il blocco. Quindi possiamo rappresentare la lunghezza con 5 bit e nei restanti 3 bit memorizziamo le informazioni per codificare l'outlier. Il bit più significativo dei tre funge da flag per indicare se si utilizza la modalità Outlier-FLE oppure Plain-FLE (una normale fixed-length encoding). I due bit successivi vengono utilizzati per codificare in modo adattivo la dimensione dell'outlier: 00, 01, 10 o 11 indicano rispettivamente dimensioni dell'outlier pari a 1, 2, 3 o 4 byte. Questa strategia adattiva migliora i rapporti di compressione e l'efficienza di archiviazione degli outlier. senza aggiungere overhead.

4.3.2 Accesso vettorializzato alla memoria

I kernel di compressione su GPU sono quasi sempre *memory bound*, basti pensare che cuSZp, nonostante gli enormi miglioramenti all' end-to-end throughput e compression rate, ha un memory throughput osservabile di 397.26 GB/s, sostanzialmente inferiore alla capacità di larghezza di banda di una Nvidia A100 che ammonta a 1555 GB/s.

Il memory throughput è migliorato in cuSZp2 grazie all'utilizzo di letture e scritture in memoria globale tramite il `float4`, ovvero vettori di 4 float forniti dall'API di CUDA. L'accesso in lettura o scrittura tramite questi tipi vettoriali viene tradotto con una sola istruzione nel linguaggio assembly delle GPU Nvidia, che

[3] quindi l'encoding descritto si applica anche nei casi in cui l'outlier non è il primo elemento nel blocco.

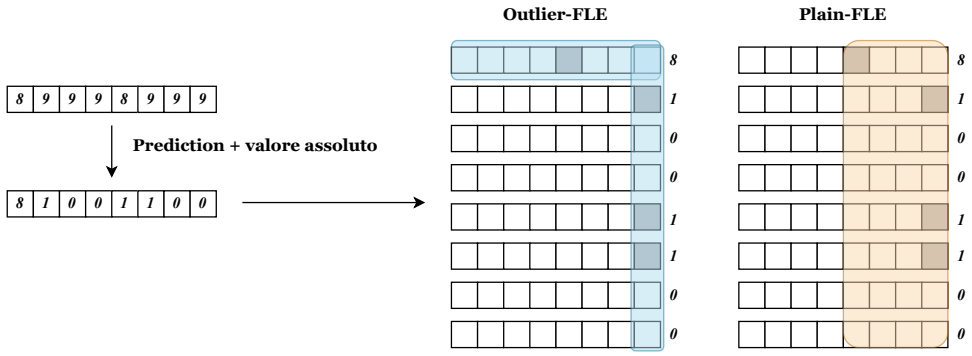


Figura 4.4. Nella codifica con Outlier-FLE si utilizzano 2 Byte per memorizzare i valori del blocco, mentre con Plain-FLE vengono utilizzati 4 Byte.

carica o scrive 128 bit. Gli accessi vettoriali vengono combinati a pattern di accesso coalescenti dei thread di un warp durante le iterazioni, risultando in un memory throughput di 1330.24 GB/s nella fase di compressione.

4.3.3 Sincronizzazione globale con *decoupled lookback*

La sincronizzazione di un compressore parallelo a blocchi rappresenta una fase critica per le prestazioni, poiché richiede operazioni di lettura e scrittura sincronizzate, organizzate in una catena di dipendenze difficilmente adattabile al modello di esecuzione della GPU. Solitamente, il problema viene risolto nei compressori GPU adottando un approccio "Reduce-then-scan", che suddivide la prefix-sum in tre fasi distinte. Nella fase di *Reduce* viene calcolata la dimensione del blocco compresso per ciascun thread all'interno di un threadblock. Successivamente, una sincronizzazione globale effettua una prefix-sum tra i vari threadblock per determinare le dimensioni a livello di dispositivo. Infine, nella fase di *Scan*, tali valori sincronizzati vengono distribuiti ai singoli thread, consentendo a ciascun blocco di dati di conoscere la propria posizione all'interno del buffer di output compresso. Poiché le operazioni atomiche sulla memoria globale sono relativamente lente, l'approccio più avanzato per la sincronizzazione globale è il semplice chained-scan in cui, tramite un'implementazione seriale, ogni blocco di thread deve attendere il completamento dei precedenti prima di procedere.

cuSZp2 maschera la latenza derivante dalla chained-scan adottando una strategia di ***decoupled lookback*** [15]. L'idea principale è che ciascun threadblock esegua un'operazione di lookback verso i propri predecessori, aggregando le loro riduzioni locali finché non incontra un blocco con offset già sincronizzato. In questo processo, cuSZp2 disaccoppia le dipendenze della chained-scan seriale (Figura 5.5), sfruttando le risorse computazionali dei threadblock inattivi per calcolare in modo ridondante gli offset dei predecessori.

Nella pratica ogni threadblock ha tre possibili stati. ***Finished***: indica che la sincronizzazione al livello di device è terminata, quindi questo thread può trovarsi in una fase di local scan o di memorizzazione dei byte compressi (oppure ha finito tutte le operazioni). ***Looking back***:In questo stato, sebbene la local reduction sia completa, la sincronizzazione a livello di dispositivo non è stata ancora propagata a questo blocco di thread. Quindi sta guardando indietro ai suoi predecessori e aggregando i loro valori di local reduction. ***Waiting***: questo stato indica che la compressione o la local reduction all'interno di questo threadblock non è stata

completata. Quando un threadblock *Looking Back* aggrega uno in *Waiting*, anch'esso rimarrà in attesa fino al suo completamento.

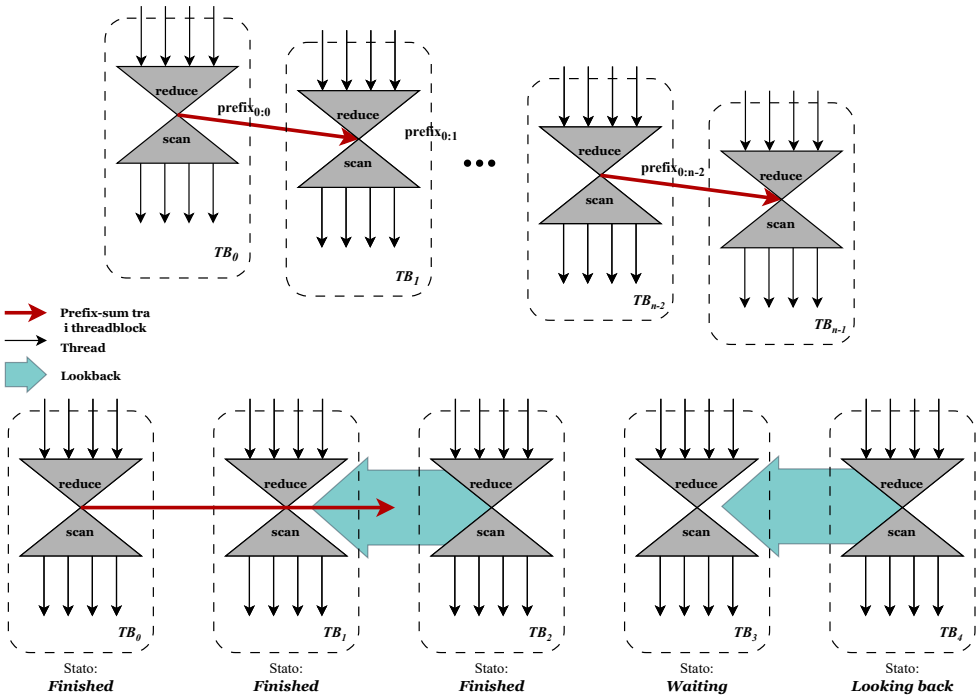


Figura 4.5. Di sopra la **chained-scan**. Sotto viene rappresentata l'esecuzione del ***decoupled lookback***, in cui il threadblock TB_2 raggiunge lo stato *Finished* tramite il lookback prima di ricevere la prefix-sum da TB_1 (ovvero prima che la freccia rossa lo raggiunga).

Capitolo 5

Operazioni collettive compresse

L'integrazione tra le comunicazioni collettive, trattate nel **paragrafo 2.4**, e le tecniche di compressione dei dati costituisce un aspetto rilevante per raggiungere nuovi picchi di prestazioni. Come anticipato nell'introduzione, nelle applicazioni MPI su larga scala, l'uso di comunicazioni collettive con messaggi di grandi dimensioni può compromettere le prestazioni complessive dell'esecuzione parallela. Per questo motivo, l'impiego della compressione rappresenta una strategia utile per ridurre l'impatto negativo sulla performance. Allo stesso tempo, l'impiego di dati compressi — in particolare nelle operazioni collettive — introduce nuove problematiche che richiedono una revisione e un adattamento degli algoritmi tradizionalmente adottati. Un ulteriore problema riguarda l'errore introdotto dalla compressione lossy, che può compromettere la qualità dei dati in output, rendendoli inadatti all'uso in simulazioni scientifiche reali.

Nel seguito verranno analizzati due framework che integrano in modo efficiente la compressione lossy all'interno delle comunicazioni collettive MPI. In particolare, il framework gZCCL sfrutta appieno la potenza computazionale delle GPU sia per la compressione dei dati che per le operazioni collettive, evitando allocazioni nella memoria host e riducendo così eventuali colli di bottiglia.

5.1 C-Coll

Riprendendo il modello di costo per l'operazione Allgather proposto da Rajeev Thakur e William Gropp [27], il tempo totale T necessario per completare una Allgather implementata con l'algoritmo ad anello può essere espresso come: $T = (p-1)(\alpha + n\beta)$. Se consideriamo che il lavoro viene effettuato comprimendo i dati, allora dobbiamo utilizzare anche c_B che esprime il costo di compressione e d_B per la decompressione di un determinato blocco di dati B (in questo caso $|B| = \left\lceil \frac{n}{p} \right\rceil$). Una prima idea algoritmica potrebbe essere quella di comprimere i dati, inviarli, ricevere dal vicino, decomprimerli, salvarli, comprimerli nuovamente ed inviarli. Seguendo questa procedura per $p-1$ passi otteniamo:

$$T = (p-1)(\alpha + n\beta + d_B + c_B) \tag{5.1}$$

Jiajun Huang et al. [9] hanno proposto con il framework C-Coll un miglioramento nell'integrazione della compressione lossy all'interno delle comunicazioni e operazioni collettive. Nel framework di *collective data movements* si osservano miglioramenti

significativi, rispetto alla CPRP2P[1], quando la compressione viene eseguita una sola volta all'inizio, e la decompressione viene rimandata alla fine dell'operazione collettiva. In particolare, nel caso di un'implementazione ad anello di `allgather`, è vantaggioso comprimere una sola volta la propria porzione di dati all'inizio, e successivamente propagare i dati compressi durante tutti i $p-1$ passaggi dell'algoritmo. Solo al termine, ciascuno dei N processi esegue $p-1$ decompressioni, riducendo così il carico computazionale complessivo durante la comunicazione; infatti

$$T_{\text{C-Coll}} = (p-1)\,(\alpha + n\beta + d_B) + c_B. \tag{5.2}$$

Al contrario della (5.1) dove è necessario comprimere $p-1$ volte, nell' algoritmo di C-Coll è necessaria una sola compressione.

Una procedura simile può essere applicata in un algoritmo di broadcast che utilizza un albero binomiale. Ci sono $\log_2 p$ fasi di scambio prima del completamento. In questo caso verrà effettuata una sola compressione dal nodo root, gli altri nodi propagheranno i dati compressi, salvandone una copia per poi decomprimerla. In questo modo ogni nodo ha un overhead di compressione e decompressione pari a $c_B + d_B$.

C-Coll contiene anche un framework per le *collective computation*. In questo caso non è possibile evitare operazioni di compressione e decompressione, poiché i dati devono essere combinati a ogni iterazione. Ad ogni passo dell'algoritmo si incorre in tre tipi di overhead: compressione/decompressione del blocco, send/recv del blocco compresso e operazione di reduce sul blocco. Tipicamente, l'overhead legato alla compressione risulta più significativo rispetto a quello di send/recv, poiché i dati compressi sono sensibilmente più piccoli rispetto alla loro versione originale. In un approccio rielaborato, l'overhead di comunicazione viene ridotto in modo sostanziale facendo attivamente polling del progresso della comunicazione durante le fasi di compressione e decompressione, con una conseguente riduzione del tempo complessivo di comunicazione. Quindi invece di comprimere l'intero buffer di dati, si divide in dei blocchi di lunghezza fissa e tra la compressione di due blocchi adiacenti si controlla attivamente il progresso della ricezione non bloccante (Figura 5.1). Dato che però i blocchi compressi non possono essere semplicemente combinati insieme, si salvano i dati compressi nello stesso buffer e si alloca una piccola porzione di memoria in testa per salvare i metadati per la decompressione.

L'approccio adottato in C-Coll, oltre a rendere più efficienti le comunicazioni collettive, fornisce una qualità dei dati ricostruiti maggiore rispetto ad algoritmi con CPRP2P. Questo perché nel framework di *collective data movement* i dati vengono compressi una sola volta, evitando la propagazione dell'errore.

Utilizzando un compressore lossy error bounded, è stato provato che l'errore aggregato derivante dalle *collective computation* rimane limitato in una certa misura dal limite sull'errore $\hat{e}$. Assumiamo che ci siano p nodi e che l'errore di compressione di ognuno di essi sia e_i, che segue una distribuzione normale $e_i \sim N(\mu_i, \sigma_i^2)$. Allora l'operazione di somma ha un errore aggregato finale nell'intervallo $\left[-2\sqrt{p}\sigma, 2\sqrt{p}\sigma\right]$ con la probabilità del 95.44%, dove σ è la varianza del limite sull'errore $\hat{e}$ del compressore lossy. Questo perché utilizziamo lo stesso limite sull'errore di compressione tra tutti i p nodi, allora l'errore aggregato avrà distribuzione $\tilde{e}_{sum} \sim N(0, n\sigma^2)$[2].

[1] Compression-enabled point-to-point communication, ovvero l'integrazione della compressine nelle comunicazioni point-to-point di MPI (send, recv).

[2] Questo perché la combinazione lineare di variabili aleatorie con distribuzione normale ha anch'essa distribuzione normale $N(\sum a_i\mu_i, \sum a_i\sigma_i^2)$, con a_i che denota delle costanti per ogni variabile.

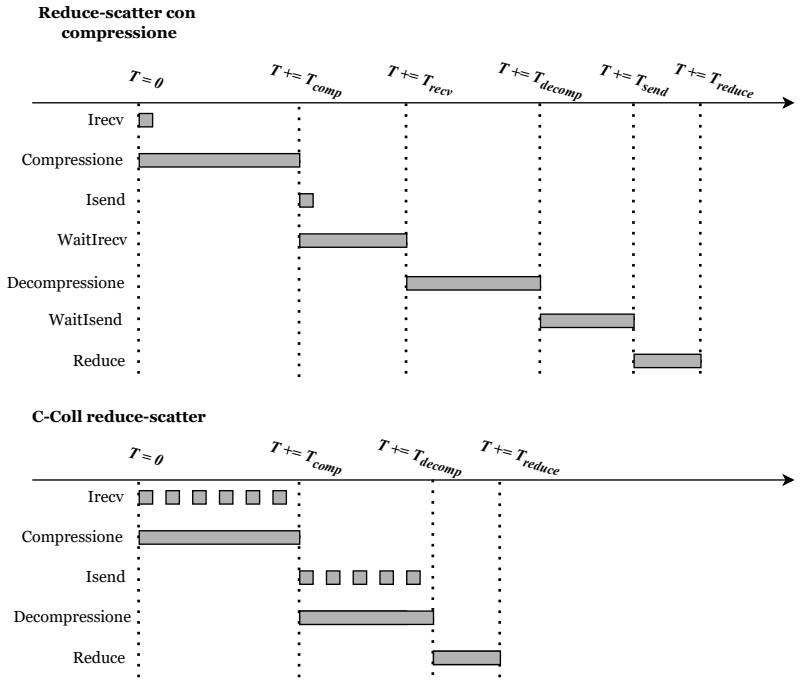


Figura 5.1. Riduzione dell'overhead dovuto alla compressione/decompressione tramite l'overlapping con le operazioni di ricezione/invio. In particolare, l'overhead legato alla ricezione e all'invio dei messaggi viene mitigato grazie al polling attivo, che consente di sovrapporre le operazioni di comunicazione con le fasi di compressione e decompressione.

Inoltre essendo che l'errore di compressione è limitato da $\hat{e}$, e l'errore per assunzione segue una distribuzione normale, possiamo approssimare $\hat{e} \approx 3\sigma$ con una precisione del 99.74%, e quindi ottenere che l'errore finale aggregato cade nell'intervallo $\left[-\frac{2}{3}\sqrt{p}\hat{e}, \frac{2}{3}\sqrt{p}\hat{e}\right]$ con una probabilità del 95.44%. Risultati simili vengono dimostrati per le operazioni di media, max e min.

5.2 gZCCL

Per testare l'efficienza del framework C-Coll è stata utilizzata una allreduce, che combina comunicazione e operazioni sui dati. L'integrazione della compressione nelle collettive ha evidenziato che il collo di bottiglia non risiede più nel trasferimento dei dati tra i nodi, bensì nell'overhead introdotto dalle fasi di compressione e decompressione (Figura 5.2). In particolare, le implementazioni GPU di queste collettive risultano penalizzate da un utilizzo inefficiente dell'hardware, dovuto ai movimenti di dati tra host e device richiesti dalla compressione. *Jiajun Huang et al.* [6] hanno proposto un nuovo framework volto a migliorare i punti deboli di C-Coll sfruttando il compressore cuSZp adattato per comunicazioni e operazioni collettive.

gZCCL è un framework sviluppato analizzando l'impatto dei compressori GPU sulle comunicazioni e sulle operazioni collettive. Dallo studio è emerso che un basso utilizzo della GPU durante l'esecuzione dei kernel di compressione comporta un aumento del costo della compressione e, conseguentemente, prestazioni subottimali delle collettive. Utilizzando un compressore GPU lossy come cuSZp, sono stati

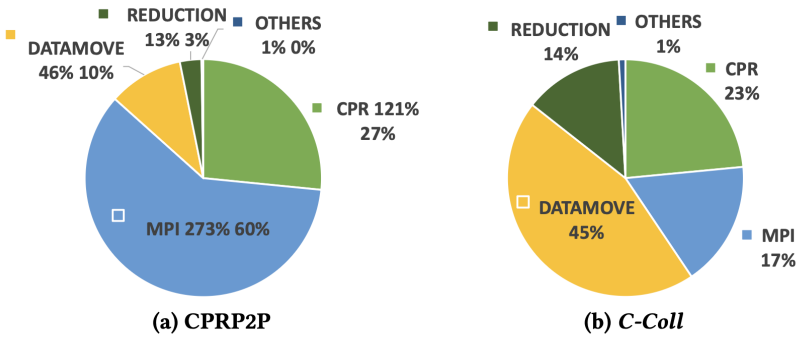


Figura 5.2. Performance di una allreduce utilizzando la compression enabled point-to-point communication (CPRP2P) confronate con il framework C-Coll: le percentuali di CPRP2P sono scalate ai tempi di esecuzione di C-Coll [6]

condotti esperimenti su messaggi di diverse dimensioni. In particolare, a partire da un dataset di 646MB, i tempi di compressione e decompressione diminuiscono progressivamente con la riduzione della dimensione dei dati, fino a raggiungere circa 5MB; al di sotto di tale soglia, per messaggi troppo piccoli, i tempi non diminuiscono ulteriormente. Questo comportamento indica che, per messaggi di dimensioni ridotte, la compressione non sfrutta appieno la potenza della GPU, introducendo un overhead non trascurabile all'interno delle operazioni collettive. Il metodo di recursive doubling garantisce una maggiore scalabilità all'aumentare del numero di processi quando i messaggi sono inferiori a 5MB, grazie a un utilizzo più intensivo della GPU. Al contrario, quando la quantità di dati per processo è sufficientemente elevata da saturare la GPU, l'algoritmo ad anello offre prestazioni migliori.

Nel framework gZCCL il compressore cuSZp è stato modificato per eliminare i trasferimenti tra device e host, che rallentano le fasi di compressione, eseguendo le allocazioni necessarie esclusivamente sul device e gestendo la pulizia dei buffer in modo da consentirne il riutilizzo in più fasi delle collettive. Inoltre, per agevolare la compressione con stream multipli—cioè per consentire la decompressione e la compressione concorrente di più blocchi sulla GPU durante le operazioni collettive—è stata introdotta la possibilità di specificare stream definiti dall'utente nei kernel di compressione e decompressione.

Per sovrapporre la compressione sui diversi stream è fondamentale evitare conflitti nell'accesso ai dati, motivo per cui in ogni stream viene allocato un buffer dedicato a memorizzare informazioni temporanee. Ad esempio, nel caso di una scatter basata su binomial tree, la procedura di compressione e decompressione multi-stream prevede l'inizializzazione di un array di stream di lunghezza p (dove p corrisponde alla dimensione del comunicatore) e la creazione, sul lato CPU, di un array per memorizzare le dimensioni dei dati compressi e i relativi offset globali per ciascun processo. In questo modo, il nodo radice può lanciare la compressione in maniera concorrente, memorizzando in sicurezza tutti i blocchi compressi—assegnando a ciascun blocco uno stream diverso—all'interno dello stesso buffer sul device, grazie all'array degli offset mantenuto sulla CPU. Al termine della compressione, una sincronizzazione garantisce che tutti gli stream abbiano completato l'operazione. Successivamente, mediante operazioni di memcpy asincrone con stream differenti, i dati compressi vengono trasferiti basandosi sugli offset calcolati (archiviati in un ulteriore buffer sul device) per poi essere inviati. Infine, i buffer vengono distribuiti utilizzando il *binomial tree* e ciascun processo procede con la decompressione del

proprio buffer utilizzando uno stream diverso da quello di default.

5.3 Eliminare il DOC workflow

Le strategie adottate nei due framework analizzati per ottimizzare le prestazioni delle operazioni collettive su dati compressi sono le seguenti:

- Minimizzare il numero di compressioni e decompressioni, riorganizzando opportunamente gli algoritmi delle collettive.
- Sovrapporre le fasi di compressione e decompressione con quelle di comunicazione e calcolo, per mascherarne il costo.
- Ridurre al minimo i trasferimenti di dati tra host e device, sfruttando al massimo le capacità computazionali della GPU.

Nonostante l'impiego di CUDA *streams* e la progettazione di un design ad-hoc che renda l'operazione collettiva fortemente GPU-centrica, l'approccio **DOC** (Decompression–Operation–Compression), che prevede la decompressione dei dati prima dell'operazione collettiva e la successiva ricompressione prima dell'invio, risulta spesso poco vantaggioso nella pratica. Questo perché introduce overhead significativi che possono annullare i benefici della compressione. I due framework che seguono mirano a eliminare il DOC workflow e proporre un modo più leggero per combinare tra loro i dati nelle varie fasi della collettiva. L'eliminazione del DOC workflow è resa possibile grazie all'impiego della **compressione omomorfica**. Formalmente, una procedura di compressione omomorfica trasforma una sequenza di oggetti $\{x_i\}_{i=1}^n$, su cui è definito un operatore binario $*$, nella loro forma compressa $\{x_i^c\}_{i=1}^n$, sulla quale è possibile eseguire operazioni tramite un altro operatore $\star$. Se la compressione omomorfica è *lossless*, allora vale la seguente uguaglianza:

$$(x_1 * x_2 * \cdots * x_n)^c = x_1^c \star x_2^c \star \cdots \star x_n^c. \tag{5.3}$$

In altre parole, la trasformazione conserva la struttura algebraica dei dati (proprio come un *omomorfismo*), permettendo di eseguire operazioni direttamente sulle rappresentazioni compresse e ottenendo lo stesso risultato che si otterrebbe operando sui dati originali non compressi. Il vantaggio di questa procedura è il poter operare direttamente sui dati compressi, evitando la decompressione e compressione ad ogni passo della collettiva.

5.4 hZCCL

Il framework hZCCL [8] è una prima implementazione della compressione omomorfica su CPU. Per la trattazione che segue sarà sufficiente conoscere la procedura con cui si elimina il DOC workflow all'interno delle collettive. Algoritmicamente, il compressore proposto in hZCCL è simile a cuSZp: il buffer viene suddiviso in blocchi tra i vari thread, in questo caso però ad ogni blocco di thread viene assegnato un blocco contiguo; poi su ogni blocco si effettua la quantizzazione, la prediction e si estrae il valore assoluto memorizzando il segno in una *sign map*; in seguito si esegue una fixed-length encoding per ogni blocco. Quindi i dati alla fine della compressione hanno la stessa forma di quelli ottenuti da cuSZp.

Il compressore omomorfico *hZ-dynamic*, invece, riceve in input due blocchi già compressi e li combina direttamente, producendo in output la loro versione

compressa risultante. A differenza della compressione omomorfica definita nel paragrafo precedente, l'approccio adottato in **hZ-dynamic** non opera interamente nel dominio compresso, ma prevede una **decompressione parziale** dei dati, seguita dalla loro combinazione e da una successiva **ricompressione parziale**.

Dopo la fase di compressione, i dati sono memorizzati come *codeword* a lunghezza fissa. In questo contesto, hZ-dynamic esegue una **Inverse Fixed-Length Encoding (IFE)**, ovvero decodifica le codeword ottenute con la *fixed-length encoding*, per ottenere la rappresentazione parzialmente decompressa, recupera il *segno* dalla mappa dei segni, combina i dati in base all'operazione richiesta, quindi salva i segni aggiornati e infine esegue una *sincronizzazione globale* seguita da una nuova *fixed-length encoding*.

Nel framework **hZCCL** sono state introdotte ulteriori ottimizzazioni per adattare il processo al blocco di dati su cui si sta lavorando. Si considerino, ad esempio, due blocchi compressi B_1^c e B_2^c, con lunghezze di codifica rispettivamente x e y, che devono essere combinati in una fase di *reduce-scatter*. Durante la compressione, ogni *thread* verifica se il blocco corrente è un *blocco di zeri*. Se $x = 0$ e $y \neq 0$, viene salvato solo B_2^c, evitando sia la combinazione degli elementi sia la decompressione parziale del blocco nullo; se $y = 0$ e $x \neq 0$, si procede in modo analogo salvando solo B_1^c. Quando invece entrambi i blocchi hanno lunghezza diversa da zero ($x \neq 0$ e $y \neq 0$), si applica la **IFE** a ciascun blocco e si esegue la combinazione elemento per elemento.

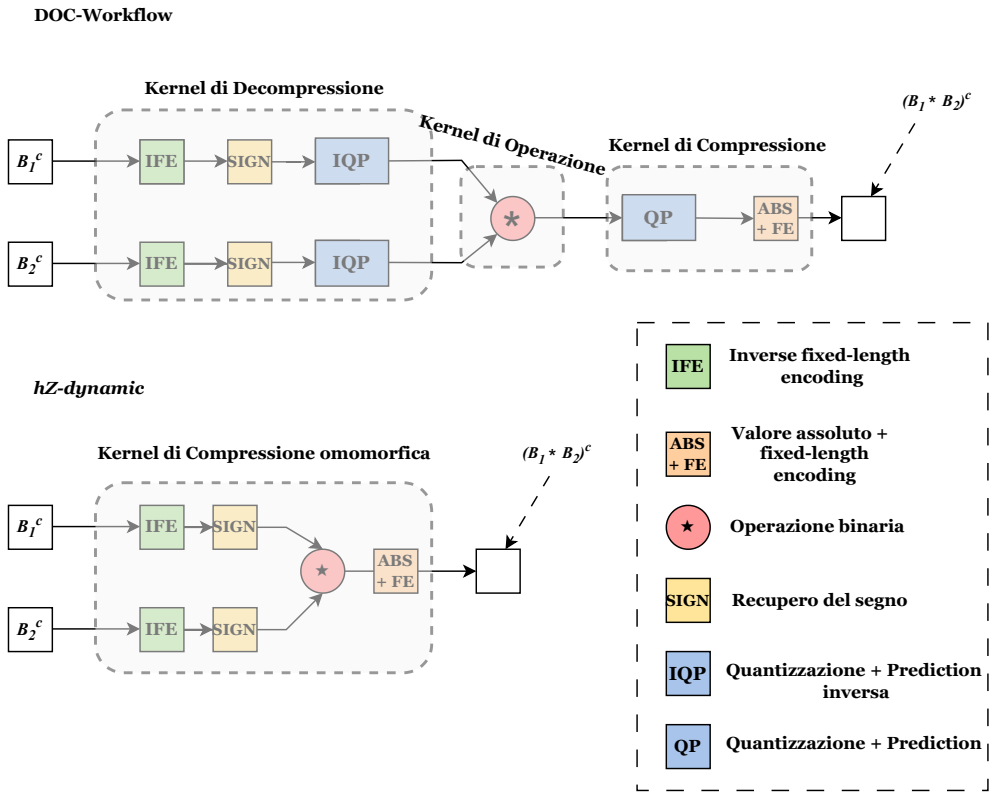


Figura 5.3. Confronto tra DOC-workflow e compressione omomorfica (*hZ-dynamic*).

Analizziamo adesso l'impatto che la compressione omomorfica di hZCCL ha

sul modello di costo per le operazioni collettive. Sia h_B il costo in termini di tempo per processare il blocco B con la compressione omomorfica. L'operazione di *reduce-scatter* con dati compressi e con DOC-workflow ha costo [3]:

$$T^{\text{RS}}_{\text{C-Coll}} = (p-1) \times \left\{ c_B + d_B + \frac{n}{p}\gamma + \alpha + \frac{n}{p}\beta \right\} \tag{5.4}$$

dove $c_B + d_B + \frac{n}{p}\gamma$ rappresenta il costo del DOC-workflow ad ogni passo dell'algoritmo. Infatti, ogni processo decomprime B con costo d_B, combina con la porzione di buffer locale che gli corrisponde in tempo $\frac{n}{p}\gamma$ e ricomprime con costo c_B.[4]

Se invece consideriamo una reduce-scatter in cui il DOC-workflow viene eliminato grazie alla compressione omomorfica, allora il costo di una reduce-scatter diventa

$$T^{\text{RS}}_{\text{hZCCL}} = (p-1) \times \left\{ h_B + \alpha + \frac{n}{p}\beta \right\} + p \times c_B + d_B \tag{5.5}$$

In questo caso, nella prima fase, ogni processo comprime tutti e p i blocchi in cui il messaggio è stato suddiviso. L'intero DOC-workflow viene sostituito dalla compressione omomorfica con costo h_B che nella pratica supponiamo essere inferiore a $c_B + d_B + \frac{n}{p}\gamma$. Nella (5.5) bisogna tenere conto che le compressioni dei vari blocchi possono essere effettuate in modo concorrente, quindi in generale $T^{\text{RS}}_{\text{hZCCL}} < T^{\text{RS}}_{\text{C-Coll}}$.

Inoltre, la *reduce-scatter* di hZCCL, combinata con una *allgather*, consente di ottenere una *allreduce* che esegue una compressione in meno: infatti, al termine della *reduce-scatter*, è possibile mantenere il blocco di dati compresso risultante dall'ultima compressione omomorfica e passarlo direttamente come input alla prima fase dell'algoritmo di *allgather*.

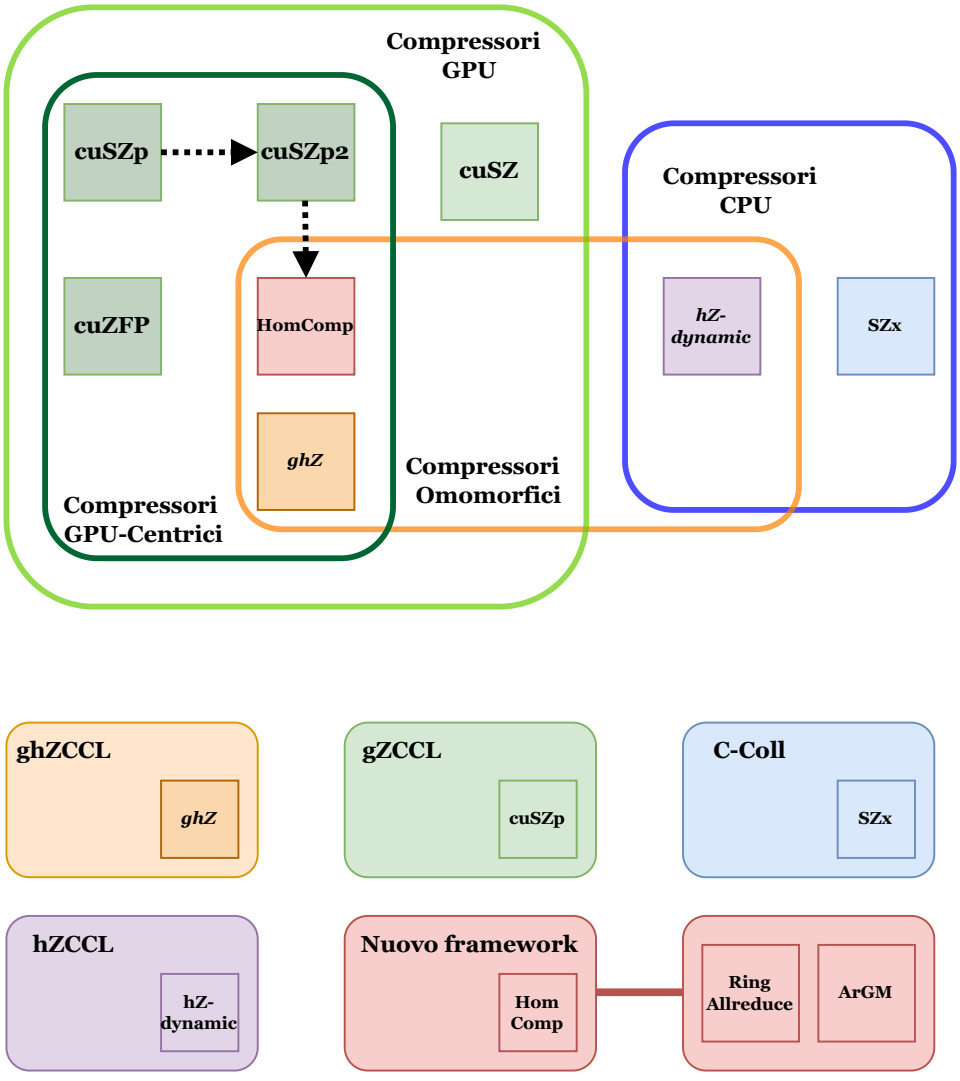
[3] Come nei capitoli precedenti consideriamo n la dimensione totale del messaggio e p il numero di processi.

[4] Il fattore n/p presente in β non è fisso, ma varia in relazione del compressore utilizzato. Nel caso peggiore (*compression factor = 1*), il numero di byte da inviare coincide con n/p; tuttavia, in generale, la dimensione del blocco compresso risulta molto inferiore ($\ll n/p$). Un'alternativa nel modellare tale dimensione consiste nel considerare $\text{blockSize}_{\max} = \max |B^c|$ e moltiplicarlo per la costante β.

Parte III

Allreduce con compressione omomorfica

In questa terza parte viene introdotto un nuovo framework di collettive che integrano il compressore omomorfico `HomComp`. Come mostrato nello schema riportato di seguito, esso si colloca tra i compressori *GPU-centrici*, omomorfici e a singolo kernel. Nei capitoli successivi verranno illustrati nel dettaglio lo schema di compressione adottato da `HomComp` e le collettive da esso implementate, insieme ad un'analisi comparativa del modello di costo delle nuove collettive rispetto al framework hZCCL e ghZCCL. Infine, verranno presentati i risultati sperimentali, mettendo a confronto le performance del nuovo framework con quelle delle collettive dello standard MPI.



Panoramica dei compressori [26] e dei framework discussi. Le frecce tratteggiate indicano che il compressore con la freccia entrante deriva da quello con la freccia uscente.

Capitolo 6

Ring allreduce con compressione omomorfica

In questa sezione vengono presentati i principali risultati conseguiti durante il tirocinio. L'obiettivo principale era implementare in modo efficiente la compressione omomorfica su GPU, prendendo come riferimento il modello su CPU proposto da hZCCL. A tal fine, è stato sviluppato un nuovo algoritmo che introduce la compressione omomorfica nelle operazioni collettive di reduce-scatter e allreduce.

Per la realizzazione del compressore è stato impiegato cuSZp2, che attualmente garantisce le migliori prestazioni su GPU e riprende in parte lo schema di compressione del compressore hZ-dynamic di hZCCL.

Il lavoro svolto si articola in due punti principali:

- **Compressore omomorfico**: sono state introdotte modifiche sostanziali al compressore (cuSZp2). In particolare, sono stati sviluppati tre kernel specifici per la compressione omomorfica e sono stati aggiunti meccanismi di controllo per garantirne l'affidabilità anche con dataset di grandi dimensioni. Il nuovo compressore non rappresenta un semplice porting di hZCCL su GPU, ma una sua reinterpretazione, progettata per integrarsi in modo naturale con la *ring allreduce*, evitando colli di bottiglia.
- **Operazioni collettive**: vengono proposti tre algoritmi che combinano la *ring allreduce* con la compressione omomorfica, utilizzando diverse strategie derivanti dai nuovi kernel.

Infine è stato implementato un algoritmo gerarchico ibrido adatto per sistemi di calcolo distribuito che prevede comunicazioni tra inter-nodo con dati compressi, e comunicazioni con dati non compressi all'interno dello stesso nodo.

6.1 Compressione omomorfica su GPU

Con l'obiettivo di implementare la compressione omomorfica su GPU, sono state utilizzate come riferimento le funzioni di compressione e decompressione di cuSZp2. Sono state realizzate due versioni distinte dello stesso schema di compressione, motivate da differenti considerazioni algoritmiche.

Prima versione (`HomComp`) (Figura 6.1): prevede l'impiego di due *kernel*, in cui l'output del primo costituisce l'input del secondo. In sintesi:

- il primo *kernel* esegue una compressione parziale di un *chunk* di dati locale;
- il secondo *kernel* riceve in input sia il *chunk* compresso parzialmente sia un *chunk* compresso proveniente da un altro processo. All'interno di questo secondo *kernel*, il blocco compresso viene parzialmente decompresso per poter essere combinato con il *chunk* locale compresso parzialmente.

Seconda versione (`HomComp`$_F$) (Figura 6.2): utilizza un unico *kernel* che integra le operazioni svolte dai due *kernel* della prima versione.

In `HomComp` l'algoritmo di compressione è identico a quello utilizzato in cuSZp2. Il *kernel* di compressione parziale, tuttavia, applica soltanto una parte delle trasformazioni: sull'intero blocco vengono eseguite la **quantizzazione** e la **prediction** (**QP**). L'output di questa procedura è quindi un array di interi contenente i dati parzialmente compressi.

Nel secondo *kernel*, i dati compressi vengono decompressi parzialmente tramite *inverse fixed-length encoding*, assegnando a ciascun elemento il proprio segno, precedentemente salvato nella *mappa dei segni* durante la fase di compressione. Contestualmente, i dati parzialmente compressi vengono combinati con quelli in fase di decompressione. Al termine, si salvano le informazioni necessarie, viene eseguita la sincronizzazione globale e, successivamente, si applicano la *fixed-length encoding* e il *block bit-shuffle*. Nella Figura 6.1 viene fornito un esempio schematico di esecuzione di `HomComp` all'interno di un'operazione collettiva in cui il processo con *rank j* riceve un blocco compresso $Comp_{j-1}$ dal *rank j-1* da combinare con il blocco locale $Data_j$. Per migliorare le prestazioni su GPU, in cuSZp2 il buffer

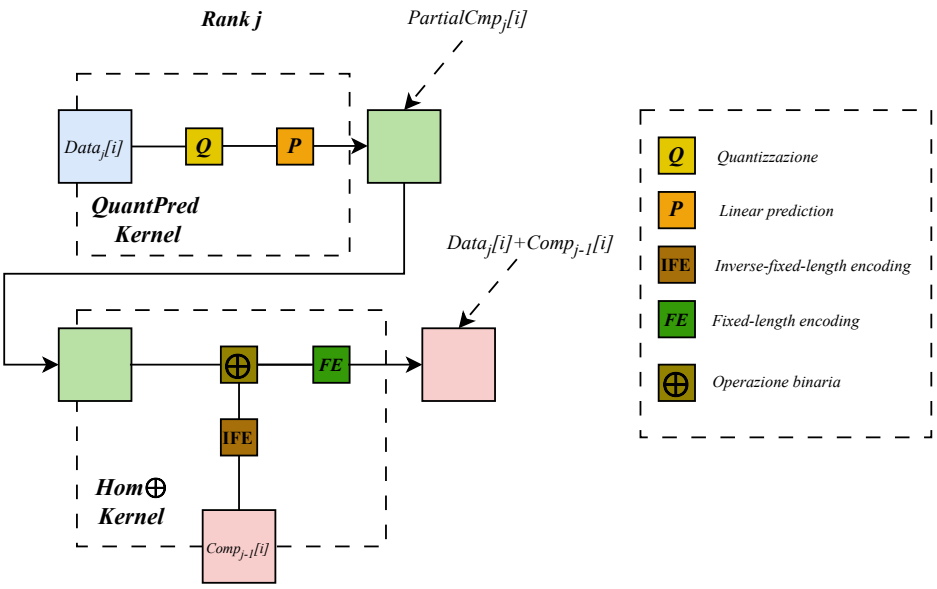


Figura 6.1. Schema di compressione di `HomComp`.

da comprimere viene suddiviso in sottoblocchi elaborati in modo indipendente da ciascun thread. In particolare, ogni *threadblock* è composto da un solo *warp* (32 thread), i quali accedono al blocco da comprimere in maniera coalescente, a gruppi

di 32 elementi, fino a un totale di 1024 elementi per thread e 32768 elementi per warp.

La suddivisione in sottoblocchi consente di sfruttare in modo più efficiente il *device*, incrementando significativamente il *memory throughput*. Oltre a questa suddivisione ottimizzata, cuSZp2, come descritto nel paragrafo 4.3, utilizza caricamenti e scritture in memoria tramite `float4`, il che può causare problemi di accesso alla memoria se il dataset non possiede la dimensione corretta. Per questo motivo è necessario verificare che il dataset abbia dimensioni compatibili prima di utilizzare il compressore; ciò può essere garantito aggiungendo del padding prima di comprimere.

Nella versione $\texttt{HomComp}_F$ a singolo *kernel*, le operazioni di quantizzazione e prediction vengono eseguite durante la fase di decompressione del blocco da parte di ciascun thread. In questo modo non è necessario memorizzare in memoria globale una copia intermedia dell'array di dati parzialmente compressi ($PartialCmp_j$ in Figura 6.1). Infatti, la compressione parziale viene effettuata all'interno dello stesso ciclo in cui gli elementi vengono combinati, rendendo sufficiente la memorizzazione temporanea dei dati unicamente nei registri. Inoltre, la fusione dei due *kernel* in un unico kernel consente di eliminare l'overhead legato all'inizializzazione del kernel aggiuntivo. Ogni chiamata a kernel incorpora un costo di lancio non trascurabile, tipicamente nell'ordine di 10–20 µs, che si somma se si utilizzano più kernel. Riducendo la frequenza dei kernel attivi (*kernel fusion*), è possibile migliorare significativamente le prestazioni, minimizzando le latenze e massimizzando l'utilizzo delle risorse GPU.

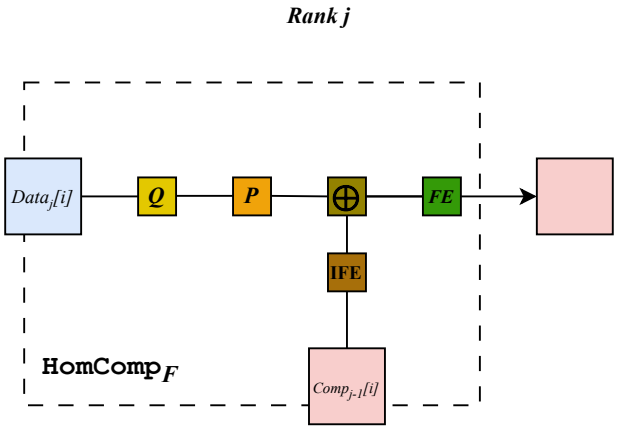


Figura 6.2. Schema di esecuzione di $\texttt{HomComp}_F$.

6.2 Ring allreduce con compressione omomorfica

`HomComp` è stato pensato per essere integrato in modo efficiente dalle operazioni collettive che utilizzano algoritmi ad anello o che, più in generale, richiedono la combinazione di un blocco del buffer locale con uno ricevuto da un altro processo. Nella pratica è necessario che ad ogni passo dell'algoritmo esegua un' operazione del tipo

$$C_p(\text{LocBuf}) \oplus D_p(\text{RecvBuf}) \tag{6.1}$$

Dove C_p sarebbe la compressione parziale e D_p la decompressione parziale descritte nel paragrafo precedente, $\oplus$ un'operazione binaria generica, mentre LocBuf e RecvBuf sono rispettivamente il buffer locale e quello ricevuto da un altro processo.

In questo tipo di collettiva l'integrazione di `HomComp/HomComp`$_F$ risulta abbastanza naturale: all' i-esimo passo dell'algoritmo il rank j applica `HomComp/HomComp`$_F$ al $[j - i + 1 \bmod size]$ blocco locale in cui è stato suddiviso il buffer originale e al blocco ricevuto dal processo con rank $j - 1$.

Nel caso di `HomComp` si possono adottare due approcci differenti per eseguire la fase di *reduce-scatter* ad anello.

Il primo approccio (Figura 6.3) prevede l'allocazione di un buffer di interi di dimensioni pari al dataset locale del processo. Prima dell'inizio dello scambio dei messaggi, viene eseguito il kernel di QP sull'intero dataset e il risultato viene salvato nel buffer. Successivamente, ogni processo procede con la compressione del proprio blocco da inviare al vicino e predispone un'operazione di ricezione non bloccante in un buffer temporaneo. Ad ogni passo dell'algoritmo sarà quindi necessario applicare soltanto il kernel che produce D_p(RecvBuf) e lo combina con la porzione di buffer di interi C_p(LocBuf) generata nella fase iniziale.

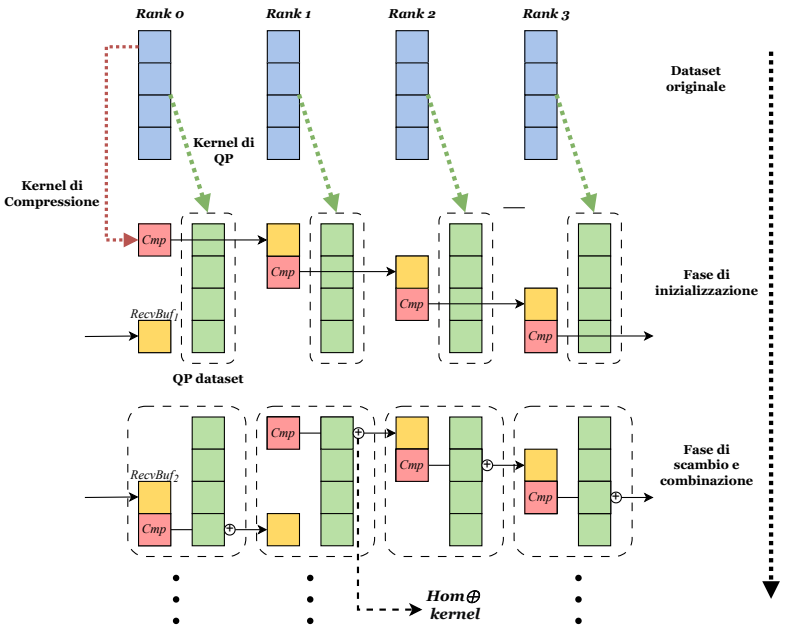


Figura 6.3. Con la prima integrazione di `HomComp` nella reduce-scatter durante la fase di inizializzazione ogni processo lancia il kernel di QP sull'intero dataset (ottenendo *QP dataset*) e comprime un chunk dello stesso (*Cmp*). Poi riceve in modo alternato i dati all'interno di due buffer ($RecvBuf_1$ e $RecvBuf_2$).

Il secondo approccio sfrutta i CUDA stream per sovrapporre l'esecuzione del kernel di QP con i kernel di compressione e di compressione omomorfica. Nel primo step dell'algoritmo ad anello originale vengono eseguite tre operazioni: compressione del primo blocco da inviare, `send` del blocco compresso e ricezione non bloccante del blocco compresso. Alle iterazioni successive, si eseguiranno la `MPI_Wait` relativa alla `Irecv`, i due kernel della compressione omomorfica e la `send` del blocco compresso aggregato.

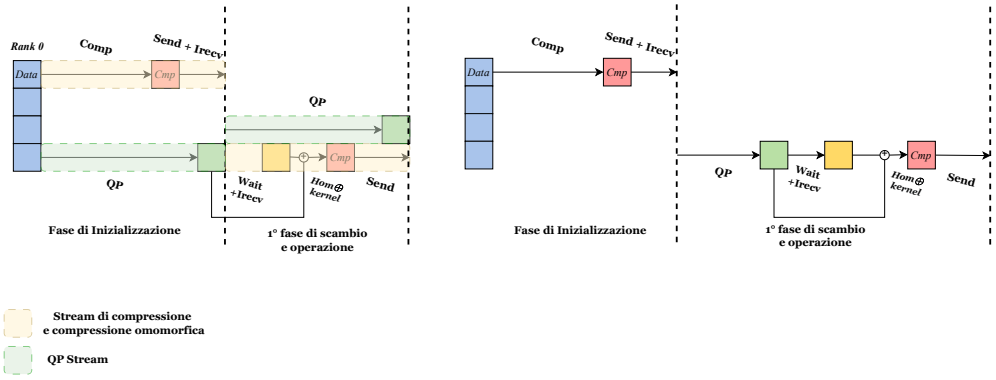


Figura 6.4. A sinistra è rappresentata l'esecuzione dei primi due passi della reduce-scatter con l'utilizzo di CUDA stream con l'esecuzione anticipata del kernel di QP. A destra invece è rappresentata l'esecuzione su stream default.

Per ottimizzare questo flusso di istruzioni, è possibile **anticipare**, durante la fase di inizializzazione, la quantizzazione e prediction del primo blocco, che verrà combinato nell'iterazione immediatamente successiva.

Questo si ottiene creando due CUDA stream: il primo stream gestisce il flusso di lavoro dedicato alla compressione iniziale e alla compressione omomorfica, mentre il secondo stream è dedicato esclusivamente ai kernel di QP, che vengono lanciati con un'iterazione di anticipo. In questo modo, ad ogni passo, la compressione omomorfica dovrà attendere soltanto un overhead minimo per ottenere i dati parzialmente compressi. Inoltre, questo secondo approccio risulta meno dispendioso in termini di memoria: richiede infatti soltanto l'allocazione di due buffer, ciascuno della dimensione di un blocco del dataset, utilizzati in modo alternato per memorizzare gli interi generati tra una fase e l'altra durante l'esecuzione del kernel di QP.

Essendo il kernel di QP relativamente leggero rispetto a quelli di compressione e di compressione omomorfica, risulta particolarmente adatto all'esecuzione concorrente mediante la strategia di anticipazione, e non introduce un overhead significativo neppure se applicato all'intero dataset, come avviene nel primo metodo.

Utilizzando la versione $\texttt{HomComp}_F$ non è necessario allocare memoria aggiuntiva perché, come detto in precedenza, la fase di QP e compressione omomorfica sono eseguite nello stesso kernel e utilizzano dei registri temporanei. Questo ci permette di utilizzare ad ogni fase direttamente $\texttt{HomComp}_F$ senza manipolazioni intermedie sui dati.

Il vantaggio di questa tipologia di *reduce-scatter* si estende anche all'implementazione della *allgather* compressa, utilizzata per completare la *allreduce*. Quest'ultima, infatti, beneficia del fatto di non dover eseguire ulteriori compressioni, poiché utilizza direttamente l'output prodotto nell'ultima fase della *reduce-scatter* con compressione omomorfica.

6.3 Allreduce gerarchica mista

Le applicazioni scientifiche che necessitano di operazioni collettive compresse — solitamente perché generano una quantità enorme di dati — vengono eseguite su sistemi di calcolo multinodo. All'interno di ciascun nodo, tipicamente sono presenti più *device*. Possiamo quindi distinguere tra comunicazioni **intra-nodo**, ovvero tra

device nello stesso nodo, e comunicazioni **inter-nodo**, ovvero tra device di nodi differenti.

I device che comunicano all'interno di uno stesso nodo possono spesso raggiungere velocità di trasferimento molto più elevate rispetto a quelle possibili nelle comunicazioni inter-nodo. Come discusso nei capitoli precedenti, il vero collo di bottiglia è rappresentato dalle interconnessioni, ed è proprio in questo contesto che la compressione può risultare determinante.

Se, invece, all'interno di uno stesso nodo disponiamo di d *device* completamente connessi, la compressione dei dati non sempre si traduce in un miglioramento delle prestazioni. Per questo motivo, una *allreduce* compressa come quella descritta in precedenza, eseguita su un insieme di nodi in cui ogni nodo partecipa con più processi (e quindi più *device*)[1], non garantisce un utilizzo efficiente dei collegamenti intra-nodo. Ciò accade perché l'overhead introdotto dalle fasi di compressione e decompressione non viene compensato da un incremento sufficientemente significativo della velocità di trasferimento.

In questo contesto, introduciamo una *allreduce* gerarchica mista (Figura 6.5):

- **Gerarchica** perché composta da due fasi: una *allreduce* inter-nodo e una *allreduce* intra-nodo;
- **Mista** perché la *allreduce* inter-nodo viene eseguita su dati compressi, mentre la *allreduce* intra-nodo viene eseguita su dati in forma originale.

La procedura viene divisa in tre fasi:

Reduce-scatter intra-nodo (1): In questa fase viene creato un cominicatore intra-nodo in cui ogni processo sullo stesso nodo ha rank in $[d]$ dove d è il numero di *device* nel nodo. Se i collegamenti tra i device interni al nodo lo permettono viene effettuata una reduce-scatter "leggera": ogni processo divide il messaggio in dei blocchi più piccoli e all'intrno di un ciclo inizia ad inviare, con una send non bloccante, l'i-esimo blocco all'i-esimo processo. Contestualmente il j-esimo processo esegue una serie di `Irecv` (in totale $d-1$) per ricevere il j-esimo blocco da tutti gli altri processi. Una volta ricevuti i contributi di tutti i processi effettua un kernel per l'aggregazione contemporanea dei $d-1$ blocchi ricevuti insieme a quello locale. Ad esempio nel caso in cui $d = 4$ ogni processo invia 3 blocchi, ne riceve 3 e combina i 3 blocchi ricevuti alla porzione di messaggio salvata in locale. Tutto questo processo è effettuato senza l'allocazione di buffer aggiuntivi e sovrapponendo le varie chiamate di invio e ricezione, sincronizzandole solo prima della somma.

Allreduce con compressione omomorfica internodo (2): Al termine della fase precedente ogni processo con rank i nel comunicatore intra-nodo possiede l'i-esimo blocco del messaggio originale combinato con il contributo di tutti i processi all'interno del suo nodo. Quindi si creano d **comunicatori internodo**, dove nel j-esimo comunicatore, con $j \in [d]$, viene assegnato il processo con rank j nel **comunicatore intra-nodo**. Ognuno di questi comunicatori esegue una allreduce con compressione omomorfica. Alla fine di questa fase ogni processo all'interno di ogni nodo possiede $1/d$ del messaggio con il contributo di tutti e p i processi totali. Quindi ogni processo con rank i nel comunicatore intra-nodo (oppure ogni processo all'interno dell' i-esimo comunicatore inter-nodo) ha l'i-esimo blocco *completato*.

[1] Si assume che ogni processo abbia associato un *device* distinto, quindi se più processi risiedono nello stesso nodo, ciascuno utilizza un *device* diverso. Inoltre per l'algoritmo che segue assumiamo che ogni nodo abbia lo stesso numero di *device*.

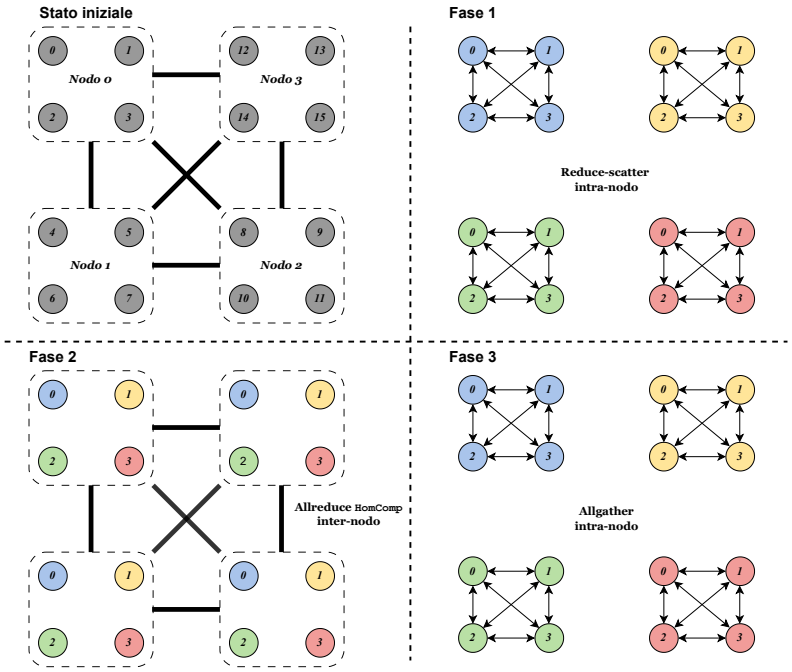


Figura 6.5. Allreduce gerarchica mista, in cui le linee nere rappresentano le interconnessioni. Nello stato iniziale i numeri rappresentano il rank nel comunicatore globale, in (1), (2) e (3) i numeri sono il rank nel comunicatore intra-nodo. I colori invece distinguono in (1) il comunicatore intra-nodo (insieme ai rank), mentre in (2) i processi con stesso colore sono all'interno dello stesso comunicatore inter-nodo.

Allgather intra-nodo (3): A questo punto si può effettuare una allgather intra-nodo in cui ogni processo invia la sua porzione completa a tutti gli altri $d-1$ processi e allo stesso tempo riceve i $d-1$ blocchi completi completando la allreduce.

6.4 Modello di costo

Con lo stesso modello utilizzato per lo studio delle collettive presentate in precedenza, analizziamo ora il costo degli algoritmi descritti sopra. Indichiamo con h_B il costo di compressione del blocco B tramite compressione omomorfica[2] in $\texttt{HomComp}_F$, con h'_B il costo di compressione del blocco B in `HomComp`, e con qp_B il costo del kernel di QP sul blocco B.

Con $\texttt{HomComp}_F$ è necessaria un'unica compressione iniziale, di costo c_B, relativa al primo blocco inviato. Nei passi successivi, sarà sufficiente applicare la compressione omomorfica tra il blocco locale e quello ricevuto:

$$T^{RS}_{\texttt{HomComp}_F} = (p-1)\left\{h_B + \frac{n}{p}\beta + \alpha\right\} + c_B + d_B. \tag{6.2}$$

[2]In modo analogo, potremmo introdurre h come costo di compressione per byte e moltiplicarlo per n/p, assumendo sempre che ogni blocco inviato e ricevuto abbia la stessa dimensione.

Confrontando $T^{RS}_{\texttt{HomComp}_F}$ con T^{RS}_{hZCCL} (Eq. 5.4), osserviamo che la reduce-scatter proposta in hZCCL richiede $p-1$ compressioni aggiuntive:

$$T^{RS}_{\text{hZCCL}} - T^{RS}_{\texttt{HomComp}_F} = (p-1) \times c_B. \tag{6.3}$$

Tuttavia, non possiamo fermarci a questa semplice differenza per concludere che `HomComp` renda la collettiva intrinsecamente più efficiente: sono necessarie ulteriori considerazioni. In primo luogo, i costi delle due compressioni omomorfiche non sono direttamente confrontabili, poiché le operazioni svolte sono differenti: in $\texttt{HomComp}_F$ si esegue una compressione parziale sui dati locali e una decompressione parziale sui dati ricevuti; in hZCCL, invece, si effettuano due decompressioni parziali (su entrambi i blocchi) ed è richiesta anche la compressione iniziale dell'intero buffer. In secondo luogo, è utile confrontare la compressione parziale (QP) e la decompressione parziale. La QP viene eseguita direttamente all'interno del loop di elaborazione degli elementi, mentre nella decompressione parziale è necessario mappare le *codeword* in interi senza segno, da cui estrarre il segno mediante la *sign map*, operazione che richiede inoltre una sincronizzazione globale per ricavare gli offset del blocco da decomprimere. Sebbene tale sincronizzazione globale possa essere eseguita in parallelo per entrambi i blocchi compressi, la QP rimane un'operazione molto più semplice e, soprattutto, evita $p-1$ compressioni aggiuntive. Queste ultime non contribuiscono direttamente alla collettiva, ma servono unicamente a trasformare i dati in un formato compatibile con il compressore omomorfico.

Consideriamo ora una reduce-scatter con `HomComp`. Nel primo approccio, ogni processo esegue inizialmente il kernel QP sull'intero buffer con costo $p \times \text{qp}_B$; nei round successivi, applica la compressione omomorfica con costo $h'B$:

$$T^{RS1}_{\texttt{HomComp}} = (p-1)\left\{h'_B + \frac{n}{p}\beta + \alpha\right\} + p \times \text{qp}_B + c_B + d_B. \tag{6.4}$$

Utilizzando invece CUDA stream, il costo diventa:

$$T^{RS2}_{\texttt{HomComp}} = (p-1)\left\{h'_B + \frac{n}{p}\beta + \text{qp}_B + \alpha\right\} + c_B + d_B. \tag{6.5}$$

Il confronto tra (6.4) e (6.5) evidenzia che nella prima viene eseguito un kernel QP non necessario, relativo al blocco che sarà poi inviato. La seconda versione elimina tale ridondanza e, grazie all'impiego degli stream, riduce ulteriormente l'overhead di qp_B. È opportuno notare che in (6.4) il kernel QP è unico sull'intero buffer, e andrebbe quindi modellato come $\text{qp} \times n$ piuttosto che come $\text{qp}_B \times p$, dato che $\text{qp}_B \times p > \text{qp} \times n$. Infine, considerando l'overhead dei kernel multipli e la mancata rappresentazione dell'overlap indotto dagli stream, possiamo assumere che $h_B < h'_B + \text{qp}_B$.

Capitolo 7

Valutazione e risultati

Dopo aver presentato il compressore omomorfico `HomComp` e la sua integrazione nell'algoritmo di *allreduce*, il seguente capitolo è dedicato alla valutazione delle prestazioni della nuova collettiva. L'analisi si focalizza principalmente sui tempi di esecuzione, confrontando la versione di *allreduce* integrata con `HomComp` e l'implementazione ad anello di `MPI_Allreduce`. Sono inoltre presentati ulteriori confronti con una *ring allreduce* GPU-centrica, utilizzata come modello di riferimento per la costruzione della versione con `HomComp`[1], oltre a un'analisi a livello più alto con ghZCCL, focalizzata sul confronto tra le idee algoritmiche e i modelli di costo[2]. La valutazione delle performance è stata effettuata sul supercomputer *Leonardo* [29, 4], costituito da due moduli di calcolo distinti. Gli esperimenti hanno utilizzato il modulo *Booster*, caratterizzato da un'architettura ottimizzata per applicazioni parallele che sfruttano intensivamente le GPU. A partire dall'architettura dei nodi di *Leonardo*, sono stati analizzati anche i tempi di esecuzione della *allreduce gerarchica mista* confrontandoli con quelli delle altre versioni. Gli studi condotti in C-Coll mostrano che, nelle operazioni collettive, l'errore aggregato rimane contenuto con altissima probabilità (95,44 % nel caso della somma). Questo risultato si estende anche a `HomComp`, che si basa sul compressore cuSZp2, caratterizzato da una garanzia di error-boundedness. Allo stesso modo, *il compression rate* di `HomComp` coincide con quello di cuSZp2, poiché entrambi adottano lo stesso schema di compressione.

7.1 Setup di valutazione e architettura di *Leonardo*

La valutazione delle diverse varianti di *allreduce* verrà condotta utilizzando dataset artificiali progettati per riprodurre, in maniera approssimata, i dati generati durante le iterazioni di simulazioni scientifiche o durante il training di LLM. I dataset sono costituiti da valori in formato `float`, generati secondo una distribuzione normale oppure tramite *random walk*, al fine di imitare la struttura tipica di dataset scientifici caratterizzati da similarità spaziale.

Le collettive verranno testate con dataset di dimensioni variabili, da *1 mln* ($\approx$ 4MB) fino a *500 mln* di *float* ($\approx$ 2GB). Le dimensioni dei dataset generati sono in un range di dimensione ragionevole dato che l'algoritmo ad anello diminuisce ulteriormente la dimensione dei messaggi inviati e osservando che messaggi troppo piccoli non beneficiano della compressione. Le prestazioni ottenute verranno innanzi-

[1] Questo confronto aggiuntivo consente di ottenere una stima più accurata delle prestazioni della nuova collettiva.

[2] Non è possibile effettuare un confronto diretto tra le due implementazioni, poiché il codice non è disponibile pubblicamente.

tutto confrontate con quelle della versione standard MPI (`MPI_Allreduce`) e con la versione GPU-centrica di *ring allreduce* implementata sopra MPI (`GPU_Allreduce`). Successivamente, le nuove collettive proposte saranno confrontate tra loro utilizzando dataset di dimensioni differenti.

In aggiunta, analizzeremo il comportamento delle applicazioni variando il numero di nodi allocati[3]. Nei test riguardanti la *allreduce* con `HomComp`, ad ogni processo verrà assegnato esattamente un nodo allocato. Nei test relativi alla *allreduce gerarchica mista*, invece, ogni nodo allocato eseguirà 4 processi, ciascuno dei quali associato a uno dei 4 *device* presenti nel nodo.

Questo tipo di allocazione è reso possibile dal fatto che rispetta l'architettura intra-nodo del modulo *Booster*. Ogni nodo è infatti composto da un processore *host* Intel Xeon Platinum 8385 CPU con 32 core e da 4 GPU Nvidia A100. La CPU dispone di 512 GB di memoria, distribuiti su 8 slot DDR4, mentre ciascuna GPU integra 64 GB di memoria HBM2e, organizzati in quattro stack da 16 GB ciascuno. Ogni GPU è connessa direttamente alle altre 3 tramite 4 collegamenti Nvidia NVLink 3.0 da 200 Gb/s.La comunicazione intra-nodo è completata da un bus PCIe Gen4.0 a 16 lane da 256 Gb/s per GPU, utilizzato per comunicare con la CPU host e con la NIC. I nodi sono interconnessi tramite una rete InfiniBand HDR e ogni nodo è dotato di due schede di rete NVIDIA Connect-X6 dual port da 200 Gb/s.

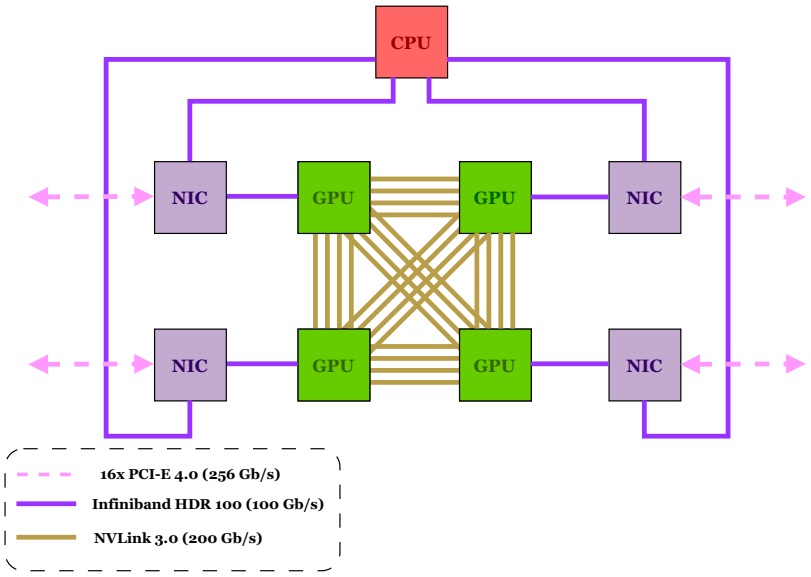


Figura 7.1. Architettura intra-nodo di *Leonardo*

7.2 Analisi delle prestazioni

Durante la fase di valutazione, al fine di ottenere una misura più realistica delle prestazioni della collettiva, sono state effettuate diverse misurazioni (circa 100 iterazioni) tramite `MPI_Wtime` per ciascun tipo di test e configurazione di allocazione. Il limite d'errore sulla compressione utilizzato per tutti i test è 1E-4. La prima

[3]Per *nodi allocati* intendiamo un sottoinsieme dei nodi del cluster su cui viene eseguito il programma.

modalità di analisi ha previsto lo studio dei tempi di esecuzione della collettiva con compressione omomorfica (`HomComp_Allreduce`), mantenendo fisso il numero di processi e variando la dimensione del dataset, e quindi dei messaggi scambiati da ciascun processo. La stessa procedura è stata applicata anche alla *allreduce gerarchica mista* (*ArGM*[4]). Per la `HomComp_Allreduce` sono stati utilizzati 32 nodi con una GPU per nodo, eseguendo la collettiva su dataset compresi tra 4 MB e 2 GB. Nel caso della ArGM, invece, sono stati allocati 16 nodi con 4 GPU per nodo. In primo luogo sono stati allocati 32 nodi per evitare l'eccessiva frammentazione del buffer in blocchi troppo piccoli, causata dalla *ring allreduce*, così da evidenziare il miglioramento introdotto dalla nuova collettiva quando la dimensione del messaggio è sufficientemente grande. Per la ArGM, invece, sono stati impiegati più device, in quanto ha mostrato una migliore scalabilità sotto questo aspetto. Come riportato in Figura 7.2, le prestazioni della nuova implementazione risultano inferiori a quelle della `MPI_Allreduce` per dataset di piccole dimensioni. L'efficacia della soluzione proposta emerge soltanto con dataset di dimensioni maggiori: tra 1 e 2 GB per la `HomComp_Allreduce` e tra 0.4 e 2 GB per la *ArGM*. Questo comportamento, osservato sui dataset di dimensioni ridotte, è riconducibile all'overhead introdotto dalla compressione, che non viene compensato dalla riduzione della latenza nella trasmissione e ricezione dei dati.

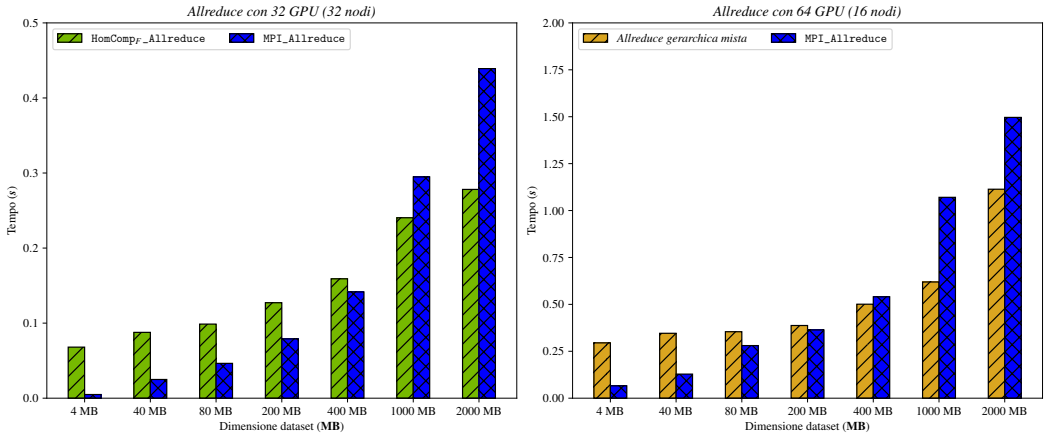


Figura 7.2. Esecuzione di `HomComp_Allreduce` e la *ArGM* con numero di GPU fissato e dimensione del dataset variabile. Il dataset utilizzato è composto da `float` generati con distribuzione normale.

Nella seconda tipologia di test (Figura 7.3)[5] è stato preso un dataset di dimensione ragionevole (1 GB), considerando il numero di *device* su cui verrà eseguito. In questo caso il confronto viene svolto fissando il dataset e variando il numero di GPU su cui viene eseguito sia per `HomComp_Allreduce` che per *ArGM*.

`HomComp_Allreduce` mantiene delle prestazioni in generale superiori a una allreduce GPU-centrica e alla `MPI_Allreduce`, tranne nel caso del test effettuato con 64 GPU. Il motivo di tale comportamento è identico a quello descritto per la ti-

[4]Introduciamo questa abbreviazione poiché verrà utilizzata estensivamente nel seguito.

[5]Si osserva una notevole variabilità nei tempi di esecuzione, che non risultano strettamente crescenti all'aumentare del numero di processi. Ciò dipende dalla modalità di misura adottata: per stimare i tempi su d GPU sono stati allocati d nodi (o $d/4$ nel caso della *ArGM*) ed eseguite 100 operazioni di allreduce. Ripetendo più volte questo processo, i tempi registrati possono variare sensibilmente a seconda dei nodi assegnati. Per questo motivo, i grafici più affidabili per un confronto sono quelli ottenuti mantenendo fissa l'allocazione ed eseguendo più test sulla stessa.

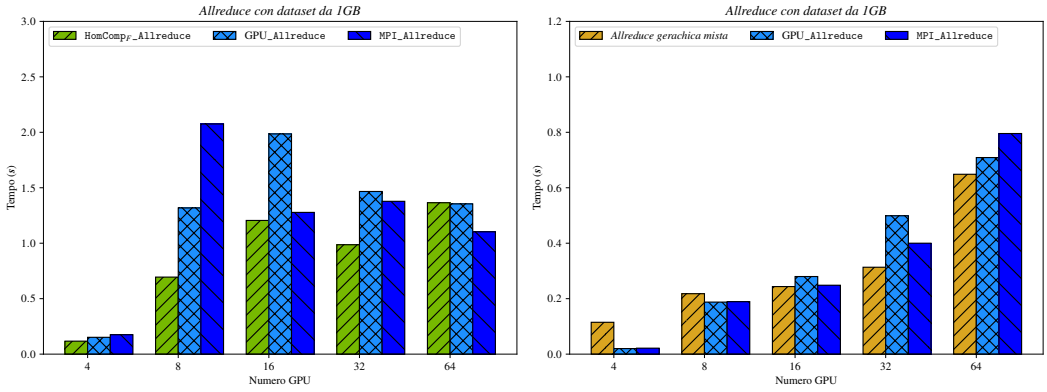
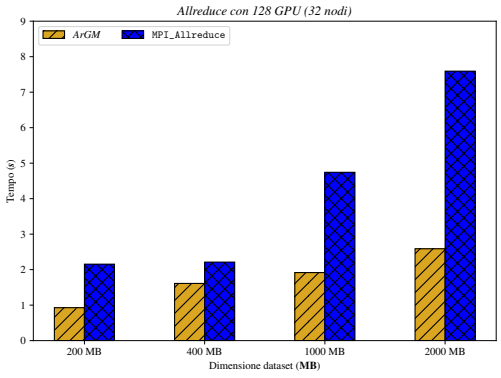


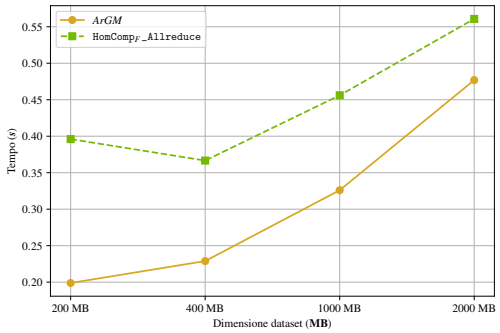
Figura 7.3. `HomComp_Allreduce` e *ArGM* con dataset fisso (`float` con distribuzione normale) e numero di GPU variabile.

pologia di test precedente. Quindi, per un numero di GPU abbastanza piccolo, il messaggio inviato e ricevuto è abbastanza grande da garantire che la latenza di invio (ricezione) del dato in forma originale sia maggiore dell'overhead di compressione (decompressione) insieme alla latenza di invio (ricezione) del dato compresso.

I dati raccolti mostrano che le nuove collettive proposte offrono un reale miglioramento delle prestazioni solo quando i messaggi hanno dimensioni elevate. Questo risultato è legato a due fattori principali: l'overhead introdotto dalla compressione omomorfica e la suddivisione del buffer originale in sottoblocchi, necessaria per l'algoritmo ad anello. A conferma di quanto discusso, un test condotto su 128 GPU (Figura 7.4 (a)) mostra che, mantenendo costante il numero di nodi allocati e variando la dimensione del dataset, il divario prestazionale tra MPI e *ArGM* diventa significativo, ad esempio con il dataset da 2 GB `HomComp_Allreduce` ottiene uno speedup di 3× rispetto alla *allreduce* di MPI. In Figura 7.4 (b) è invece riportato un confronto diretto tra le prestazioni delle due nuove collettive su un'allocazione di 32 nodi con 4 device per nodo. Come è evidente, la *ArGM* ottiene risultati superiori rispetto a $\texttt{HomComp}_F$, che non sfrutta le comunicazioni intra-nodo ad alta velocità.



(a) *ArGM* con allocate 128 GPU eseguito su dataset di `float` generati tramite *randomwalk*.



(b) Confronto tra `HomComp_Allreduce` e *ArGM*, con 128 GPU allocate (32 nodi, 4 GPU per nodo), su dataset *randomwalk* da 0.2 a 2 GB.

Figura 7.4

Profiling del kernel

Utilizzando lo strumento di profiling *Nsight Compute* (NCU), è stata condotta un'analisi del kernel di compressione omomorfica $\texttt{HomComp}_F$ sul dataset di dimensione 1 GB, con l'obiettivo di individuare eventuali colli di bottiglia e opportunità di ottimizzazione. Il report prodotto dal profiling è stato confrontato con quello relativo al kernel di compressione e decompressione di cuSZp2.

Dal confronto è emerso che il nuovo kernel ottiene valori di *occupancy* identici a quelli del kernel di compressione e decompressione; infatti, viene eseguito con le stesse dimensioni di blocco utilizzate in cuSZp2 (32 thread). Diversamente, però, dal punto di vista dell'utilizzo dei registri, la compressione e decompressione impiegano rispettivamente 56 e 48 registri per thread, mentre $\texttt{HomComp}_F$ ne utilizza 63, riuscendo comunque a raggiungere un *compute throughput* pari al 45.81%, superiore a quello della normale compressione (39.25%).

Il kernel rimane tuttavia *memory bounded*, come nel caso della compressione. Il maggiore utilizzo di registri è giustificato dall'impiego di variabili automatiche, necessarie per memorizzare rapidamente i risultati delle operazioni di quantizzazione e prediction. L'incremento del *compute throughput* deriva invece dall'integrazione della combinazione degli elementi direttamente all'interno del kernel di compressione.

Confronto con ghZCCL

L'algoritmo ad anello, discusso nei capitoli precedenti, non è l'unica strategia per eseguire una *allreduce*. Il framework **ghZCCL** [7] integra infatti la compressione omomorfica con una *recursive doubling allreduce* GPU-centrica. Questa scelta nasce dall'esigenza di evitare il sottoutilizzo della GPU, tipico degli algoritmi ad anello quando il numero di dispositivi coinvolti è elevato.

Senza entrare nei dettagli del compressore *ghZ* (che trasferisce su GPU i concetti già introdotti in hZCCL), confrontiamo i costi delle due varianti. Per l'*allreduce* con $\texttt{HomComp}_F$ si ha:

$$T^{AR}_{\texttt{HomComp}_F} = (p-1) \times \left\{ h_B + d_B + 2\left(\frac{n}{p}\beta\right) + 2\alpha \right\} + c_B, \tag{7.1}$$

mentre per l'*allreduce* di ghZCCL:

$$T^{AR}_{ghZCCL} = c + \log_2(p-1) \times \{h + n\beta + \alpha\} + h + d, \tag{7.2}$$

dove h è il costo della compressione omomorfica dell'intero buffer (dimensione n), c e d i costi di compressione e decompressione dell'intero dataset. Poiché le due strategie si basano su algoritmi collettivi differenti, un confronto diretto non è immediato. Tuttavia, possiamo evidenziare alcune osservazioni chiave:

Scalabilità – Il *recursive doubling* richiede un numero di step e compressioni che cresce in modo logratimico con il numero di processi, mentre nell'anello la crescita è lineare. Questo rende il *recursive doubling* più adatto a scenari con molti dispositivi, dove l'anello rischia di frammentare eccessivamente il buffer portando a un sottoutilizzo della GPU. Va notato che in (7.1) la compressione riguarda solo una porzione del blocco, mentre in (7.2) viene compresso/decompresso l'intero messaggio.

Dimensione dei messaggi – Per messaggi di grandi dimensioni, l'anello può risultare vantaggioso, perché frammenta il buffer in blocchi più piccoli, garantendo un utilizzo efficiente della GPU e comunicazioni più leggere rispetto al *recursive doubling*.

Fase di QP – In `HomComp` si applica la QP ai dati locali, che vengono poi combinati

con i dati compressi ricevuti a seguito di una decompressione parziale. In *ghZ*, invece, durante la compressione omomorfica entrambi i blocchi vengono decompressi parzialmente e combinati (come in hZCCL). In generale, le operazioni della fase di QP sono più semplici: richiedono meno letture da global memory (una sola per recuperare il float) e non necessitano di sincronizzazione globale.

Conclusioni

La compressione rappresenta uno dei principali strumenti a nostra disposizione per ridurre il carico sulle interconnessioni durante le operazioni collettive. Si tratta infatti di una tecnica generica, applicabile a qualsiasi tipologia di dato che contenga ridondanza di informazione, siano essi numeri o stringhe. La compressione omomorfica, in particolare, si rivela un approccio efficace per mitigare gli overhead associati al *DOC workflow* nelle operazioni collettive. Con l'introduzione di `HomComp` e delle due varianti della *ring allreduce* che integrano la compressione omomorfica, lo scenario di utilizzo della compressione nel calcolo distribuito è stato ampliato. Tuttavia, come evidenziato sia da questo studio sia da framework come ghZCCL, la *ring allreduce* mostra limitazioni di scalabilità con l'aumentare del numero di GPU. Un possibile sviluppo futuro potrebbe quindi consistere nell'estendere l'impiego di `HomComp` ad altri algoritmi di comunicazione collettiva. Ad esempio, si potrebbe valutare l'integrazione di `HomComp` nel *recursive doubling* o nell'algoritmo di *Rabenseifner*, modificando opportunamente l'implementazione già presente in ghZCCL.

Ad esempio durante la fase di vector halving nell'*allreduce* di *Rabenseifner*, si potrebbe adottare un algoritmo basato su un'euristica in grado di stimare un limite inferiore per la dimensione del blocco da inviare affinché l'overhead di compressione risulti conveniente rispetto alla latenza di trasmissione del blocco originale. In questo modo, nelle prime fasi del vector halving, quando i blocchi da inviare sono ancora di grandi dimensioni, essi vengono compressi. Nelle fasi finali, invece, quando la dimensione dei blocchi scende al di sotto del limite inferiore, i dati vengono decompressi un'ultima volta e lo scambio prosegue utilizzando blocchi non compressi.

Oltre all'estensione ad altri algoritmi, le prestazioni e la versatilità di `HomComp` potrebbero essere ulteriormente migliorate attraverso diversi interventi:

- Riduzione del numero di registri utilizzati durante la compressione omomorfica.
- Aumento dell'occupancy del kernel, incrementando la dimensione del *thread block* in modo che più di un warp possa essere eseguito in parallelo. Attualmente il compressore utilizza *thread block* di 32 thread.
- Maggiore flessibilità nella dimensione dei buffer: al momento sono supportati soltanto buffer di dimensione multipla di 2^{15}. Per messaggi di dimensioni arbitrarie è necessario introdurre padding in ciascun blocco, con un conseguente overhead.

Gli spunti per lavori futuri non si limitano all'estensione o al perfezionamento delle soluzioni già esistenti. Un possibile filone di ricerca riguarda l'applicazione della compressione omomorfica a nuove tipologie di dato, nonché lo studio e l'implementazione su GPU di schemi di compressione e decompressione parziale ad alte prestazioni, in grado di avvicinarsi il più possibile alla definizione formale di compressione omomorfica.

Bibliografia

[1] G. Barlas. *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann, 2015.

[2] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt. Integrating online compression to accelerate large-scale data analytics applications. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1205–1216, 2013.

[3] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox, 2014.

[4] D. De Sensi, L. Pichetti, F. Vella, T. De Matteis, Z. Ren, L. Fusco, M. Turisini, D. Cesarini, K. Lust, A. Trivedi, D. Roweth, F. Spiga, S. Di Girolamo, and T. Hoefler. Exploring gpu-to-gpu communication: Insights into supercomputer interconnects. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 1–15. IEEE, Nov. 2024.

[5] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

[6] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, Y. Huang, K. Raffenetti, H. Zhou, K. Zhao, X. Lu, Z. Chen, F. Cappello, Y. Guo, and R. Thakur. gzccl: Compression-accelerated collective communication framework for gpu clusters. *arXiv preprint arXiv:2308.05199*, 2023. Accessed: 2025-09-02.

[7] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, X. Lu, K. Zhao, Z. Chen, F. Cappello, Y. Guo, and R. Thakur. ghzccl: Gpu-aware homomorphic compression-accelerated collective communication library. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS 2025)*, pages 1–13. ACM, 2025.

[8] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, J. Zizhe, L. Xin, K. Zhao, X. Lu, Z. Chen, F. Cappello, Y. Guo, and R. Thakur. hzccl: Accelerating collective communication with co-designed homomorphic compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC24)*. IEEE Press, 2024.

[9] J. Huang, S. Di, X. Yu, Y. Zhai, Z. Zhang, J. Liu, X. Lu, K. Raffenetti, H. Zhou, K. Zhao, Z. Chen, F. Cappello, Y. Guo, and R. Thakur. C-coll: Accelerating collective communication with error-bounded lossy compression. *arXiv preprint arXiv:2304.03890*, 2023.

[10] Y. Huang, S. Di, G. Li, and F. Cappello. cuszp2: A gpu lossy compressor with extreme throughput and optimized compression ratio. *Proceedings of*

the International Conference for High Performance Computing, Networking, Storage and Analysis (SC24), 2024.

[11] Y. Huang, S. Di, X. Yu, G. Li, and F. Cappello. cuszp: An ultra-fast gpu error-bounded lossy compression framework with optimized end-to-end performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2023)*, pages 1–13, 2023.

[12] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, 2017.

[13] D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2016.

[14] Lawrence Livermore National Laboratory. Zfp: Compressed floating-point and integer arrays. https://zfp.llnl.gov/, 2023.

[15] D. Merrill and M. Garland. Single-pass parallel prefix scan with decoupled look-back. Technical Report NVR-2016-002, NVIDIA Corporation, 2016.

[16] MPI Forum. Mpi: A message-passing interface standard. https://www.mpi-forum.org/docs/.

[17] NVIDIA Corporation. Cuda toolkit documentation. https://docs.nvidia.com/cuda/.

[18] NVIDIA Corporation. Nvidia nvlink high-speed gpu interconnect. https://www.nvidia.com/en-us/data-center/nvlink/, 2016.

[19] NVIDIA Corporation. nvcomp: Gpu-accelerated lossless and lossy compression library. https://developer.nvidia.com/nvcomp, 2023.

[20] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[21] R. Rabenseifner. Optimization of collective reduction operations. In *International Conference on Computational Science (ICCS 2004)*, Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2004.

[22] M. Ruefenacht, M. Bull, and S. Booth. Generalisation of recursive doubling for allreduce: Now with simulation. *Parallel Computing*, 69:24–44, 2017.

[23] D. Salomon. *A Concise Introduction to Data Compression*, pages 51–58, 235–238. Springer, London, UK, 4th edition, 2007.

[24] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2017.

[25] S. W. Son, Z. Chen, W. Hendrix, A. Agrawal, W.-k. Liao, and A. Choudhary. Data compression for the exascale computing era - survey. *Supercomputing Frontiers and Innovations*, 1(2):76–88, Sep. 2014.

[26] SZ Development Team. Sz: Error-bounded lossy compression for scientific data. https://szcompressor.org.

[27] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[28] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, and F. Cappello. cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 3–15. ACM, Sept. 2020.

[29] M. Turisini, G. Amati, and M. Cestari. Leonardo: A pan-european pre-exascale supercomputer for hpc and ai applications, 2023.

[30] D. Unat, I. Turimbetov, M. K. T. Issa, D. Sağbili, F. Vella, D. D. Sensi, and I. Ismayilov. The landscape of gpu-centric communication, 2024.

[31] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[32] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.