UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

"Implementazione di algoritmi approssimati per k-center clustering su architetture Processing-In-Memory"

Relatore:
Prof. Francesco Silvestri

Laureando:
Francesco Visentin

ANNO ACCADEMICO 2021-2022

Data di laurea 19/07/2022

Ai miei genitori, ad Anna

e a tutti i cari che mi hanno accompagnato in questo percorso.

Indice

1 Introduzione

La grande maggioranza delle moderne architetture hardware sono caratterizzate da una forte dicotomia tra processore e memorie. Questa netta separazione rende il sistema altamente dipendente dallo spostamento dei dati, attività che diventa necessaria per permettere l'accesso, l'elaborazione ed il salvataggio di questi.

Questo limite architetturale prende il nome di 'Memory Wall' e rappresenta uno dei principali coni di bottiglia all'interno dei moderni sistemi hardware.

Il crescente gap prestazionale che divide CPU e memorie, unito alla forte dipendenza dall'accesso a grandi quantità di dati che caratterizza molte tra le moderne aree di interesse del settore informatico, quali machine learning, graph processing, bioinformatica... hanno fatto si che il peso di questo cono di bottiglia stia risultando, negli anni, sempre più marcato ed influente rispetto alle prestazioni generali del sistema.
Recenti studi sperimentali evidenziano infatti come lo spostamento dei dati tra CPU e memorie rappresenti il 62% [1] (riportato nel 2018), 40% [2] (riportato nel 2013) ed il 35% [3] (riportato nel 2014) del totale consumo energetico rispettivamente per applicativi a fini commerciali, scientifici e per dispositivi mobili.

Le architetture processing-in-memory (PIM) rappresentano un promettente, e sempre più popolare, paradigma architetturale che mira a risolvere il problema del 'muro della memoria' inserendo unità computazionali aggiuntive all'interno dei chip DRAM.
Spostando una parte della computazione direttamente nelle memorie il modello PIM permette di diminuire la dipendenza dell'intero sistema dal processore, riducendo così drasticamente il numero ed il peso degli spostamenti necessari per accedere ai dati.

Nelle pagine a seguire verrà riportata in primo luogo una panoramica generale sulle architetture PIM, focalizzandosi sui principali vantaggi/svantaggi legati ai diversi modelli architetturali attualmente definiti.
Viene a seguire fornita, come benchmark prestazionale, l'implementazione di un algoritmo approssimato per la risoluzione del problema computazionale k-center clustering, scritta per essere eseguita sull'architettura PIM realizzata da UPMEM, azienda produttrice dei primi chip DRAM contenenti un architettura PIM ad essere stati messi in commercio.

2 Processing-in-Memory (PIM)

2.1 Criticità della dipendenza dal processore.

La netta separazione tra processore e memoria centrale è una conseguenza diretta del modello di von Neumann [4] che definisce, come uno dei suoi principi fondanti, la divisione tra unità computazionali ed unità di memoria in due are distinte, collegate da bus dati esterni.

La scelta di delegare totalmente l'elaborazione dei dati al processore rende necessario un continuo spostamento di dati tra memorie e CPU. Questo fa si che il sistema risulti altamente sbilanciato. Il processore copre infatti un ruolo predominante all'interno dell'architettura e, per questo motivo, è il componente a cui negli anni è stata dedicata la maggiore attenzione tecnologica.

In contrasto le unità di memoria, come la memoria centrale, e le unità di trasporto hanno assunto un ruolo secondario e risultano, in confronto, molto meno ottimizzate.

La crescente disparità prestazionale tra unità computazionali ed unità di memorizzazione/trasporto ha portato oggi a sistemi in cui il costo legato allo spostamento dei dati tra i vari componenti domina i costi computazionali in termini di energia richiesta.

In particolare, ad oggi, l'energia richiesta per un accesso in memoria centrale risulta superiore di circa due/tre ordini di grandezza rispetto all'energia richiesta da una complessa operazione di addizione. Ad esempio, [3] riporta che l'energia di un accesso in memoria risulta $\sim 115\times$ l'energia consumata da un'operazione di addizione.

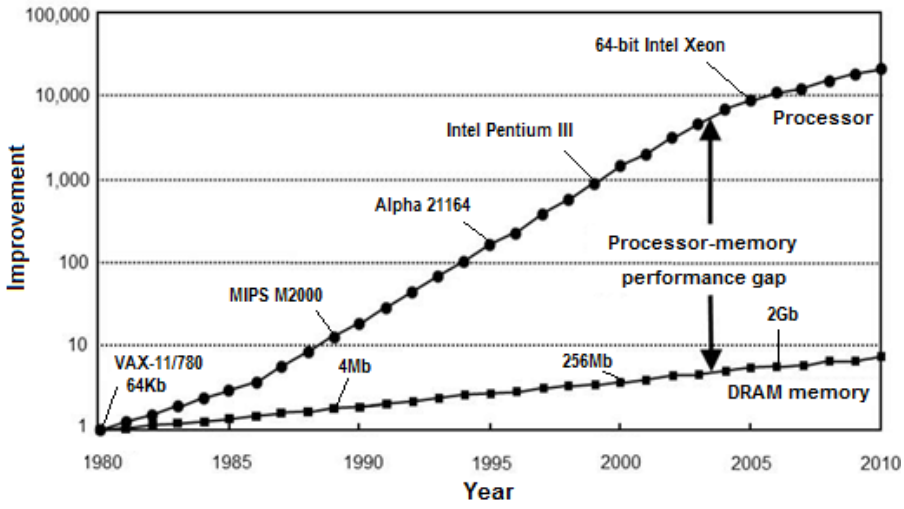

Figura 1:

Miglioramento annuale delle velocità dei processori e delle memorie DRAM secondo [5]

Da queste limitazioni energetiche, unite alle crescenti difficoltà tecniche che limitano l'ottimizzazione delle memorie DRAM [6] [7], nasce la necessità per i moderni sistemi hardware di adottare un cambio di paradigma architetturale. Citando [8] :
"Riteniamo che le future architetture di calcolo dovrebbero diventare più incentrate sui dati: dovrebbero (1) essere in grado di eseguire calcoli con il minimo spostamento dell'informazione e (2) processare l'informazione dove ha più senso (ovvero dove i dati fisicamente risiedono)".

Il modello Processing-in-Memory nasce per rispondere a questa esigenza.

2.2 Panoramica sulle architetture PIM

Mentre le applicazioni continuano a processare quantità sempre maggiori di dati, i costi legati agli spostamenti dell'informazione all'interno del sistema diventano sempre più significativi. Le architetture Processing-in-Memory permettono di effettuare una parte, o tutta l'elaborazione dei dati, direttamente in memoria riducendo così il numero di spostamenti necessari e limitando l'impatto del bottleneck di von Neumann sulle prestazioni del sistema.

Nei paragrafi a seguire viene proposta una breve panoramica sulla storia, sulle tecnologie promotrici e sugli attuali approcci implementativi che stanno alla base della tecnologia PIM.

2.2.1 Recente spinta tecnologica

Il paradigma Processing-in-Memory viene esplorato ormai da più di 50 anni.
L'architettura 'Logic-in-Memory computer' [9] proposta da Harold Stone nel 1970 ne è uno dei primi esempi. A questa si sono susseguiti negli anni numerosi altri modelli ed approcci architetturali focalizzati ad integrare computazione e memorie.
Molti tra questi lavori furono però ostacolati dalle limitazioni tecnologiche del tempo che ne impedirono la concreta implementazione.

Recentemente, la crescente necessità da parte di molti applicativi moderni di accedere a grandi quantità di dati, unita alle difficoltà tecniche che rendono complesso aumentare ulteriormente la densità, diminuendo contemporaneamente consumo e latenza delle moderne architetture DRAM, hanno portato le aziende produttrici a sviluppare due nuovi design per la realizzazione di unità di memoria centrale, entrambi utilizzabili per superare le precedenti barriere che ostacolavano l'implementazione di architetture PIM.

La prima principale innovazione è l'introduzione delle '3D-stacked memory' [10, 11], particolari architetture dove molti 'strati di memoria' (tipicamente DRAM) vengono sovrapposti uno all'altro (Figura 2). Questi strati sono poi collegati tra loro tramite dei TVS ("Through-Silicon Vias") verticali che offrono velocità di comunicazione nettamente migliori rispetto ai normali canali dati usati nelle memorie DRAM.
In aggiunta ai multipli livelli di DRAM, numerose architetture '3D-stacked memory' includono, tipicamente nello strato più basso, un livello logico anch'esso direttamente collegato alle memorie tramite TSV.

Questo strato può essere sfruttato, rispettando degli opportuni requisiti spaziali, energetici e termici, per implementare della logica computazionale 'general purpose' direttamente all'interno del chip DRAM.

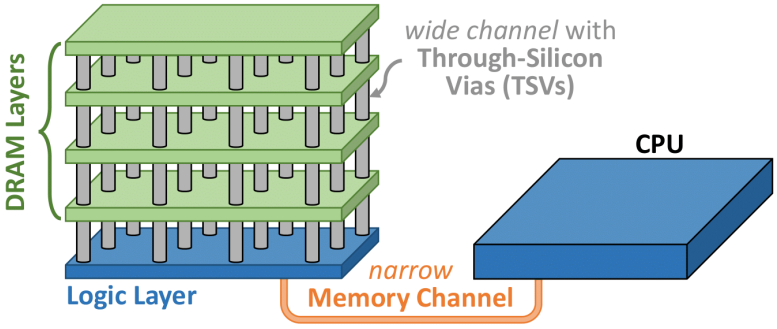

Figura 2: (Riprodotta da [12])
Schema ad alto livello dell'architettura '3D-stacked memory'.

La seconda importante innovazione tecnologica riguarda l'utilizzo, come memorie centrali, di nuove memorie non volatili (NVM) capaci di immagazzinare dati con densità molto superiori rispetto alle tradizionali memorie DRAM, mantenendo latenza e consumi comparabili. Tra queste le tecnologie più promettenti sono:

- 'Phase-Change memory (PCM)' [13, 14].
- 'Magnetic RAM (MRAM)' [15].
- 'Metal-oxide resistive RAM (RRAM)' o memresistori [16].

Le tecnologie NVM, ad oggi, non soffrono dei problemi di scalabilità che limitano le memorie DRAM e permettono quindi maggiore flessibilità ai loro progettisti.
In particolare offrono la possibilità di integrare delle funzionalità PIM direttamente al loro interno. Recenti studi [17, 18] dimostrano infatti come le celle delle memorie NVM possono essere utilizzate, similmente a quanto avviene per le memorie DRAM, per effettuare operazioni logiche Booleane direttamente al loro interno.

2.2.2 Approcci al modello PIM

Partendo dalle nuove tecnologie precedentemente descritte possiamo distinguere due principali metodi per realizzazione un'architettura PIM.

Entrambi i due approcci descritti a seguire rappresentano dei modelli generali e presentano, ad oggi, delle limitazioni e/o criticità legate alle nuove tecnologie utilizzate. Per questo motivo complete architetture PIM, basate su 3D-stacked memory o sul modello processing-using-memory, non sono ancora state commercializzate.
È tuttavia, attualmente disponibile in commercio, un'architettura 'ibrida' (sfrutta il modello processing-near-memory ma utilizzando tradizionali 2D DRAM) prodotta da UPMEM che verrà descritta nel dettaglio nella prossima sezione.

1. Processing using memory

Questo primo approccio permette, tramite minime modifiche alle strutture delle moderne architetture di memoria, di effettuare operazioni computazionali bit per bit direttamente sulle righe delle memorie SRAM, DRAM e non volatili (NVM) sfruttando le proprietà operazionali delle celle di memoria contenute all'interno di queste.

Questo modello 'minimalistico' risulta particolarmente efficace per ottimizzare quelle particolari istruzioni che richiedono l'accesso simultaneo (ed idealmente ripetuto) a numerose celle di dati come ad esempio operazioni di copia, di inizializzazione o operazioni logiche bit per bit.

Le architetture processing-using-memory tendono però ad essere meno efficienti se utilizzate per operazioni più complesse rispetto alle operazioni logiche elementari (AND, OR e NOT). Queste operazioni più complesse, per essere supportate, richiedono infatti un maggiore overhead di area occupata o modifiche più significative ai componenti che organizzano, manipolano e spostano i dati all'interno dei chip di memoria.

2. Processing near memory

Questo secondo approccio sfrutta la tecnologia delle architetture '3D-stacked memory' per implementare delle unità computazionali 'general purpose' direttamente all'interno del livello logico delle memorie 3D.
Questa logica PIM integrata rende questo approccio più generale e flessibile rispetto al precedente. A seconda del design scelto per l'architettura è possibile infatti eseguire all'interno delle memorie sia parti di un'applicazione (da singole istruzioni ad intere funzioni) che interi thread e programmi.

Questo modello, sfruttando l'elevata velocità delle connessioni TVS che collegano direttamente le unità logiche agli strati di memoria, riesce a garantire una latenza molto ridotta ed una banda elevata.

La maggiore complessità tecnologica di questo approccio comporta però anche degli svantaggi: l'elevato costo e la capacità ridotta delle memorie 3D, unite alle limitazioni hardware che il livello logico deve rispettare (la ridotta area a disposizione ed i vincoli sul consumo energetico) vanno infatti a limitare le prestazioni della logica PIM integrata.

Approach	Enabling Technologies
Processing-Near-Memory	Logic layers in 3D-stacked memory Silicon interposers Logic in memory controllers
Processing-Using-Memory	SRAM DRAM Phase-change memory (PCM) Magnetic RAM (MRAM) Resistive RAM (RRAM)/memristors

Figura 3: (Riprodotta da [19])
Riassunto delle tecnologie utilizzate dai due principali approcci PIM

2.3 Architettura UPMEM

L'architettura PIM proposta da UPMEM permette di evitare le limitazioni legate agli approcci precedentemente descritti adottando un modello 'ibrido' in cui dei convenzionali chip 2D DRAM vengono affiancati da delle unità computazionali 'general purpose' chiamate 'DRAM Processing Units (DPUs)' inserite direttamente all'interno del chip.

La scelta di combinare unità di memoria standard ed unità computazionali all'interno dello stesso chip comporta numerosi vantaggi rispetto ad altri modelli PIM.
(1) L'utilizzo di memorie 2D DRAM, tecnologia che gode di processi di fabbricazioni e design maturi, permette di evitare gli svantaggi legati alle emergenti architetture '3D-stacked memory'. (2) Le DPU, a differenza dei modelli processing-using-memory, non sono limitiate a sole operazioni elementari ma possono svolgere una vasta gamma di operazioni similmente a quanto effettuato dai moderni processori 'general-purpose'. (3) La possibilità, per i thread contenuti nelle DPU, di eseguire indipendentemente rende l'architettura adatta ad eseguire programmi in parallelo. (4) Infine, UPMEM, mette a disposizione un intero pacchetto software che permette di programmare il codice eseguito dalle DPU direttamente in linguaggio C.

Figura 4: Modulo PIM prodotto da UPMEM

2.3.1 Organizzazione del sistema

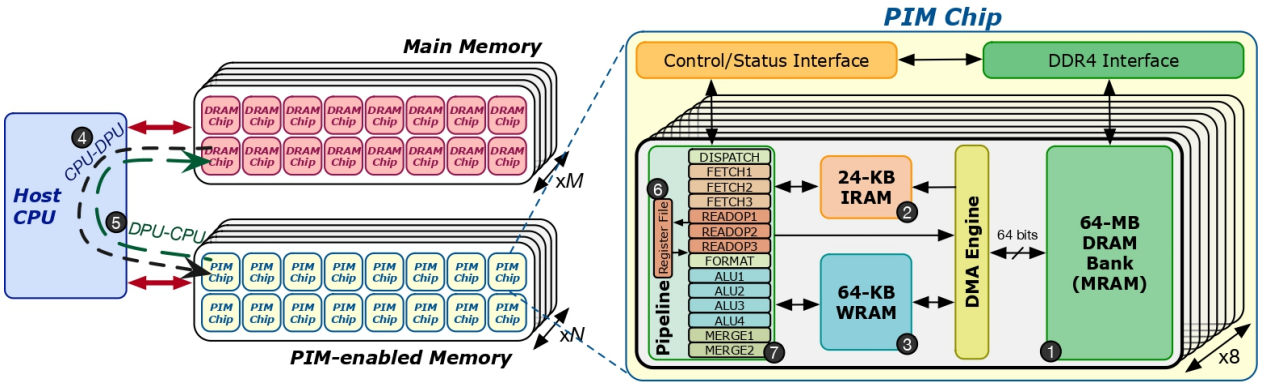

Figura 5: (Riprodotta da [20])
Architettura PIM basata sulla tecnologia UPMEM formata da: una CPU centrale, una memoria DRAM standard e delle memorie 'PIM-enabled'

La figura sovrastante rappresenta, nella parte sinistra, un'architettura PIM basata sulle memorie realizzate da UPMEM.
Il modello sovrastante contiene: (1) un processore centrale che coordina l'intero sistema, (2) dei banchi di memoria DRAM standard, (3) delle memorie 'PIM-enabled' dotate di unità computazionali integrate.

Ogni memoria UPMEM (Figura 4) è formata da un modulo standard DDR4-2400 contenente numerosi chip PIM. Come illustrato nella parte destra della figura 5, all'interno di ogni chip PIM sono presenti 8 DPU ognuna avente accesso ad esclusive aree di memoria. Queste vengono utilizzate, oltre che per la computazione, anche per comunicare con la CPU centrale.

Il passaggio di dati diretto tra diverse DPU non è supportato. Ogni comunicazione necessita prima di transitare attraverso la CPU centrale.

La CPU centrale è responsabile di coordinare il lavoro svolto dalle DPU.
In fase di esecuzione la CPU può decidere di allocare il numero desiderato di DPU (formando un 'DPU set') per caricare al loro interno del codice eseguibile (detto 'DPU kernel'). Questo codice può poi essere eseguito sia in modalità sincrona, sospendendo temporaneamente il thread chiamante della CPU centrale, che in modalità asincrona.

Ad oggi, all'interno di un sistema PIM basato su questa tecnologia, è possibile inserire al massimo 20 moduli di memoria UPMEM. Un'architettura così strutturata può arrivare a contenere 2560 DPU per un totale di 160-GB di memoria 'PIM-enabled'.

2.3.2 Architettura delle DPU

Una DPU è un semplice processore RISC a 32 bit contenente 24 thread, chiamati 'tasklet', ognuno dei quali dotato di 24 registri 'general purpose' da 32 bit.

La pipeline di una DPU ha una profondità di 14 livelli, solo gli ultimi tre di questi hanno però la possibilità di eseguire in parallelo. Per riempire totalmente la pipeline e sfruttare così al meglio le capacità di ogni thread, le istruzioni devono quindi essere inviate con 11 cicli di distanza [20].

Ogni DPU ha accesso esclusivo a tre diverse aree di memoria:

- MRAM: blocco di memoria DRAM da 64-MB accessibile sia dalla DPU che dalla CPU centrale. Permette il passaggio dati tra CPU-DPU ed DPU-CPU sia in modalità sequenziale che in parallelo permettendo di comunicare contemporaneamente con più DPU.

  La dipendenza diretta tra DPU e blocchi di memoria MRAM fa si che le capacità computazionali del sistema scalino linearmente con la quantità di memoria presente.

- IRAM: blocco di memoria da 24-KB usato per contenere gli opcode delle istruzioni da eseguire.

- WRAM: blocco di memoria da 64-KB usato dalla DPU come memoria centrale. Qui vengono posizionati variabili globali ed activation record. Perché una DPU possa elaborare un dato questo deve prima essere copiato dalla MRAM alla WRAM tramite istruzioni DMA bloccanti.

  Una DPU può raggiungere frequenze superiori a 400-MHz. A questa frequenza la larghezza di banda massima tra MRAM e WRAM può raggiungere ∼800-MB/s.

3 Implementazione dell'algoritmo

3.1 Introduzione

In questa sezione viene fornita l'implementazione di un algoritmo approssimato per la risoluzione del problema computazionale k-center clustering scritta per essere eseguita sull'architettura PIM realizzata da UPMEM.

La sezione 3.2 offre una panoramica generale sul problema computazionale k-center clustering.

I paragrafi successivi vanno invece a presentare nel dettaglio il codice prodotto.
La sezione 3.3 introduce l'algoritmo ad alto livello tramite pseudo codice, le restanti sezioni 3.4 ed 3.5 includono infine dei frammenti di codice C, focalizzati principalmente ad evidenziare le interazioni con l'architettura UPMEM.

Nello specifico queste ultime due sezioni contengono il codice relativo ad:

- **Host application:** codice eseguito all'interno della CPU centrale che, tramite l'Host library [21] fornita da UPMEM, coordina l'intera esecuzione del programma. L'host application si occupa di allocare le DPU, avviarne l'esecuzione e gestire il trasferimento dati tra queste e la memoria centrale.
- **DPU kernel:** codice eseguito direttamente all'interno delle DPU. Sfrutta le primitive di sincronizzazione messe a disposizione dall'UPMEM Runtime library [22] per gestire l'esecuzione in parallelo dei vari tasklet e la comunicazione con l'host application.

3.2 K-center clustering

'K-center clustering' è un particolare problema computazionale appartenente alla famiglia dei clustering 'center-based'.

In particolare questo problema richiede di trovare, dato in input un insieme di punti P, l'insieme di k centri che permettono di minimizzare la massima distanza di ogni punto dal centro del suo cluster.

Più formalmente possiamo definire il problema come:

> Sia $k > 0$ e sia (M, d) uno spazio metrico, il problema computazionale k-center clustering è un particolare problema di ottimizzazione che, dato un insieme finito di punti $P \subseteq M$, richiede di determinare un sottoinsieme di k punti $S \subseteq P$ tale da minimizzare la seguente funzione obbiettivo:
>
> $$\Phi_{kcenter}(P, S) \ = \ max_{x \in P} \ d(x, S) \tag{1}$$
>
> Dove $d(x, S)$ indica il valore della distanza minima tra x ed i punti di S:
>
> $$d(x, S)) \ = \ min\{d(x, y) \ : \ y \in S\} \tag{2}$$

Nelle definizioni precedentemente riportate appare come centrale il concetto di 'distanza tra due punti'. Dato un insieme di punti in uno spazio affine esistono varie formule che permettono di esprimere questo valore in modi diversi.
In questa particolare trattazione la distanza tra due punti viene calcolata utilizzando la distanza di Minkowsky secondo cui:

> Siano $X, Y \in \mathbb{R}^n$ con $X = (x_1, x_2, ..., x_n)$ ed $Y = (y_1, y_2, ..., y_n)$, la distanza d_L tra X ed Y è:
>
> $$d_L(X, Y) \ = \ \left(\sum_{i=1}^{n} |x_i - y_i|^n \right)^{1/n} \tag{3}$$

3.2.1 Approcci risolutivi

K-center clustering, come numerosi altri problemi di ottimizzazione, appartiene alla famiglia di problemi NP-Hard. Per questo motivo l'utilizzo di soluzioni esatte risulta eccessivamente dispendioso a livello computazionale (sopratutto per data set di grandi dimensioni) ed è quindi necessario utilizzare algoritmi approssimati.

Nello specifico, all'interno del codice riportato nelle sezioni a seguire, sono stati utilizzati i due seguenti algoritmi approssimati:

- Farthest-First Traversal [23]
- MapReduce-Farthest-First Traversal

1. Farthest-First Traversal:

Dato P un insieme di N punti nello spazio metrico (M, d) ed dato $k > 0$ il numero di centri desiderato l'algoritmo procede come segue:

```
S ← {c_1}          //dove c_1 punto arbitrario ∈ P
for i ← 2 to k do
    Trova il punto c_i ∈ {P \ S} che massimizza d(c_i, S)
    S ← {c_1}
return S
```

E possibile dimostrare come 'Farthest-First Traversal' sia un algoritmo di approssimazione con fattore pari a due. Vale quindi che:

> Sia S l'insieme dei centri restituiti dall'applicazione di 'Farthest-First Traversal' sull'insieme di punti P, vale allora che:

$$\Phi_{kcenter}(P, S) \;\leq\; 2 \cdot \Phi^{opt}_{kcenter}(P, k) \tag{4}$$

2. MapReduce-Farthest-First Traversal:

Dato P, un insieme di N punti nello spazio metrico (M, d), e dato $k > 0$ il numero di centri desiderato, l'algoritmo procede come segue:

Fase 1:
- Map phase: divido P in m sottoinsiemi arbitrari di uguale grandezza $P_1, P_2, ..., P_m$
- Reduce phase: per ogni $i \in [1, m]$ eseguo 'Farthest−First Traversal' su P_i per determinare un set di k centri $T_i \subseteq P_i$

Fase 2:
- Map phase: vuota.
- Reduce phase: esegui 'Farthest−First Traversal' sull'insieme $T = \cup_{i=1}^{m} T_i$ formato dall'unione dei sottoinsiemi precedenti per ricavare l'insieme $S = \{c_1, c_2, ..., c_k\}$ contenente i centri finali

E possibile dimostrare come questo sia un algoritmo di approssimazione con fattore pari a quattro. Vale quindi che:

Sia S l'insieme dei centri restituiti dall'applicazione di 'MapReduce-Farthest-First Traversal' sull'insieme di punti P, vale allora che:

$$\Phi_{kcenter}(P, S) \leq 4 \cdot \Phi^{opt}_{kcenter}(P, k) \tag{5}$$

Tra i due algoritmi Farthest-First Traversal gode di un ottimo fattore di approssimazione ma richiede, per poter essere eseguito, che l'intero insieme di punti P sia caricato in memoria. Per via di questa questa limitazione, nel codice sottostante, FFT verrà eseguito unicamente all'interno della CPU centrale e verrà usato come benchmark di riferimento per calcolare il costo del clustering.

All'interno delle DPU verrà invece eseguito MapReduce-Farthest-First Traversal. Questo algoritmo permette di sfruttare al massimo il forte parallelismo del sistema assegnando ad ogni DPU un sottoinsieme di punti di piccole dimensioni su cui calcolare il proprio set di centri T_i. I centri ottenuti vengono poi caricati in memoria centrale dove la CPU applicherà nuovamente FFT per ottenere i valori finali.

3.3 Pseudo codice

L'algoritmo presentato sfrutta due round PIM, ognuno composto da quattro fasi:

1. Passaggio input: i dati vengono trasferiti dalla memoria centrale alle DPU.
2. Elaborazione dei dati all'interno delle DPU.
3. Passaggio output: i dati vengono trasferiti dalle DPU alla memoria centrale.
4. Elaborazione finale, effettuata dalla CPU, dei dati restituiti.

Il primo round viene utilizzato per calcolare i centri del clustering eseguendo l'algoritmo 'MapReduce-Farthest-First Traversal'. Il secondo round sfrutta invece i dati già presenti all'interno delle DPU (vedi 3.4.4) per calcolare il costo totale del clustering effettuato[1].

A seguire vengono riportate, tramite dello pseudo codice molto generale, le descrizioni degli algoritmi utilizzati dall'host application e dai due DPU kernel.

Una descrizione più dettaglia dei vari codici viene fornita nella sezioni 3.4 ed 3.5.

Host application

```
Round 1:
    Fase 1)
      - Divido l'insieme di punti in tanti buffer quante sono le DPU allocate.
      - Carico i dati dalla memoria centrale alle DPU.
    Fase 2)
      - Eseguo DPU Kernel 1 in modalita' sincrona.
    Fase 3)
      - Recupero in un unico buffer i centri intermedi calcolati dalle DPU.
    Fase 4)
      - Calcolo i centri finali applicando FFT sull'insieme precedentemente recuperato.

Round 2:
    Fase 1)
      - Carico nelle DPU i centri finali calcolati nel round 1.
    Fase 2)
      - Eseguo DPU Kernel 2 in modalita' sincrona.
    Fase 3)
      - Recupero in un unico buffer i costi calcolati dalle DPU.
    Fase 4)
      - Calcolo il costo del clustering trovando il massimo tra i costi restituiti
        dalle DPU.
```

[1] Il costo del clustering, sia S l'insieme dei centri finali e sia P l'insieme dei punti, è definito come:

$$costo = max\{d(x,s) \ : \ x \in P, \ s \in S\} \tag{6}$$

DPU Kernel 1: Calcolo dei centri

```
Inizializzazione:
  - Resetto WRAM e variabili condivise.
  - Alloco buffer per contenere i centri calcolati.
  - Scelgo casualmente il primo centro.

Calcolo dei centri:
  while: numero centri calcolati != k, do:
    - Divido in blocchi l'insieme dei punti.
    - Assegno ad ogni tasklet un insieme N di blocchi.

    while: blocco corrente != ultimo blocco del tasklet, do:
      - Trovo punto avente distanza massima per blocco corrente.
      if: primo blocco || distanza punto > distanza centro candidato precedente, do:
        - Il punto corrente diventa il nuovo centro candidato

      - Assegno un nuovo blocco al tasklet

    - Trovo centro avente distanza massima tra i centri candidati calcolati dai tasklet.
    - Aggiungo punto trovato al buffer dei centri.

  - Carico centri finali della DPU in memoria MRAM.
```

DPU Kernel 2: Calcolo dei costi

```
Inizializzazione:
  - Resetto WRAM e variabili condivise.
  - Carico su un buffer i centri finali precedentemente calcolati.

Calcolo dei costi:
  - Divido in blocchi l'insieme dei punti.
  - Assegno ad ogni tasklet un insieme N di blocchi.

  while: blocco corrente != ultimo blocco del tasklet, do:
    - Trovo punto avente costo massimo per blocco corrente.
    if: primo blocco || costo punto > costo massimo precedente, do:
      - Il costo corrente diventa il nuovo costo massimo del tasklet

    - Assegno un nuovo blocco al tasklet

  - Trovo costo del clustering come massimo tra i costi calcolati dai tasklet.
  - Carico costo finale della DPU in memoria MRAM.
```

3.4 Host Application

L'host application contiene il codice utilizzato per coordinare l'esecuzione dell'intero programma. Nei paragrafi a seguire sono descritte, tramite dei frammenti di codice, le principali operazioni svolte.

Molte tra le funzioni dell'Host library necessitano di essere chiamate all'interno dello statement `DPU_ASSERT(...)`. Questo garantisce, in caso di errore, la corretta terminazione del programma ed il rilascio delle risorse DPU.

3.4.1 Allocazione DPU

```
//Alloco le DPU.
struct dpu_set_t dpu_set;
DPU_ASSERT(dpu_alloc(NR_DPUS, NULL, &dpu_set));
```

L'allocazione del numero desiderato di DPU avviene tramite il comando `dpu_alloc(...)`. La funzione inserisce un riferimento a queste, all'interno del puntatore alla struttura `struct dpu_set_t` passato come parametro.

A differenza del numero di DPU allocate il numero di tasklet associati ad ogni DPU va definito in fase di compilazione utilizzando i seguenti tre parametri:

- `NR_TASKLET`: definisce il numero di tasklet per DPU.
- `STACK_SIZE_DEFAULT`: definisce la dimensione di default dello stack per i tasklet.
- `STACK_SIZE_TASKLET_<X>`: definisce la dimensione dello stack per un preciso tasklet. (I taskelt sono identificati da un ID che va da [0, `NR_TASKLET`-1])

Le DPU raggiungono prestazioni ottimali quando il numero di thread in esecuzione supera la profondità della pipeline. Gli algoritmi paralleli strutturati per sfruttare il massimo numero possibile di DPU, ognuna dotata di 16 tasklet attivi, permettono quindi di raggiungere le massime prestazione del sistema [24].

3.4.2 Caricamento dati da CPU a DPU

```
//Suddivisione punti per ogni DPU.
//La dimensione del blocco assegnato ad ogni DPU deve essere allineata su 8 byte.
uint32_t points_per_dpu = p.n_points/NR_DPUS;
uint32_t points_per_last_dpu = p.n_points-points_per_dpu*(NR_DPUS-1);
uint32_t mem_block_per_dpu = points_per_last_dpu*p.dim;
uint32_t mem_block_per_dpu_8bytes = ((mem_block_per_dpu*sizeof(T) % 8) == 0) ?
                                    mem_block_per_dpu : roundup(mem_block_per_dpu, 8);
```

```
//Carico i dati
DPU_FOREACH(dpu_set, dpu, i) {
    DPU_ASSERT(dpu_prepare_xfer(dpu, P + i*points_per_dpu*p.dim));
}
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, 0,
                         mem_block_per_dpu_8bytes*sizeof(T), DPU_XFER_DEFAULT));
```

3.4.3 Recupero dati da DPU a CPU

```
//Recupero i dati
DPU_FOREACH(dpu_set, dpu, i) {
    DPU_ASSERT(dpu_prepare_xfer(dpu, R + i*data_length_8bytes));
}
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_FROM_DPU, DPU_MRAM_HEAP_POINTER_NAME,
                         dpu_center_set_addr, data_length_8bytes*sizeof(T),
                         DPU_XFER_DEFAULT));
```

L'Host library mette a disposizione diverse funzioni per consentire il trasferimento dei dati tra host e DPU.

Nel codice riportato la funzione `dpu_prepare_xfer(...)` assegna ad ogni DPU un buffer da utilizzare per il trasferimento/ricezione dei dati mentre la funzione `dpu_push_xfer(...)` permette di riempire/caricare il buffer dati precedentemente predisposto tramite un unico trasferimento da/verso un set di DPU.

I buffer di memoria utilizzati devono rispettare alcune limitazioni sull'allineamento dei byte in base alla memoria a cui tentano di fare accesso:

- **IRAM**: gli indirizzi devono essere allineati su 8 byte.
- **WRAM**: gli indirizzi devono essere allineati su 4 byte.
- **MRAM**: gli indirizzi devono essere allineati su 8 byte.

La variabile `DPU_MRAM_HEAP_POINTER_NAME` rappresenta un puntatore ad un indirizzo MRAM che delimita l'inizio dell'area di memoria attualmente disponibile. Questa può essere usata liberamente dal programma.

3.4.4 Avvio delle DPU

```
//Carico il programma per calcolare i centri ed avvio le DPU. (Round 1)
DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY1, NULL));
DPU_ASSERT(dpu_launch(dpu_set, DPU_SYNCHRONOUS));

...

//Carico il programma per calcolare il costo del clustering ed avvio le DPU. (Round 2)
DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY2, NULL));
DPU_ASSERT(dpu_launch(dpu_set, DPU_SYNCHRONOUS));
```

Le funzioni `dpu_load(...)` ed `dpu_launch(...)` permettono rispettivamente di caricare e di eseguire il codice DPU kernel all'interno delle DPU. L'esecuzione può avvenire sia in modalità sincrona che in modalità asincrona.

I parametri `DPU_BINARY1` ed `DPU_BINARY1` sono delle stringe contenenti i path degli eseguibili dei DPU kernel.

Chiamate ripetute a queste funzioni permettono di sfruttare più round PIM.
In particolare l'algoritmo presentato, come descritto nella sezione 3.3, sfrutta due round PIM: il primo per calcolare i centri del clustering ed il secondo per calcolare il costo del clustering effettuato.

L'utilizzo di esecuzioni multiple permette di sfruttare la persistenza delle risorse di sistema all'interno delle DPU. Numerose risorse tra cui variabili globali, primitive di sincronizzazione e l'intero contenuto della memoria MRAM non sono infatti rilasciate dalla Runtime library al termine di un'esecuzione.

All'interno del secondo round ogni DPU può quindi calcolare il costo del clustering per il proprio sottoinsieme di punti senza necessitare di ricaricarli nuovamente dalla memoria centrale.

3.4.5 Calcolo dei centri finali

```
//Estrae "n_points" centri da "points_buffer" inserrendoli in "centers_buffer".
static void get_centers(T* point_buffer, T* centers_buffer, uint32_t n_points,
                        uint32_t n_centers, uint32_t dim, uint32_t first_offset) {

    //Scelgo casualmente il primo centro.
    //Il parametro first_offset e' un indice randomico generato dal chiamante.
    uint32_t n_centers_found = 1;
    for (unsigned int i = 0; i < dim; i++) {
        centers_buffer[i] = point_buffer[i + first_offset];
    }

    while (n_centers_found < n_centers) {
        D candidate_center_dist = 0;

        for (unsigned int i = 0; i < n_points; i++) {
            uint32_t point_index = i*dim;
            D min_center_dist = INIT_VAL;

            for(unsigned int j = 0; j < n_centers_found; j++) {
                uint32_t center_index = j*dim;
                D dist = 0;

                //Calcolo distanza di Minkowsky
                for (unsigned int k = 0; k < dim; k++) {
                    dist += power(point_buffer[point_index+k],
                                  centers_buffer[center_index+k], dim);
                }
                min_center_dist = (dist < min_center_dist) ? dist : min_center_dist;
            }

            if (candidate_center_dist <= min_center_dist) {
                candidate_center_dist = min_center_dist;
                for (unsigned int k = 0; k < dim; k++) {
                    centers_buffer[n_centers_found*dim + k]=point_buffer[point_index+k];
                }
            }
        }
        n_centers_found++;
    }
}
```

Una volta recuperati i centri dalle DPU l'host application si occupa di calcolare i centri finali. Il codice riportato implementa Farthest-First Traversal utilizzando la distanza di Minkowsky per il calcolo della distanza tra i punti dell'insieme ed i vari centri.

3.5 DPU Kernel

Il DPU kernel contiene il codice eseguito dalle DPU all'interno dei vari round PIM.
In ogni DPU questo codice viene eseguito in parallelo nei vari tasklet attivi.

Entrambi i round sfruttano del codice simile a quello riportato a seguire per gestire la concorrenza tra tasklet e per inizializzare adeguatamente le risorse del sistema.

In particolare la funzione `me()` definita nella Runtime library restituisce l'ID del tasklet corrente. Questo ID può essere utilizzato per rendere esclusive alcune parte di codice, come avviene nell'esempio sottostante dove il tasklet avente ID pari a 0 (unico tasklet sempre presente) viene utilizzato per inizializzare le variabili condivise.

La funzioni `mem_reset()` ed `mram_read(...)` permettono invece rispettivamente di resettare la WRAM e di copiare dei dati dalla MRAM in un buffer WRAM.

All'interno dell'architettura UPMEM le operazioni di moltiplicazione e di divisione risultano computazionalmente molto costose [25]. Per questo motivo, quando possibile, all'interno del codice a seguire queste saranno sostituite da operazioni di bit shift.

```
//Variabili condivise.
__host struct dpu_arguments_t DPU_INPUT_ARGUMENTS;
BARRIER_INIT(my_barrier_1, NR_TASKLETS);
static T* centers_set;
static D max_distance = 0;
...

int main() {
    unsigned int tasklet_id = me();
    uint32_t n_points = DPU_INPUT_ARGUMENTS.n_points_dpu_i; //Recupero parametri
    ....

    if (tasklet_id == 0) {
        mem_reset();
        centers_set = (T*) mem_alloc((n_centers*dim) << T_SHIFT);

        //Scelta casuale del primo punto.
        mram_read(DPU_MRAM_HEAP_POINTER + first_offset, centers_set,
                  roundup((dim << T_SHIFT), 8));

        //Resetto varibili globali.
        n_centers_found = 1;
        max_distance = 0;
    }

    barrier_wait(&my_barrier_1); //Fine inizializzazione
...
```

3.5.1 Primo round DPU: calcolo dei centri intermedi

Codice usato per calcolare i centri intermedi di ogni DPU seguendo la logica descritta nella sezione 3.3.

La funzione `get_furthest_point(...)` trova il punto avente distanza massima dai centri già presenti tramite una logica simile a quella utilizzata in 3.4.5 per trovare il prossimo centro candidato.

```
while (n_centers_found < n_centers) {
    D candidate_center_distance = 0;
    for(uint32_t point_index = base_block_addr; point_index < last_point_addr;
                                                point_index += buffer_size*NR_TASKLETS){
        uint32_t l_size_bytes = buffer_size;
        uint32_t points_block_i = points_per_block;

        if (point_index + buffer_size >= last_point_addr) {
            l_size_bytes = last_point_addr-point_index;
            l_size_bytes = (l_size_bytes%8 == 0) ? l_size_bytes: roundup(l_size_bytes,8);
            points_block_i = (last_point_addr - point_index)/(dim << T_SHIFT);
        }
        mram_read((__mram_ptr void const*) point_index, buffer, l_size_bytes);
        candidate_center_distance = get_furthest_point(buffer, points_block_i, dim,
                                            candidate_center, candidate_center_distance);
    }
    uint32_t centers_offset = n_centers_found*dim;
    if (tasklet_id == 0) {
        max_distance = candidate_center_distance;
        for (unsigned int i = 0; i < dim; i++) {
            centers_set[centers_offset + i] = candidate_center[i];
        }
        my_barrier_1.count = NR_TASKLETS; //Resetto il counter della barriera.
    }
    barrier_wait(&my_barrier_2); //Ogni tasklet ha ora il suo candidato centro

    mutex_lock(my_mutex); //Regolo accesso dei tasklet
    if (candidate_center_distance > max_distance) {
        max_distance = candidate_center_distance;
        for (unsigned int i = 0; i < dim; i++) {
            centers_set[centers_offset + i] = candidate_center[i];
        }
    }
    if (tasklet_id == 0) {
        n_centers_found++; //Aggiorno il numero di centri trovati.
        my_barrier_2.count = NR_TASKLETS; //Resetto il counter della barriera.
    }
    mutex_unlock(my_mutex);

    barrier_wait(&my_barrier_1);
}
```

3.5.2 Secondo round DPU: calcolo dei costi

Codice usato per calcolare il costo del clustering relativamente ai punti assegnati ad ogni DPU seguendo la logica descritta nella sezione 3.3.

```
D tasklet_max_distance = 0;
for(uint32_t point_index = base_block_addr; point_index < last_point_addr;
                                                point_index += buffer_size*NR_TASKLETS){
    uint32_t l_size_bytes = buffer_size;
    uint32_t points_block_i = points_per_block;

    if (point_index + buffer_size >= last_point_addr) {
        l_size_bytes = last_point_addr-point_index;
        l_size_bytes = (l_size_bytes%8 == 0) ? l_size_bytes: roundup(l_size_bytes,8);
        points_block_i = (last_point_addr - point_index)/(dim << T_SHIFT);
    }

    mram_read((__mram_ptr void const*) point_index, buffer, l_size_bytes);
    tasklet_max_distance = get_max_distance(buffer, points_block_i, n_centers, dim,
                                            tasklet_max_distance);
}
//Confronto tra distanze massima di ogni tasklet
//per ricavare costo totale della DPU
...
```

```
static D get_max_distance(T* buffer, uint32_t n_points, uint32_t n_centers, uint32_t dim,
                          D tasklet_max_distance) {

    for (unsigned int i = 0; i < n_points; i++) {
        uint32_t point_index = i*dim; //Riduco numero di moltiplicazioni effettuate.
        D min_center_dist = INIT_VAL;

        for(unsigned int j = 0; j < n_centers; j++) {
            uint32_t center_index = j*dim;
            D dist = 0;
            for (unsigned int k = 0; k < dim; k++) {
                dist += power(buffer[point_index+k], centers_set[center_index+k], dim);
            }
            min_center_dist = (dist < min_center_dist) ? dist : min_center_dist;
        }
        if (tasklet_max_distance <= min_center_dist) {
            tasklet_max_distance = min_center_dist;
        }
    }
    return tasklet_max_distance;
}
```

Riferimenti bibliografici

[1] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 316–331, 2018.

[2] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE international symposium on workload characterization (IISWC)*, pp. 56–65, IEEE, 2013.

[3] D. Pandiyan and C.-J. Wu, "Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 171–180, IEEE, 2014.

[4] A. W. Burks, H. H. Goldstine, and J. Von Neumann, "Preliminary discussion of the logical design of an electronic computer instrument," 1946.

[5] D. Efnusheva, A. Cholakoska, and A. Tentov, "A survey of different approaches for overcoming the processor-memory bottleneck," *International Journal of Computer Science and Information Technology*, vol. 9, no. 2, pp. 151–163, 2017.

[6] S. I. Association *et al.*, "International technology roadmap for semiconductors (itrs)," 1999.

[7] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[8] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: Enabling in-memory computation," *Microprocessors and Microsystems*, vol. 67, pp. 28–41, 2019.

[9] H. S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 73–78, 1970.

[10] S. S. T. A. JEDEC, *High Bandwidth Memory (HBM) DRAM*. JEDEC Solid State Technology Association, 2015.

[11] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–29, 2016.

[12] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions," *arXiv preprint arXiv:1802.00320*, 2018.

[13] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 2–13, 2009.

[14] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," *SIGARCH Comput. Archit. News*, vol. 37, p. 14–23, jun 2009.

[15] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 256–267, 2013.

[16] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on circuit theory*, vol. 18, no. 5, pp. 507–519, 1971.

[17] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2016.

[18] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky, "Logic operations in memory using a memristive akers array," *Microelectronics Journal*, vol. 45, no. 11, pp. 1429–1437, 2014.

[19] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-memory: A workload-driven perspective," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 3–1, 2019.

[20] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture," *arXiv preprint arXiv:2105.03814*, 2021.

[21] "UPMEM host library." `https://sdk.upmem.com/2021.3.0/203_HostAPI.html`.

[22] "UPMEM runtime library." `https://sdk.upmem.com/2021.3.0/202_RTL.html`.

[23] T. F. Gonzalez, "Clustering to minimize the maximum intercluster distance," *Theoretical computer science*, vol. 38, pp. 293–306, 1985.

[24] "UPMEM coding tips." `https://sdk.upmem.com/2021.3.0/fff_CodingTips.html#multi-threaded-programs-are-more-efficient-than-single-threa`

[25] "UPMEM coding tips." `https://sdk.upmem.com/2021.3.0/fff_CodingTips.html#bit-variables-are-expensive`.