



UNIVERSITÀ DEGLI STUDI DI FERRARA

---

OTTIMIZZAZIONE DI UN CODICE  
LATTICE BOLTZMANN PER  
INTEL XEON PHI  
*‘KNIGHTS LANDING’*

---

*Relatore:*

**Sebastiano Fabio  
SCHIFANO**

*Correlatore:*

**Alessandro  
GABBANA**

*Laureando:*  
**Giovanni PAGLIARINI**

CORSO DI LAUREA IN INFORMATICA

ANNO ACCADEMICO 2017 – 2018

---

# *Indice*

---

	Page
<b>INTRODUZIONE</b>	<b>5</b>
<b>1 L’architettura</b>	<b>6</b>
1.1 Trend . . . . .	6
1.1.1 Parallelismo . . . . .	7
1.1.2 Acceleratori . . . . .	7
1.1.3 Limiti di trasferimento . . . . .	8
1.1.4 Modelli di programmazione parallela . . . . .	9
1.2 OpenMP . . . . .	10
1.2.1 Multi-Threading . . . . .	10
1.2.2 SIMD . . . . .	11
1.3 Intel Xeon Phi . . . . .	12
1.3.1 Knights Corner . . . . .	12
1.3.2 Knights Landing . . . . .	13
<b>2 Il Benchmark</b>	<b>17</b>
2.1 Metodi reticolari Lattice Boltzmann . . . . .	17
2.2 Benchmark 1: Vortici di Taylor-Green . . . . .	20
2.3 Benchmark 2: Flusso di Couette . . . . .	21

<b>3 L'implementazione</b>	<b>23</b>
3.1 Codice base . . . . .	23
3.1.1 <code>stencil</code> . . . . .	26
3.1.2 <code>propagate</code> . . . . .	27
3.1.3 <code>collide</code> . . . . .	27
3.2 Layout di memoria . . . . .	28
3.2.1 <i>AoS</i> . . . . .	28
3.2.2 <i>SoA</i> . . . . .	29
3.2.3 <i>CSoA</i> . . . . .	31
3.2.4 <i>CAoSoA</i> . . . . .	31
3.3 Multi-threading . . . . .	32
3.4 Vettorizzazione . . . . .	33
3.5 Streaming Stores . . . . .	33
3.6 Prefetch Software . . . . .	34
3.7 Roofline Analysis . . . . .	36
3.8 Analisi Consumi e Frequency Scaling . . . . .	38
<b>4 I Risultati</b>	<b>40</b>
4.1 Parallelizzazione . . . . .	41
4.1.1 Multi-threading e vettorizzazione . . . . .	41
4.1.2 Hyper-Threading . . . . .	43
4.2 Streaming Stores . . . . .	45
4.3 Vector Length . . . . .	46
4.4 Prefetch Software . . . . .	47
4.5 Flat Mode vs. Cache Mode . . . . .	50
4.6 Frequency Scaling . . . . .	52
<b>5 Le Conclusioni</b>	<b>54</b>
<b>Bibliografia</b>	<b>57</b>

*alle mie famiglie  
che continuano a sostenermi*

---

# *INTRODUZIONE*

---

In questo lavoro di tesi viene esposta l'implementazione e l'ottimizzazione di un codice tridimensionale per simulazioni reticolari di Boltzmann su architettura *Intel Xeon Phi* di seconda generazione, denominata *Knights Landing*.

Particolare attenzione è data all'ottimizzazione dei pattern di accessi alla memoria, sia mediante l'implementazione di data-layout che rendono efficiente l'utilizzo della cache, sia mediante l'utilizzo di particolari aspetti dell'architettura in questione, come l'impiego di istruzioni vettoriali non-temporali e il prefetch dei dati.

Per la valutazione delle prestazioni viene impiegato come benchmark un codice di fluidodinamica, le cui due principali routine di, *propagazione* e *collisione*, essendo rispettivamente *memory-intensive* e *cpu-intensive*, forniscono un test rappresentativo di quanto efficacemente le varie ottimizzazioni introdotte impattino l'utilizzo della memoria e dell'unità di calcolo del processore.

Il testo è strutturato nel seguente modo: nel Capitolo 1 verrà presentata l'architettura *Knights Landing (KNL)* assieme ad una panoramica dei principali trend nel calcolo ad alte prestazioni. Il Capitolo 2 è una breve introduzione ai metodi reticolari di Boltzmann (*LBM*) in cui vengono in particolare destritti due semplici benchmark utilizzati per la valutazione delle prestazioni. I Capitoli 3 e 4 mostrano rispettivamente l'implementazione del codice, assieme alle ottimizzazioni e agli strumenti utilizzati per l'analisi, ed i risultati ottenuti. Conclusioni e possibili sviluppi futuri sono infine presentati nel Capitolo 5.

## *Capitolo 1*

---

# *L'architettura*

---

### **1.1 Trend**

L'evoluzione dei processori in quanto a potenza di calcolo ha avuto un andamento esponenziale per almeno mezzo secolo. Questo processo si è scontrato più volte con limiti fisici, in fase di produzione e di funzionamento dei componenti, ma limiti recenti hanno motivato particolarmente lo studio di approcci diversi per aumentare la capacità di calcolo.

Figura 1.1 rappresenta l'andamento di alcune caratteristiche dei processori prodotti a partire dagli anni settanta, e delinea una transizione nella ricerca delle prestazioni. Fino all'anno 2005 circa, la frequenza di clock dei processori svolgeva un ruolo determinante delle performance. In seguito, a causa degli elevati consumi energetici e delle problematiche legate alla dissipazione del calore, questo processo ha subito un drastico rallentamento, con la frequenza di clock del singolo core rimasta da allora praticamente costante.

### 1.1.1 Parallelismo

Per continuare a soddisfare la necessità di disporre di strumenti di calcolo sempre più potenti, negli ultimi decenni sono stati sviluppati maggiormente aspetti architetturali, già disponibili da diversi decenni, relativi al parallelismo: da un lato aumentando il numero di *processing units* per supportare l'esecuzione simultanea di più processi a livello hardware, dall'altro unità vettoriali in grado di replicare una stessa istruzione su diversi input in parallelo. Innanzitutto sono state introdotte sullo stesso chip più unità di elaborazione in grado di eseguire indipendentemente. Inoltre, con il *Simultaneous Multi-Threading (SMT)*, un solo core può gestire con efficienza più contesti thread, il che permette un'utilizzo ottimizzato delle risorse computazionali disponibili. La scelta di avere diversi core a voltaggio più basso, piuttosto che uno solo molto potente, riflette anche la necessità di ridurre i costi di consumo: nel grafico vediamo che, in concomitanza dell'appiattimento di clock e consumi, sono comparse sul mercato CPU multi-core.

Per ultimo, la replicazione parallela di circuiti ha permesso l'esecuzione della stessa operazione su più dati, *Single Instruction Multiple Data (SIMD)*, con istruzioni vettoriali, che nello stesso ciclo di clock CPU operano su 2,4,8,16 dati in input in contemporanea, aumentando notevolmente il throughput.

### 1.1.2 Acceleratori

A partire dal 2006 si è diffuso l'utilizzo di unità di elaborazione esterne ai processori, dedicate al calcolo altamente parallelo a basso consumo. Inizialmente gli acceleratori consistevano in GPU, unità nate per la *computer graphics* ma con caratteristiche architetturali particolarmente vantaggiose per il calcolo parallelo in generale. Nel tempo, si sono evolute le GPU in strumenti General-Purpose (*GPGPU*), utilizzabili programmando l'offload dell'esecuzione e capaci di elevate prestazioni a costi ridotti.

L'architettura degli acceleratori sta nel paradigma *many-core*: centinaia/migliaia di core a voltaggio basso sullo stesso chip, con unità di calcolo vettoriale, e una gestione ottimizzata di parallelismo thread-level, a discapito di prestazioni single-core.

L'utilizzo di acceleratori presenta difficoltà nella programmazione. Innanzitutto, il campo di applicazione è ristretto alle sole applicazioni che presentano

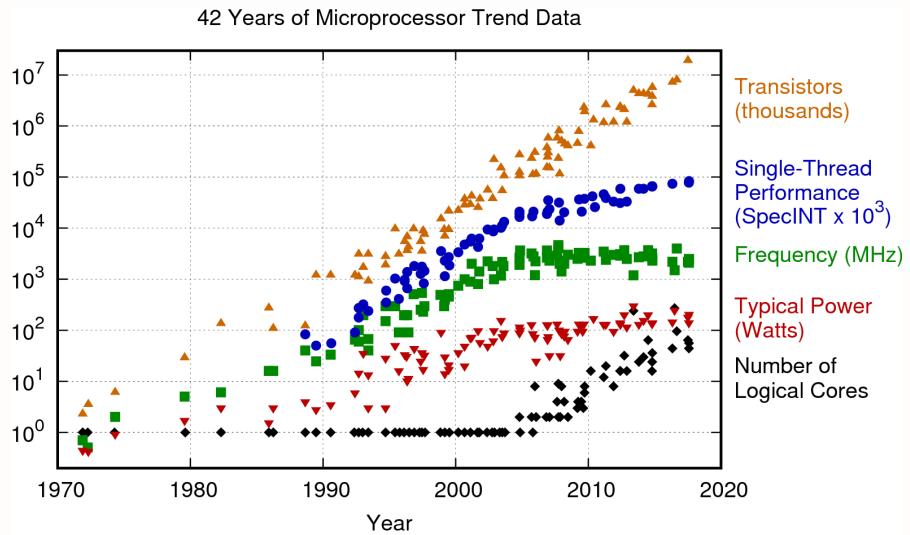


Figura 1.1: Trend di alcune caratteristiche dei processori nell’arco di 42 anni [1]. Attorno ai primi anni duemila, limiti fisici e di costi, assieme ad alte richieste di calcolo, hanno motivato una maggiore attenzione ad aspetti relativi al parallelismo e a tecnologie a basso consumo nella progettazione dei processori.

un certo livello di riuso dei dati, regolarità di controllo e accessi alla memoria. Inoltre, un’efficiente parallelizzazione del codice deve tenere conto di aspetti progettuali dell’architettura target, il che si scontra con la portabilità di codice. Per astrarre da questo ultimo problema, assieme ad architetture parallele e scalabili sono nati standard e strumenti per la divisione del carico di lavoro, come *MPI*, *OpenMP* e *OpenACC*.

### 1.1.3 Limiti di trasferimento

Un importante limite che grava su tutti i sistemi di elaborazione sta nel divario secolare tra velocità di elaborazione e banda di trasferimento elaboratore-memoria.

Nel tempo, infatti, la banda di memoria ha presentato limitazioni più stringenti rispetto alle CPU, e non si è evoluta con la stessa rapidità [2]. Innanzitutto, la memoria è più distante dall’unità di calcolo rispetto ai registri del processore, e quindi la propagazione del segnale elettrico impiega più tempo. Aumentare il clock della memoria per massimizzare il data-rate aumenta i consumi energetici, e in più, oltre un certo valore, si verifica il deterioramento del segnale. Per mitigare

il problema con il tempo sono stati aggiunti vari livelli intermedi di memorie più piccole e veloci e tra il processore e la memoria. Un esempio di una tipica gerarchia di memoria è mostrato in Fig. 1.2. Un'altra parziale soluzione al problema è data dal replicare in parallelo i canali di trasferimento, il cui numero è comunque limitato da fattori di progettazione [3].

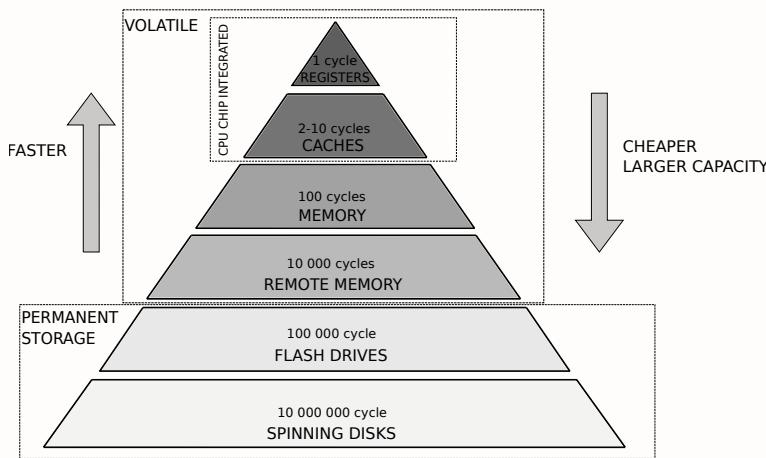


Figura 1.2: Diagramma rappresentante il rapporto tra dimensione e latenza delle memorie. Le memorie più veloci sono quelle on-chip: registri della CPU e cache, a seguire le RAM. Le memorie non-volatili sono le più capienti ma i tempi di trasferimento sono lunghi. Notare che tra un livello del diagramma e il successivo si perde un ordine di grandezza in termini di cicli di attesa.

### 1.1.4 Modelli di programmazione parallela

I thread sono le unità schedulabili indipendentemente dall'OS, che a runtime ne gestisce l'esecuzione assegnandoli ai core secondo certe politiche. Di norma la parallelizzazione a livello thread avviene utilizzando funzioni primitive apposite per indicare punti di fork/join o di comunicazioni inter-thread, e tradizionalmente lo standard di riferimento per multi-threading è stata l'interfaccia *POSIX thread*. Tuttavia, pattern implementativi primitive-based per il calcolo parallelo ostacolano la *manutenibilità* del codice, richiedendo conoscenza dell'architettura in uso e risultando in genere in difficoltà di riadattamento di codice.

Con la crescente eterogeneità di architetture per *HPC* è nata la necessità di scrivere codice parallelo ad un livello di astrazione più alto, e si sono diffusi modelli

di programmazione directive-based, come *OpenMP* e *OpenACC*.

## 1.2 OpenMP

L'API *OpenMP* (*Open Multi-Processing*) nasce per rafforzare portabilità e scalabilità di codice.

### 1.2.1 Multi-Threading

Per la parallelizzazione thread-level presenta un interfaccia semplice: anziché esplittare creazione, comunicazioni e distruzione dei thread, poniamo delle singole direttive che suggeriscono al compilatore quali regioni di codice sono parallele e in che modo in che modo spartire il workload. Un esempio è riportato in Figura 1.3, dove le  $N$  iterazioni di un ciclo vengono spartite tra i thread disponibili con una sola riga di codice. L'esempio rappresenta bene l'elasticità del paradigma directive-based: nonostante il compilatore sia in grado di operare ulteriori ottimizzazioni se i parametri sono noti in fase di compilazione, molti dettagli (primo tra tutti il numero di thread) sono astratti a compile-time e possono essere impostati in ambiente di esecuzione.

```

1 int i;
2 int a[N], b[N];
3
4 #pragma omp parallel for
5 for (i = 0; i < N;++)
6     a[i] = b[i];

```

Figura 1.3: Parallelizzazione di un ciclo con *OpenMP*. La direttiva prima del ciclo for è sufficiente per indicare al compilatore di spartire le iterazioni tra i thread. Come risultato, più thread lavoreranno simultaneamente su aree diverse di **a** e **b**. Parametri come il numero di thread e le modalità di spartizione possono essere specificati in linea, ma non sono obbligatori.

### Divisione del Workload

Nella parallelizzazione di un ciclo abbiamo diverse modalità di assegnare  $N$  iterazioni a  $t$  thread diversi. Per quello che interessa qui, la politica di assegnazione può

essere *statica* o *dinamica* (Fig. 1.4). Nel primo caso il lavoro viene diviso in blocchi da  $N/t$  iterazioni consecutive, e ad ogni thread viene assegnato un blocco. La politica dinamica, invece, assegna inizialmente ad ogni thread, e a seguire ad ogni thread appena liberato, un gruppo di iterazioni consecutive (chiamato *chunk*), di dimensione costante  $ch$ .

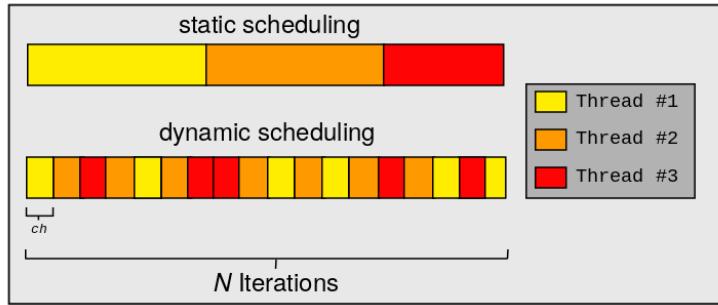


Figura 1.4: Differenza tra politiche di assegnazione iter/thread, esempio con 3 thread ( $t = 3$ ). Scheduling statico divide le  $N$  iterazioni in 3 blocchi consecutivi (quanto più possibile di uguale cardinalità) e ne assegna uno ad ogni thread. Con una politica dinamica, invece, la divisione avviene in blocchi da  $ch$  iterazioni; inizialmente si assegnano i primi 3 staticamente, e, successivamente, ogni volta che un thread si libera gli assegna un nuovo blocco, fino a terminare le iterazioni.

In generale è preferibile scheduling statico se il workload per iterazione è costante. Lo scheduling dinamico comporta maggiori overhead, ma può essere vantaggioso in casi in cui il workload omogeneo con un tempo di esecuzione molto diverso tra le varie istanze della regione da parallelizzare.

### 1.2.2 SIMD

Dalla versione 4.0 (Luglio 2013 [4]) OpenMP offre anche supporto alla programmazione *SIMD*. A scopo esemplificativo, Figura 1.5 mostra la vettorizzazione di un ciclo: la direttiva `omp simd` suggerisce al compilatore di generare istruzioni vettoriali, che traversino le singole iterazioni operando su più dati. Il compilatore `omp-unaware` si limiterà ad ignorare il suggerimento, mentre compilatori come `icc` cercheranno di “tassellare” l’esecuzione del ciclo con istruzioni agenti su vettori di `a` e `b`.

```

1 double *a, *b;
2 posix_memalign((void *)&a, 4096, N);
3 posix_memalign((void *)&b, 4096, N);
4
5 #pragma omp simd aligned(a, b: 4096)
6 for(i = 0; i < N; i++)
7   a[i] = b[i]*2;

```

Figura 1.5: Vettorizzazione di un ciclo con OpenMP. Di norma il ciclo verrebbe tradotto nell'esecuzione sequenziale di  $N$  moltiplicazioni in doppia precisione, mentre il pragma suggerisce, se l'instruction set lo permette, l'uso di  $N/VL$  *multiply* vettoriali (con  $VL$  lunghezza del vettore), che compiono lo stesso lavoro richiedendo meno cicli di clock. In alcune architetture l'allocazione allineata migliora ulteriormente le performance grazie ad una implementazione ottimizzata.

Come per il multi-threading, teoricamente ci si aspetta che le performance migliorino di un fattore pari alla dimensione dei vettori, ma la complessità dei processori pone diversi ostacoli a questo trend ideale. Su alcune architetture un fattore che migliora le prestazioni della vettorizzazione è l'**allineamento** degli indirizzi: istruzioni vettoriali di load e store sono possono essere implementate in maniera più efficiente se le locazioni di memoria coinvolte sono allineate ad (= multiple di) un certo valore, tipicamente 64 Byte [5]. Nell'esempio viene allocata memoria allineata con `posix_memalign`.

## 1.3 Intel Xeon Phi

### 1.3.1 Knights Corner

Dal tentativo di *Intel* di immettersi nel mercato delle GPU, viene lanciata nel 2013 la prima architettura many-core con il nome di *Xeon Phi coprocessor*. Si tratta di un coprocessore indirizzato al calcolo ad alte prestazioni che presenta un alto livello di parallelismo. Deriva da un progetto abbandonato di una GPU (*Larrabee* [6]), e a seconda del modello mette a disposizione fino a 60 core, in grado di supportare in *Hyper-Threading* fino a  $\sim$ 240 OS thread. Ogni core consiste in un'implementazione low-power derivata dell'architettura Pentium, per cui l'instruction set è compatibile con x86(-64). Inoltre, con l'estensione *AVX2* supporta istruzioni vettoriali fino a

256 bit, raggiungendo un picco computazionale ideale attorno a 2 TFlops per aritmetica Single-Precision, 1 TFlops per Double-Precision.

*Knights Corner* è stato impiegato nel progetto di *Tiahne-2*, supercomputer cinese che per due anni e mezzo è rimasto al primo posto nella classifica dei supercomputer più performanti [7]. Tuttavia, il suo successo è stato limitato da difficoltà nell'estrarne buone prestazioni, e da alcune lacune nel supporto, ad esempio, di istruzioni vettoriali non-allineate o di esecuzione *out-of-order*. Ciò nonostante, ha aperto la strada a nuove necessità che la generazione successiva, *Knights Landing*, ha tentato di assecondare [8].

### 1.3.2 Knights Landing

Disponibile dal 2016, *Knights Landing (KNL)* è la seconda generazione di architetture *Xeon Phi*. Rispetto alla precedente *Knights Corner* introduce diverse innovazioni, prima fra tutte la possibilità di essere utilizzato come *main processor*: *Knights Landing* nasce proprio dalla necessità di un'architettura *MIC* standalone che rimuova i colli di bottiglia dati dai trasferimenti *PCIe* [8]. Inoltre, introduce un'organizzazione di memoria configurabile che offre due tipi di RAM, a soddisfare le richieste di capacità e velocità di trasferimento. Altri miglioramenti includono un aumento delle prestazioni single-thread di un fattore 3x (6+ TFLOPs Single Precision, 3+ TFlops DP) rispetto alla generazione precedente ed un instruction set più ampio, assieme ad una gestione elastica di registri vettoriali.

Un grosso vantaggio che *Knights Landing* porta con sè è un maggiore sostegno alla portabilità di codice, poiché implementa, oltre ai modelli “offload” tipici degli acceleratori, gli stessi modelli di programmazione parallela delle tradizionali architetture multi-core. Infine, l'architettura di memoria e la mesh di coerenza cache sono configurabili in diverse modalità per il tuning delle performance. Di seguito riportiamo le caratteristiche architettoniche principali; si prenda Figura 1.6 come riferimento per la struttura.

## CPU

Il nucleo della computazione consiste in una maglia bidimensionale di tasselli interconnessi; ogni tassello ospita 1MB di Cache L2 e due core, aventi due unità per calcolo Floating-Point e vettoriale (*VPU*) ciascuno. La coerenza delle cache

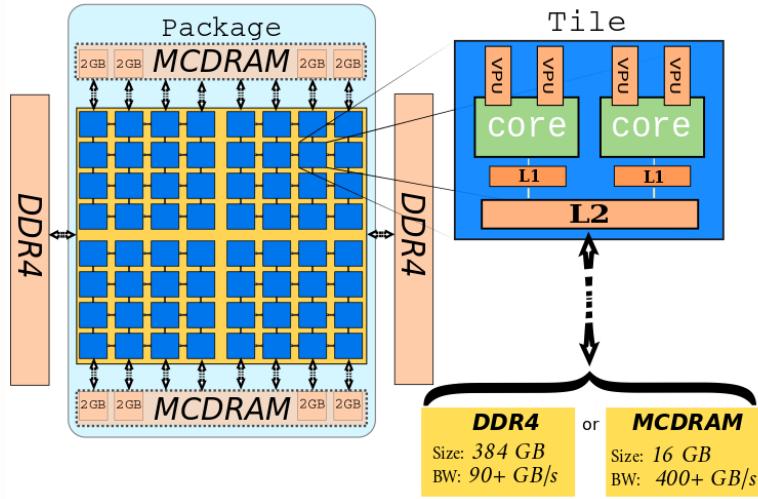


Figura 1.6: Struttura *Knights Landing*: Il chip comprende  $\sim 60$  core raggruppati a due a due in tile. Ogni tile possiede una cache L2, ed è collegato agli altri tile tramite una maglia di collegamenti. Inoltre, ogni core ha, oltre ad una propria cache L1, un'unità dedicata per il calcolo vettoriale. *KNL* offre due memorie: una on-package veloce (*MCDRAM*) ed una lenta. L'architettura di memoria è configurabile in diverse modalità, per cui si può scegliere come utilizzare le due risorse.

L2 è mantenuta tramite una mesh 2D di collegamenti, che può essere configurata secondo diverse modalità per ricercare le performance migliori. In questa tesi useremo solo la modalità *Quadrant*, nella quale la mesh è divisa in quattro quadranti, ed ognuno si interfaccia alla memoria tramite uno dei 4 memory controller. Sulla CPU in uso (*Xeon Phi<sup>TM</sup> CPU 7230*) la mesh è composta da 64 core a 1.30-1.50 GHz [9], che con *Hyper-Threading* fino ad un fattore 4 (come per Knights Corner), offrono un totale di 256 thread del sistema operativo.

## VPU

Nel nuovo set di istruzioni *AVX-512* troviamo operazioni vettoriali fino a 512 bit, contro i 256 bit del precedente *AVX2*, ed abbiamo inoltre la possibilità di gestire i registri vettoriali in maniera flessibile. Rispetto a *KNC*, con l'aggiunta di registri logici e masking di istruzioni, permette di utilizzare, ad esempio, un registro da 512 bit come un vettore di 16 interi/single-precision, 8 double-precision, oppure come 2 vettori a 256 bit. Una VPU ha un throughput di una istruzione

AVX-512 per ciclo, quindi, se consideriamo l'impiego di *FMA* (*Fused Multiply-Add*), ogni core è in grado di eseguire operazioni su 32 double-precision per ciclo. AVX-512 include anche specifiche istruzioni vettoriali algebriche non banali, di scatter/gather, conflict detection.

## Memorie

Il processore è inclusivo di una memoria veloce on-package, chiamata *MCDRAM* (*Multi-Channel RAM*). Questa è costituita da otto dispositivi da 2GB ciascuno (per un totale di 16GB), ognuno dei quali ha il proprio memory controller. La MCDRAM costituisce una novità poiché garantisce una picco di banda aggregata pari a  $450\text{GB}/\text{s}$ , contro i  $320\text{GB}/\text{s}$  di *KNC*. Nonostante l'elevata velocità di trasferimento, anche questa memoria può diventare facilmente un ostacolo per le performance [10]. Oltre a questa, troviamo una RAM *DDR4* da 384GB off-package, con un picco di banda molto più contenuto,  $90\text{GB}/\text{s}$ .

Come anticipato, anche le memorie sono configurabili: in particolare, dato che la MCDRAM è ottimizzata per accessi multipli piuttosto che per il singolo accesso, si è pensato alla possibilità di utilizzarla come cache per la DDR, anche parzialmente. Le modalità disponibili sono le seguenti:

- *Flat*: DDR e MCDRAM utilizzate come memorie principali;
- *Cache*: MCDRAM utilizzata come cache per la DDR;
- *Hybrid*: MCDRAM partizionata in 50%-50% o 25%-75%: la prima partizione viene utilizzata come cache, mentre la seconda, assieme alla DDR, è mappata come memoria principale.

In questa trattazione utilizzeremo principalmente la prima modalità, e brevemente la seconda a solo scopo comparativo.

In modalità flat le due memorie sono configurate come due nodi *NUMA* (*Non-Uniform Memory Access* [11]), e si ha la possibilità di impostare quale utilizzare per un dato processo (memory affinity) con il comando `numactl`.

## Core Affinity

L'assegnazione thread/core è onere del sistema operativo, ma il KNL offre l'interfaccia *KMP\_AFFINITY*, che permette di impostare a runtime la politica o il mapping di assegnazione preferiti.

La configurazione avviene tramite due variabili d'ambiente:

- *KMP\_HW\_SUBSET* indica quanti core fisici si vogliono utilizzare ed il fattore di *Hyper-Threading* (es.  $24c, 2t = 24$  core, 2 slot/core  $\rightarrow$  48 slot);
- *KMP\_AFFINITY* imposta la modalità di assegnazione thread/slot e accetta i valori (Fig. 1.7):
  - **compact** I thread vengono occupati prima gli slot del primo core, poi quelli del secondo, ecc. ;
  - **balanced** I thread vengono divisi equamente tra i core disponibili e la distribuzione avviene in modo sequenziale;
  - **scatter** I thread vengono divisi equamente tra i core disponibili e la distribuzione avviene a Round Robin.

In aggiunta, il modificatore *granularity* permette di specificare quanto stringente sia il binding; useremo una granularità a livello thread, così da evitare migrazioni di thread tra slot dello stesso core.

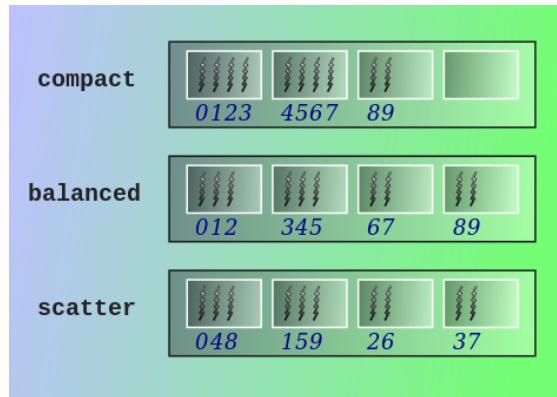


Figura 1.7: Esempio di binding thread/core con 10 thread e 4 core, usando tre politiche diverse.

## *Capitolo 2*

---

### *Il Benchmark*

---

In questo lavoro di tesi si è usato usato come benchmark un codice di fluidodinamica computazionale, basato sul metodo Lattice Boltzmann. In questa sezione diamo una breve descrizione dell'algoritmo utilizzato e dei test numerici utilizzati per garantire la correttezza del codice. Si rimanda altrove (per esempio [12]) per una dettagliata introduzione ai metodi reticolari di Boltzmann.

#### **2.1 Metodi reticolari Lattice Boltzmann**

I metodi Lattice Boltzmann (LB) sono una classe di tecniche di fluidodinamica computazionale, alternativi rispetto ad approcci standard che prevedono la discretizzazione ed integrazione diretta delle equazioni di Navier Stokes. I metodi LB, infatti, operano ad una scala intermedia, detta mesoscopica, in cui una descrizione statistica delle interazioni microscopiche permette di simulare correttamente l'evoluzione macroscopica di un fluido.

L'evoluzione temporale del sistema è descritta dall'equazione discreta di Lattice Boltzmann:

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, \mathbf{e}_i, t + \Delta t) - f_i(\mathbf{x}, \mathbf{e}_i, t) = \frac{\Delta t}{\tau} (f_i^{eq}(\mathbf{x}, \mathbf{e}_i, t) - f_i(\mathbf{x}, \mathbf{e}_i, t)) + F_i^{ext} \quad . \quad (2.1)$$

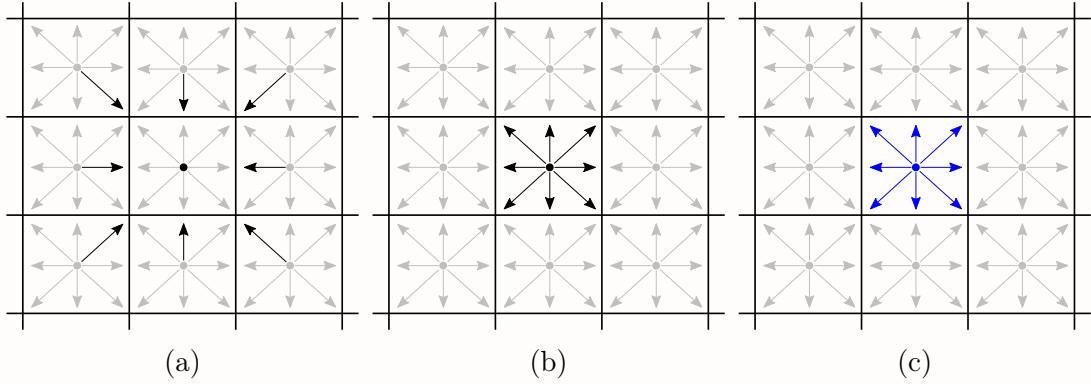


Figura 2.1: Rappresentazione grafica dello schema computazionale, per un caso bidimensionale. I vari pannelli descrivono l’algoritmo: all’istante precedente l’operazione di *streaming* (a), all’istante immediatamente successivo l’operazione di *streaming* (b), all’istante successivo l’operazione di *collide* (c).

dove:

- $f_i(\mathbf{x}, \mathbf{e}_i, t)$  è la funzione di densità di probabilità (discreta), che definisce la probabilità di osservare una particella all’istante di tempo  $t$  una particella in posizione  $\mathbf{x}$  con velocità  $\mathbf{e}_i$ ,  $\mathbf{x}, \mathbf{e}_i \in \mathbb{R}^3$ .
- $f_i^{eq}$  è un espansione in polinomi di Hermite della distribuzione di Maxwell-Boltzmann.
- $\tau$  è il tempo di rilassamento, un parametro che permette di regolare la viscosità del fluido.
- $F_i^{ext}$  descrive una forza esterna che agisce sul sistema.

Uno dei vantaggi fondamentali del metodo LB sta nella semplicità dell’algoritmo, il quale, come si può notare da Eq. 2.1, ha un carattere puramente locale, il che permette una diretta parallelizzazione. È conveniente dividere l’evoluzione del sistema in due parti: la prima, corrispondente alla parte a sinistra dell’uguale in Eq. 2.1, corrisponde ad un’evoluzione spaziale in cui le diverse pseudo-particelle vengono spostate secondo uno stencil prestabilito. Faremo riferimento a questo passo dell’algoritmo come operazione di *streaming*. La seconda, corrispondente alla parte di destra dell’uguale in Eq. 2.1, corrisponde ad un’evoluzione temporale in cui viene simulata la collisione tra le diverse particelle per determinare lo stato

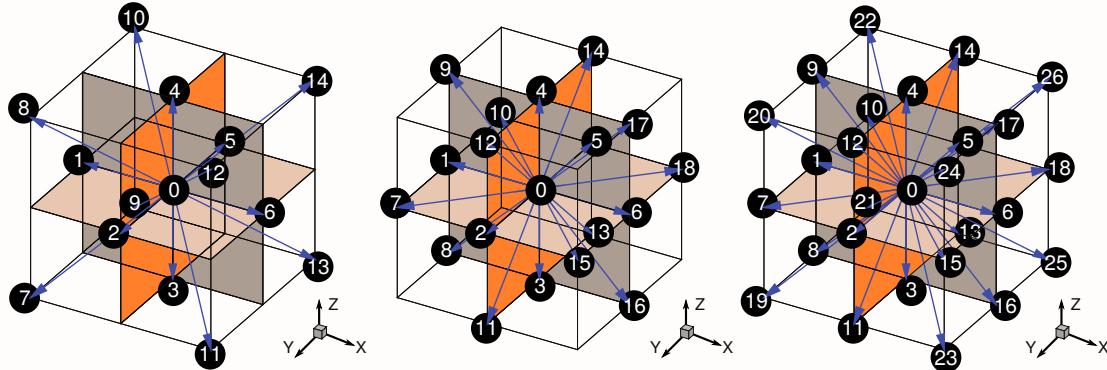


Figura 2.2: Stencil utilizzati nelle simulazioni, da sinistra a destra: D3Q15, D3Q19, D3Q27.

del sistema al tempo  $t + 1$ . Faremo riferimento a questo passo dell'algoritmo come operazione di *collide*. Una rappresentazione grafica dell'algoritmo è descritta in Fig 2.1. Esistono diversi modelli LB, cui in genere ci si riferisce usando la nomenclatura  $DdQq$ , dove  $d$  rappresenta il numero di dimensioni spaziali, mentre  $q$  rappresenta il numero di pseudo-particelle usate nella definizione dello stencil. In questo lavoro utilizzeremo tre diversi stencil tridimensionali i quali permettono la descrizione di fluidi “quasi-incompressibili” iso-termici. Gli stencil per i modelli D3Q15, D3Q19, D3Q27 sono rappresentati in Fig. 2.2. Si può dimostrare che questi modelli forniscono un’approssimazione al secondo ordine delle equazioni di Navier Stokes:

$$\nabla \cdot \mathbf{u} = 0 \quad (2.2)$$

$$n \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \nu \Delta \mathbf{u} + \mathbf{F} \quad . \quad (2.3)$$

con viscosità cinematica data da

$$\nu = \frac{1}{3} \left( \tau - \frac{1}{2} \right) \quad (2.4)$$

Nel resto del capitolo riassumiamo le condizioni iniziali e la soluzione analitica di alcuni test numerici tipicamente impiegati nella validazione di codici di fluidodinamica.

## 2.2 Benchmark 1: Vortici di Taylor-Green

I vortici di Taylor-Green sono un esempio di flusso instabile a decadimento esponenziale. In Fig. 2.3 mostriamo lo snapshot di un esempio di simulazione.

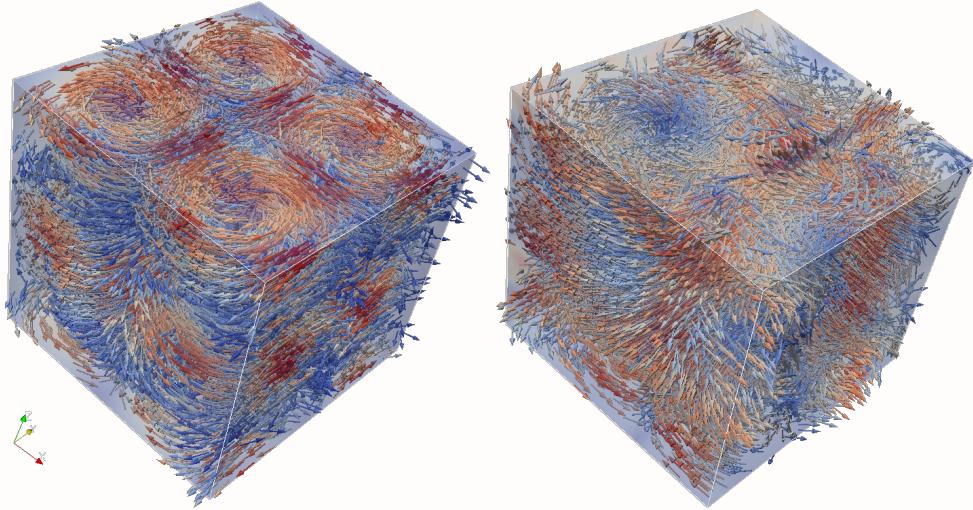


Figura 2.3: Snapshot di una simulazione dei vortici di Taylor-Green, con a sinistra lo stato iniziale del sistema e a destra uno stato intermedio.

### Condizioni iniziali

Le condizioni definiscono il profilo di velocità  $\mathbf{u} = (u_x, u_y, u_z)$  al tempo  $t = 0$ :

$$u_x = u_0 \cos ax \sin by \sin cz \quad (2.5)$$

$$u_y = u_0 \sin ax \cos by \sin cz \quad (2.6)$$

$$u_z = u_0 \sin ax \sin by \cos cz \quad (2.7)$$

con  $0 \leq x, y, z, \leq 2\pi$  e  $a, b, c \neq 0$ .

### Soluzione analitica

Introduciamo l'osservabile

$$\bar{u}^2(t) = \int (u_x^2(t) + u_y^2(t) + u_z^2(t)) dx dy dz \quad (2.8)$$

È semplice dimostrare che  $\bar{u}^2(t)$  decade proporzionalmente a una funzione  $F(t)$  definita come

$$F(t) = \exp -3\nu t \quad (2.9)$$

La validazione del codice utilizzato in questo lavoro è mostata in Fig. 2.4

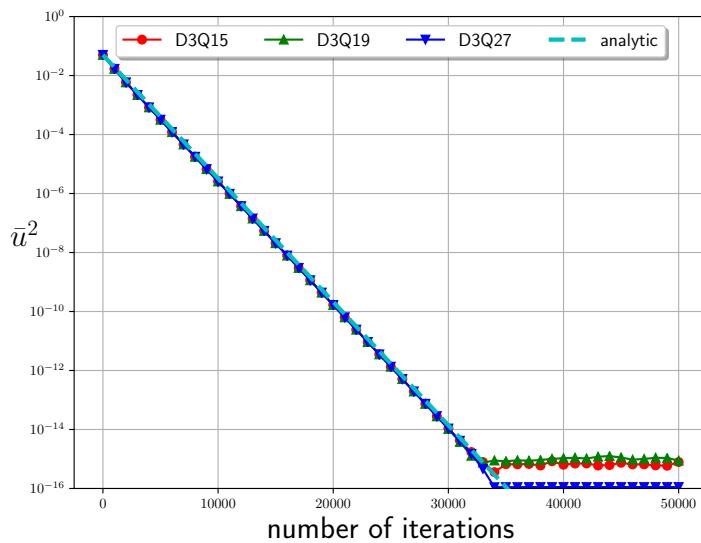


Figura 2.4: Validazione del codice in cui il decadimento dei vortici segue Eq. 2.9 fino al raggiungimento della precisione macchina per tutti gli stencil considerati.

## 2.3 Benchmark 2: Flusso di Couette

Il flusso di Couette è il flusso laminare di un fluido viscoso che si sviluppa tra due pareti piane parallele, poste a distanza  $h$ , una delle quali in moto a velocità costante  $U$  tangenzialmente rispetto all'altra parete.

### Condizioni iniziali

$$u(z, 0) = 0, \quad 0 < z < h \quad (2.10)$$

$$u(z = 0, t) = 0, \quad u(z = h, t) = U, \quad t > 0 \quad (2.11)$$

con una forza esterna costante  $\mathbf{F} = (F_x, 0, 0)$  agente sul sistema.

## Soluzione analitica

Per le condizioni iniziali sopra descritte esiste la seguente soluzione analitica stazionaria per  $t \gg$ :

$$u(z) = \frac{F_x}{2\nu} z(h - z) + U \frac{z}{h} \quad (2.12)$$

Tale equazione viene utilizzata in Fig. 2.5 per la validazione del codice.

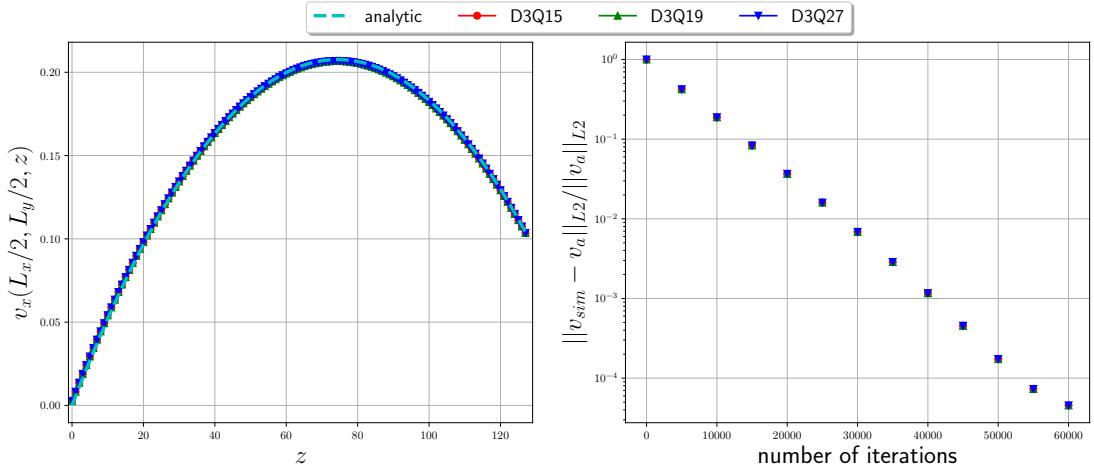


Figura 2.5: Validazione del codice. Considerando lo stato stazionario e utilizzando Eq. 2.12 vengono mostrati nel pannello di sinistra il confronto tra il profilo di velocità predetto analiticamente e quello ottenuto tramite simulazioni. Nel pannello di destra viene mostrato l'andamento temporale dell'errore relativo.

## *Capitolo 3*

---

# *L'implementazione*

---

In questa trattazione utilizziamo un codice Lattice Boltzmann in tre dimensioni. Il tuning delle performance sull'architettura prevede: da una parte, multi-threading e vettorizzazione delle operazioni algebriche per sfruttare la capacità di calcolo; dall'altra, vettorizzazione di load/store e impiego di istruzioni speciali per trarre vantaggio il più possibile della banda di memoria disponibile. Per garantire un certo livello di portabilità, dove possibile, ci rifaremo al modello di programmazione *OpenMP* (vedi Sez. 1.2).

### **3.1 Codice base**

Una simulazione LB consiste di una fase di inizializzazione in cui si assegna ad ogni pseudo-particella un valore in virgola mobile, ed una fase che simula l'avanzamento del sistema alternando per un certo numero di iterazioni le operazioni di *streaming* e *collide*. Queste leggono ed aggiornano tutto il reticolo, potenzialmente generando dipendenze se i calcoli avvengono in-place.

Un possibile modo di evitare queste dipendenze consiste nell'instanziare due reticolli, **f1** ed **f2**: il primo viene inizializzato e conterrà i risultati finali di ogni iterazione, mentre il secondo è usata come struttura di appoggio per i risultati di

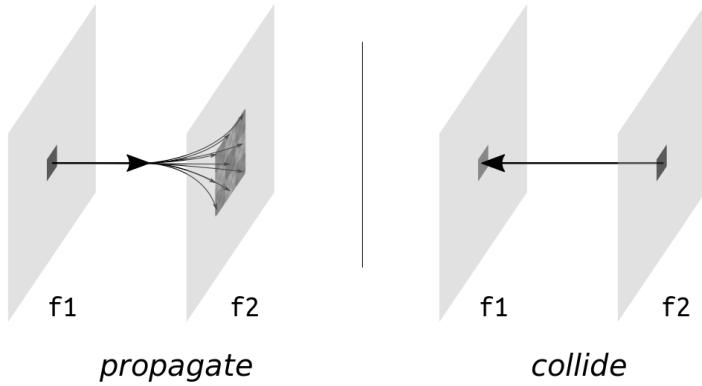


Figura 3.1: Per trattare il reticolo vengono allocate due aree di memoria separate ed eseguire le routine di propagazione e collisione out-of-place. Si inizializza il primo reticolo, dopodiché alternatamente: si propagano le particelle di **f1** memorizzando i risultati in **f2**, e si simulano le collisioni in **f2** salvando i risultati in **f1**. In figura sono rappresentate le due fasi su un sito di un lattice bidimensionale.

**propagate** (vedi Figura 3.1). Poiché **propagate** e **collide** aggiornano ogni punto del reticolo esattamente una volta e lavorano con un'area in sola lettura ed una in sola scrittura, questa soluzione elimina le dipendenze dati e consente la gestione parallela delle operazioni su siti diversi.

Il reticolo rappresenta una texture periodica di un'area infinita di fluido, per cui i siti sugli estremi virtualmente hanno come vicini siti estremi che stanno dall'altro lato della struttura. Eseguire la **propagate** tenendo conto di questi particolari non è efficiente, perché richiederebbe branch e condurrebbe a accessi alla memoria meno regolari. Piuttosto, sceglieremo di instanziare il reticolo riservando spazio per siti di bordo, a servire per questa pseudo-vicinanza. Chiamiamo la parte interna del reticolo *bulk*, e i piani di bordo esterno *halo* (Figura 3.2a). Ad ogni ciclo ci preoccupiamo di copiare i risultati della collisione dei i piani marginali (del bulk) nelle aree di halo che stanno dall'altro lato del lattice: in questo modo, quando propagheremo i bordi, i piani marginali riceveranno particelle dei siti virtualmente vicini. Figura 3.2b mostra questa strategia su un lattice bidimensionale.

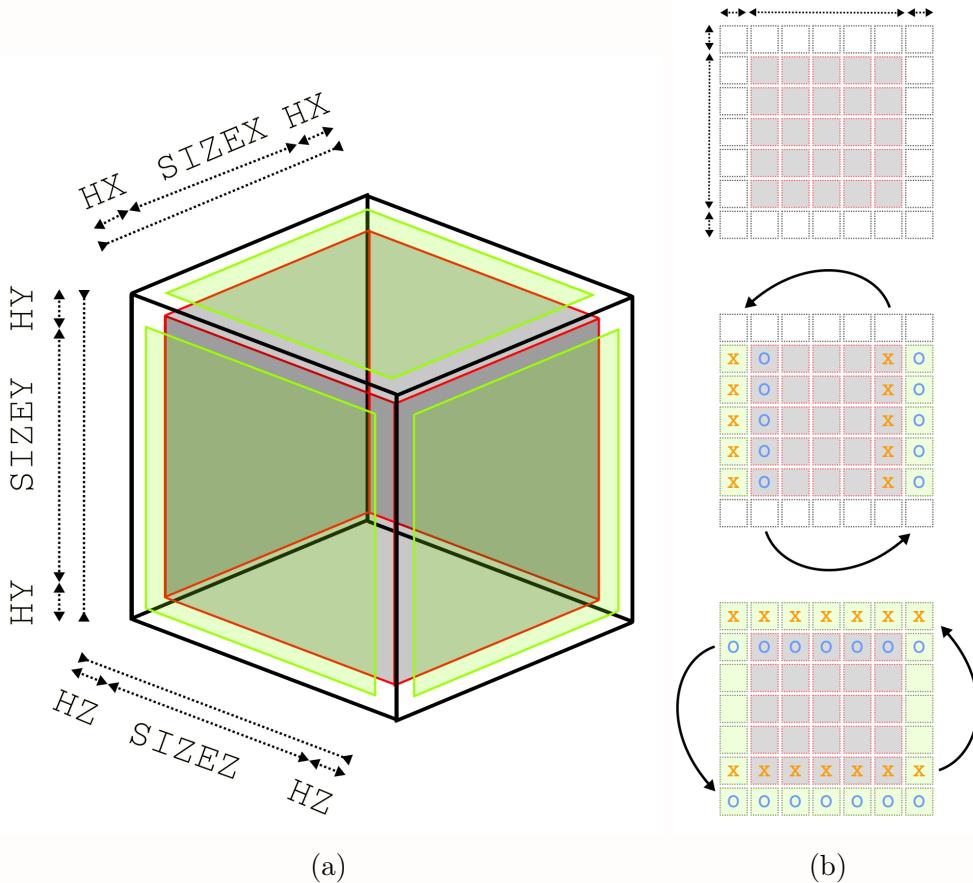


Figura 3.2: Rappresentazione del reticolo, costituito da *bulk* (indicato in grigio) e dai siti di bordo (verde), necessari per simulare con efficienza la vicinanza dei siti marginali. A sinistra è mostrata la struttura di un reticolo tridimensionale. A destra, l'operazione di copia dei bordi su un reticolo bidimensionale 7x7.

L'implementazione dell'algoritmo avrà l'aspetto del codice in Figura 3.3, dove:

- `propagate` esegue lo streaming di ogni sito;
- `collide` calcola le collisioni tra tutte le pseudo-particelle;
- `bc` esegue le operazioni necessarie alle condizioni di bordo.

```

1 // Tipo di dato popolazione
2 typedef double data_t;
3
4 // Struttura usata per il layout del reticolo
5 typedef struct {
6     data_t p[NPOP];
7 } pop_aos_t;
8
9 // Reticolo & mirror
10 pop_aos_t *f1 = (pop_aos_t *)malloc(NX*NY*NZ*sizeof(pop_aos_t));
11 pop_aos_t *f2 = (pop_aos_t *)malloc(NX*NY*NZ*sizeof(pop_aos_t));
12
13 // Inizializzazione di f1
14 init(f1) ;
15
16 for(iter = 1; iter <= NITER; iter++) {
17     ...
18
19     // Border Conditions
20     bc(f1, stencil, param, HX-HXMAX, NX-HX+HXMAX);
21
22     // Propagate f1 -> f2
23     propagate(f2, f1, stencil, HX, NX-HX);
24
25     // Collision f1 <- f2
26     collide(f1, f2, stencil, HX, NX-HX);
27
28     ...
29 }
```

Figura 3.3: L'algoritmo inizializza il reticolo, e simula l'evoluzione del sistema iterando a turno *propagazione* e *collizione* di ogni sito. Si usa una struttura ausiliaria per appoggiare i risultati intermedi tra le due operazioni, come rappresentato in Figura 3.1.

### 3.1.1 stencil

Un modello LB è definito dal numero di pseudo-particelle per sito e dalla matrice che indica come queste si propagano e interagiscono. Instanziamo questa matrice con una struttura di quattro array, nei quali per ogni popolazione memorizzeremo tre coordinate spaziali relative ed un peso, che definiscono il modo in cui ciascun sito interagisce con i vicini:

```
typedef struct {
    int     nx[NPOP]; /* neighbors x offsets */
    int     ny[NPOP]; /* neighbors y offsets */
    int     nz[NPOP]; /* neighbors z offsets */
    data_t  w[NPOP]; /* quadrature weights */
} stencil_t;
```

### 3.1.2 propagate

`propagate` è il kernel che implementa la streaming di ogni sito, ovvero ne sposta le particelle nei siti vicini secondo lo stencil del modello. Il suo funzionamento è semplice: per ogni particella  $ip$ , si preoccupa di calcolare le coordinate del sito di destinazione e di copiare il valore nella relativa popolazione  $ip$ .

```
1 for(ix = startX; ix < endX; ix++)
2     for(iy = HY; iy < (HY + SIZEY); iy++)
3         for(iz = HZ; iz < (HZ + SIZEZ); iz++) {
4
5             // dest offset
6             size_t idx_d = IDX(ix, iy, iz);
7
8             #pragma unroll
9             for(ip = 0; ip < NPOP; ip++) {
10                 int ix_s = ix - stencil->nx[ip];
11                 int iy_s = iy - stencil->ny[ip];
12                 int iz_s = iz - stencil->nz[ip];
13
14                 // source offset
15                 size_t idx_s = IDX(ix_s, iy_s, iz_s);
16
17                 nxt[idx_d].p[ip] = prv[idx_s].p[ip];
18             }
19         }
```

### 3.1.3 collide

In questa implementazione la collisione di un sito è locale: consiste nel caricarne le particelle, eseguire su di esse un certo numero di calcoli (che varia a seconda del modello e benchmark) e aggiornarne il valore.

Una piccola ottimizzazione che compiamo su entrambi i kernel consiste nell'indicare lo *srotolamento* dei loop sulle popolazioni. Conoscendo il valore di `NPOP` in compilazione, ed essendo questo valore in genere basso, non vogliamo che ad

ogni ciclo-NPOP avvenga l'inizializzazione e l'incremento delle variabili di controllo; piuttosto, preferiamo che il ciclo venga svolto a compile-time e che il compilatore codifichi istruzioni di iterazioni diverse una dopo l'altra.

```

1      // Compute density and velocity
2      rho = ux = uy = uz = 0.0;
3
4      #pragma unroll
5      for(ip = 0; ip < NPOP; ip++) {
6          rho += prv[idx].p[ip];
7
8          ux += prv[idx].p[ip] * stencil->nx[ip];
9          uy += prv[idx].p[ip] * stencil->ny[ip];
10         uz += prv[idx].p[ip] * stencil->nz[ip];
11     }
12
13     rhoi = 1./rho;
14
15     ux = rhoi*ux;
16     uy = rhoi*uy;
17     uz = rhoi*uz;
18
19     // Apply the forcing scheme and evolve the discrete LBGK equation:
20
21     #pragma unroll
22     for(ip = 0; ip < NPOP; ip++) { ... }
23
24 }
```

## 3.2 Layout di memoria

Il reticolo presenta tre dimensioni spaziali ed una aggiuntiva per l'indice di particella. In questa sezione presentiamo i diversi metodi usati per disporre i siti del reticolo in memoria.

### 3.2.1 *AoS*

L'implementazione base, come mostrata in Fig. 3.3, instanzia il reticolo come un array di siti, dove ogni sito consiste a sua volta in un array di *NPOP* pseudo-particelle; di conseguenza in aree di memoria contigue troviamo popolazioni dello stesso sito. Questa disposizione è la scelta migliore per `collide`, poiché massimizza

la località del singolo sito e permette un buon utilizzo della cache. Chiamiamo questo layout *Array of Structures (AoS)*.

### 3.2.2 SoA

Un primo approccio differente consiste nel raggruppare le particelle prima per indice e poi per sito. Questa organizzazione è concettualmente simile ad instanziare  $NPOP$  pseudo-reticolari, ognuno con sole popolazioni dello stesso indice: prima tutte le particelle di indice 0, poi quelle di indice 1, così fino a  $NPOP - 1$ . Chiamiamo *Structure of Arrays (SoA)* un reticolo instanziato in questo modo (Figura 3.4 per una rappresentazione):

```
typedef struct { data_t s[NX*NY*NZ]; } pop_soa_t;
pop_soa_t f[NPOP];
```

Questa scelta essere ragionevole se si pensa che la propagazione lascia invariato l'indice delle particelle. Come già detto, infatti, nella streaming di un sito, una particella  $ip$  viene copiata nella popolazione  $ip$  di un sito prossimo, ed in questo layout le locazioni di origine e destinazione si troverebbero più vicine, favorendo la località degli accessi in `propagate`. D'altra parte, streaming e collisione di un sito richiedono accessi a tutte le sue popolazioni, che essendo qui delocalizzate porterebbero ad una inefficienza nell'uso del caching. Per la propagazione è comunque possibile superare il problema dando un certo ordinamento agli accessi.

Memorizzare particelle con stesso indice in locazioni adiacenti comporta un altro vantaggio: poiché particelle dello stesso indice sono propagate secondo lo stesso offset, con *SoA* celle di memoria contigue richiedono di essere usate allo stesso modo, il che favorisce l'impiego di **istruzioni vettoriali**.

Per come è strutturato, nell'utilizzare questo layout ordiniamo gli accessi in lettura di `propagate` in modo da propagare prima particelle dallo stesso indice:

```
for(ip = 0; ip < NPOP; ip++)
    for(ix = startX; ix < endX; ix++)
        for(iy = HY; iy < (HY + SIZEY); iy++)
            for(iz = HZ; iz < (HZ + SIZEZ); iz++) { ... }
```

Senza questo accorgimento la cache non verrebbe utilizzata correttamente, risultando in basse prestazioni di memoria (e di calcolo di conseguenza).

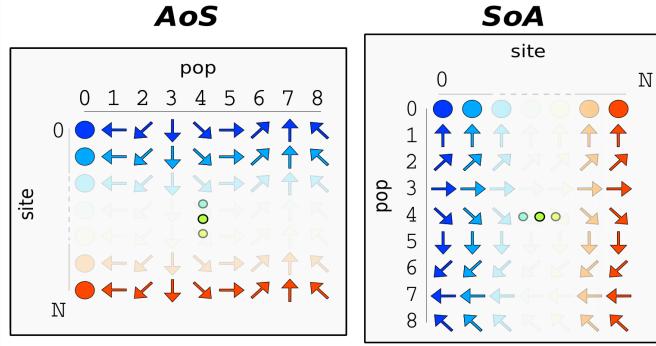


Figura 3.4: Rappresentazione in memoria di reticolli *AoS* e *SoA*, usando uno stencil semplice: **D2Q9** (Fig. 2.1). *AoS* vede interlacciate le  $NPOP$  popolazioni di uno stesso sito; *SoA* memorizza sequenzialmente popolazioni dello stesso indice di siti vicini.

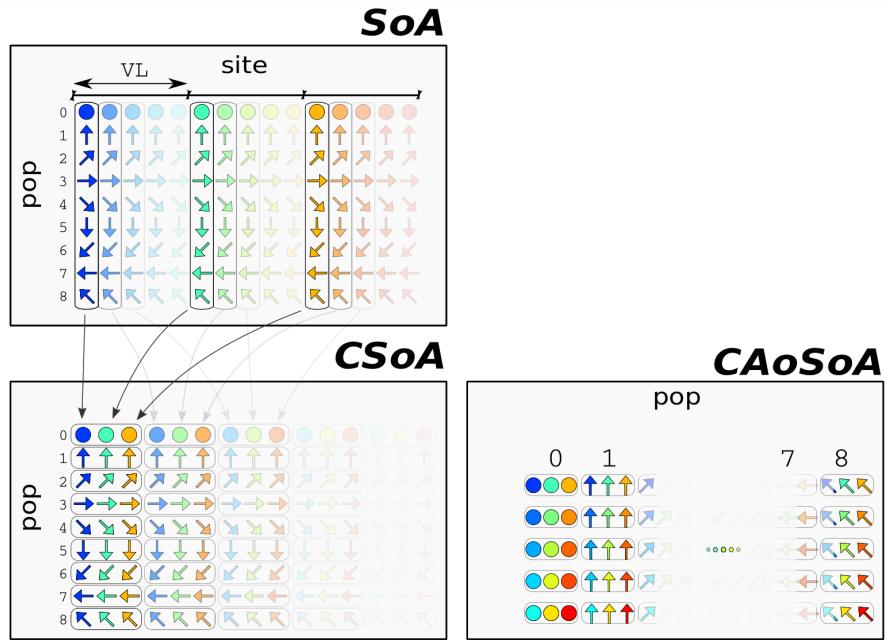


Figura 3.5: Passaggio da *SoA* a versioni clusterizzate: le righe vengono separate in  $VL$  partizioni di  $NZ/VL$  particelle, dopodiché si assemblano nello stesso “cluster” le prime particelle di ogni partizione, poi le seconde, ecc. Compiuta questa operazione possiamo ordinare i cluster prima per indice di particella, ottenendo *CSoA*, oppure prima per spazialità, salvando consecutivamente cluster di particelle diverse degli stessi siti.

### 3.2.3 CSoA

SoA favorisce la vettorizzazione delle istruzioni ma genera una potenziale inefficienza in `propagate`. Per ogni vettore di dati da spostare la destinazione in memoria potrebbe essere un indirizzo non-allineato, e, pensando ad architetture dove l'allineamento degli accessi determina un miglioramento ulteriore, possiamo preoccuparci di fare di meglio. Possiam favorire l'uso di istruzioni vettoriali **allineate** con una versione modificata di SoA che riorganizza i dati in cluster (*Clustered SoA*). L'idea (Figura 3.5) è di dividere le righe degli pseudo-reticolli SoA in partizioni di uguale dimensione, e di memorizzare le  $i$ -esime particelle di ogni partizione nello stesso vettore di dimensione VL (= partizioni per riga). Il risultato è che ogni pseudo-reticolo è tassellato di cluster contenenti particelle di siti equamente distanti fra loro; è proprio questa tassellazione che, allineato il base address del lattice, ci assicura accessi allineati.

```
#define NZoVL (NZ / VL) // Clusters per line
typedef struct { data_t c[VL]; } vdata_t;
typedef struct { vdata_t s[NX * NY * NZoVL]; } pop_csoa_t;
pop_csoa_t f[NPOP];
```

Come per SoA, propaghiamo le particelle prima per indice.

### 3.2.4 CAoSsA

Passando da AoS a SoA abbiamo separato le particelle per indice. Ora partendo dalla versione Clusterizzata di *SoA* compiamo l'operazione inversa: aggreghiamo tra loro i cluster con stessi indici spaziali ma popolazioni diverse. Otteniamo così una struttura dati doppiamente nidificata, che chiamiamo *Clustered Array of Structures of Arrays*:

```
typedef struct { vdata_t p[NPOP]; } pop_caosoa_t;
pop_caosoa_t f[NX * NY * NZoVL];
```

Questo metodo ibrida le idee alla base di SoA e CSoA: avvicinare particelle di siti diversi ma stesso indice per favorire la vettorizzazione in `propagate`, e migliorare la località degli accessi per `collide`. Figura 3.5 mostra le differenze tra i tre layout.

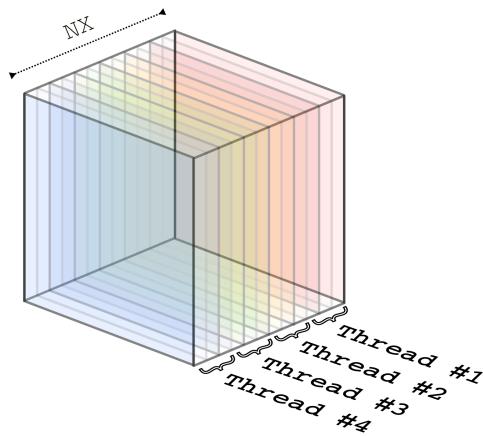


Figura 3.6: Applicazione del multi-threading: si divide il reticolo in blocchi piani, e si assegnano a thread diversi.

### 3.3 Multi-threading

L'indipendenza dei calcoli permette di aggiornare più siti in parallelo, ed avendo per mano un'architettura many-core possiamo dividere il reticolo in aree e farle processare da diversi contesti thread. Lo strumento a cui facciamo riferimento è *OpenMP* (Sezione 1.2)

Come mostra Figura 3.6, dividiamo il reticolo in piani lungo  $NX$  e assegnamo a ciascun dei thread un gruppo di piani, indicando a compile-time il numero di thread e l'assegnazione iterazione/thread:

```
#pragma omp parallel for num_threads(NUM_THREADS) schedule(SCHEDULE) \
    private(ix, iy, ic, ik, ip)
for(ix = startX; ix < endX; ix++) { /* Processo piano ix... */ }
```

Per *(C)SoA* propagare accediamo prima per indice di popolazione e poi per indici spaziali, perciò la strategia utilizzata è diversa:

```
#pragma omp parallel for collapse(3) ...
for(ip = 0; ip < NPOP; ip++)
    for(ix = startX; ix < endX; ix++) { ... }
```

Il ciclo su  $NPOP$  conta in genere poche di iterazioni; per questo motivo, con la clausola `collapse` richiediamo che la spartizione consideri anche i livelli di iterazione più interni.

In Sezione 4.1 confrontiamo i risultati ottenuti impiegando 1, 2, 3 o 4 thread per core, utilizzando politiche di assegnazione iterazione/thread statica e dinamica (spiegate in Sezione 1.2.1), e affinità bilanciata e scatterata (Sezione 1.3.2).

## 3.4 Vettorizzazione

`propagate` e `collide`, dovendo ripetere la stessa serie di operazioni su ogni sito di lattice, si prestano bene all’uso di istruzioni *SIMD*. Nella nostra implementazione vettorizziamo le righe lungo `NZ`, la dimensione spaziale più interna:

```
#pragma omp simd aligned(stencil, prv, nxt: DATA_ALIGNMENT)
for(iz = HZ; iz < (HZ + SIZEZ); iz++) // Each site in line
```

Nei layout clusterizzati esplicitiamo l’uso di vettori attraverso un ulteriore livello di indirizzamento. Nell’accesso, la dimensione `LZ` viene segmentata e la coordinata `iz` di ogni sito scomposta in `ic*VL+ik`.

La vettorizzazione avviene più internamente, nel ciclo che scorre il singolo cluster:

```
for(ic = HZoVL; ic < (HZoVL + SIZEoVL); ic++) // Each cluster in line
#pragma omp simd aligned(prv, nxt: DATA_ALIGNMENT)
for(ik = 0; ik < VL; ik++) // Each site in cluster
```

La dimensione del cluster è un parametro di interesse poiché influenza la geometria degli accessi, ed inoltre limita superiormente il fattore effettivo di vettorizzazione. Per valori inferiori ad 8, infatti, `icc` non sfrutta tutto il potenziale dell’instruction set di *Knights Landing*. Con  $VL \geq 8$  permettiamo la vettorizzazione ad 8 Double, che è una delle novità di AVX-512.

Per quanto spiegato in Sezione 1.2.2, allochiamo stencil e reticoli indicando l’allineamento (che terremo fisso a 4096 Byte). Ricorriamo ad altro strumento standard per assicurarci che in CSoA e CAoSoA i cluster siano allineati:

```
typedef struct __attribute__((aligned(CLUSTER_ALIGNMENT)))
{ data_t c[VL]; } vdata_t;
```

## 3.5 Streaming Stores

Le load/store non-temporali sono istruzioni che operano direttamente sulla memoria primaria, senza seguire le norme di coerenza cache: pertanto non danno

origine a latenze per trasferimenti di intere cache line. In particolare, le store non-temporali (o “streaming stores”) si rivelano utili nel caso si voglia salvare un dato al quale non si ha intenzione di riaccedere nel breve periodo. Possono addirsi al nostro caso, dove `propagate` e `collide` aggiornano il reticolo scrivendolo in un’area write-only: perciò studiamo ne studiamo gli effetti.

La non-temporalità non è contemplata in OpenMP, pertanto per implementarla ci rivolgiamo a direttive Intel. Con questa scelta rinunciamo un poco alla portabilità di codice, poiché, nonostante rimanga compilabile da compilatori diversi da *icc*, è probabile che questi ignorino l’*hint* della non-temporalità, per cui generino applicativi validi, ma non ottimizzati.

Per usufruire delle store non-temporali sostituiamo in `propagate` il pragma di vettorizzazione OpenMP con le due direttive:

```
#pragma simd
#pragma vector aligned nontemporal
```

Nell’instruction set corrente le non-temporali vettoriali operano solo con indirizzi allineati, quindi per evitare segmentation fault il compilatore genera streaming stores vettoriali (`vmovnt`) solo se è sufficientemente sicuro dell’allineamento degli operandi [8]. Nel caso di AoS, ad esempio, gli accessi in scrittura sono non allineati, pertanto per questo layout non è possibile testarne l’uso.

## 3.6 Prefetch Software

Lavorando con codici bandwidth-hungry l’utilizzo corretto della cache risulta di fondamentale importanza. Infatti, ogni miss ad un livello di cache comporta il trasferimento di una riga dal livello sovrastante nel migliore dei casi; nel peggio può portare a latenze più lunghe, dipendenti dai livelli di gerarchia cache e dalla politica di caching. Durante questi trasferimenti, gestiti principalmente dal memory controller, il processore è praticamente idle. Una tecnica usata per utilizzare meglio il processore ed ammortizzare i cache penalty è il prefetch [13], ovvero il caricamento preventivo di dati ad un livello di cache più vicino, quando ancora non sono necessari alla computazione ma il processore sarebbe altrimenti in stallo.

Il precaricamento motivato dal principio di località spaziale e temporale è un concetto più generale, applicabile in diverse forme (il caching stesso è una di queste). Nelle CPU moderne (tra cui le *Xeon Phi*) troviamo ad esempio unità dedicate

che predicono accessi a dati o istruzioni, e che fanno uso di banda altrimenti inutilizzata per caricamenti preventivi [14]. Anche i compilatori moderni sono in grado di riconoscere pattern negli accessi ai dati e decidere di inserire strategicamente istruzioni di prefetch nel testo di un'applicazione.

Come per il sistema di caching stesso, il rischio nell'impiego è che si facciano previsioni sbagliate, che *inquinino* la memoria e causino ulteriori penalty per trasferimenti non necessari. D'altro canto, quando gli accessi seguono uno schema preciso, come è nel caso di `propagate` e `collide`, è presumibile che esista una strategia vantaggiosa con cui pre-caricare.

## Impiego

La modalità di impiego a cui ci rifacciamo è il compiler-assisted prefetch: richiediamo esplicitamente al compilatore l'inserimento di istruzioni di prefetch tramite chiamate a primitive. Potenzialmente il prefetch software può essere applicato ad ogni livello di caching indipendentemente dall'architettura; l'instruction set *x86*, infatti, fornisce le istruzioni `prefetcht0`, `prefetcht1` e `prefetcht2` senza associarle ad determinati livelli di cache [15]. L'implementazione di queste tre istruzioni dipende dalla specifica gerarchia di cache dell'architettura, pur mantenendo l'ordinamento relativo delle operazioni: il prefetch *T0* porta i dati nella memoria più vicina al processore; *T1* a quella appena più lontana (se esiste), ecc.

Su Knights Landing (Struttura in Figura 1.6), `prefetcht0` porta i dati da L2/RAM a L1 e `prefetcht1/t2` portano i dati da RAM a L2, dove la RAM può essere la *MCDRAM* o la classica *DDR4* a seconda della configurazione [16].

Utilizziamo la funzione intrinseca `_mm_prefetch`, che dà la possibilità di indicare la locazione di memoria interessata ed il tipo di istruzione di prefetch preferita: `_mm_prefetch(MemAddress, PrefetchType);`, dove *PrefetchType* assume i valori `_MM_HINT_T0`, `_MM_HINT_T1` o `_MM_HINT_T2`.

Impieghiamo il prefetch solo nella `propagate` CAoSoA perché, piccola anticipazione, arrivati a questo punto questa struttura dati mostrerà un'inefficienza nell'utilizzo della banda di memoria. Come mostrato nel listato 3.7, nel corpo del ciclo più interno pre-carichiamo aree di memoria successive alla corrente, che con tutta probabilità conterranno popolazioni o siti visitati nelle iterazioni successive.

```

1  for(ip = 0; ip < NPOP; ip++)
2  {
3      int ix_s = ix - stencil->nx[ip];
4      int iy_s = iy - stencil->ny[ip];
5      int ic_s = ic - stencil->nz[ip];
6
7      // source offset
8      size_t idx_s = CIDX(ix_s, iy_s, ic_s);
9
10     // Prefetch di &(prv[idx_s].p[ip].c[ik]) + PREF_DIST ...
11
12     #pragma unroll
13     #pragma simd
14     #pragma vector aligned nontemporal
15     for(ik = 0; ik < VL; ik++)
16         nxt[idx_d].p[ip].c[ik] = prv[idx_s].p[ip].c[ik];
17 }
```

Figura 3.7: Compiler-assisted refetching impiegato in CAoSoA propagate.

Gli offset di memoria con i quali pre-carichiamo determinano il successo di questa euristica. In Sezione 4.4 confrontiamo diversi impieghi del prefetch T0 e T1, singolarmente oppure in maniera combinata, con due prefetch diversi a diverse distanze, ad esempio:

```

_mm_prefetch( (const char *) (&(prv[idx_s].p[ip])+PREF_DIST) , MM_HINT_T0);
_mm_prefetch( (const char *) (&(prv[idx_s].p[ip])+PREF_DIST_2) , MM_HINT_T1);
```

## 3.7 Roofline Analysis

Abbiamo già accennato al divario tra potenza di calcolo del processore e banda di trasferimento da/a memoria. In HPC, uno obiettivo dominante nella stesura di un codice ottimizzato sta nel renderlo meno memory-bound possibile, in modo che il throughput dipenda direttamente dalle caratteristiche del processore piuttosto che da quelle della memoria. Il riferimento per quantificare i limiti della computazione è il *modello roofline* [17].

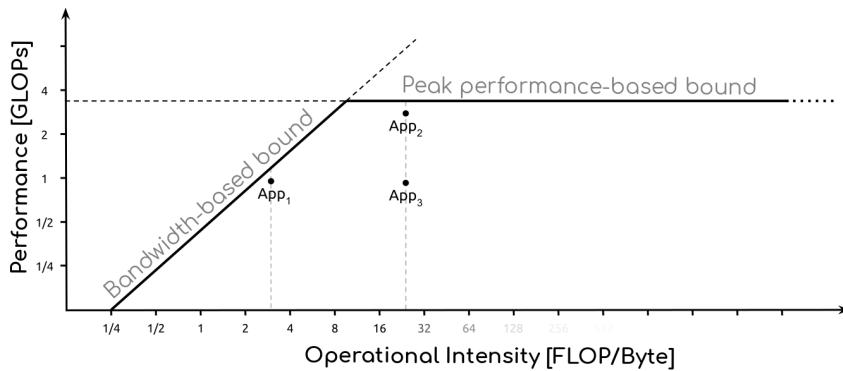


Figura 3.8: Esempio di Modello Roofline. I picchi di performance e banda di trasferimento di una architettura danno forma al tetto che limita le prestazioni effettive di un codice, a seconda della sua intensità computazionale. In questo esempio, APP<sub>1</sub> è bandwidth-hungry: ha un'intensità algebrica bassa, per cui il processore spende diversi cicli di clock aspettando che i dati siano trasferiti. Di contro, APP<sub>2</sub> e APP<sub>3</sub> rappresentano due programmi intensivamente operativi. In particolare, il primo dei due è più ottimizzato poiché raggiunge il picco di performance.

Un grafico roofline dà indicazioni di quanto un kernel o un applicativo sia cpu-bound piuttosto che memory-bound all'interno del contesto di una architettura (Figura 3.8 per un esempio). Due sono le metriche di interesse che riguardano il codice: le **prestazioni** e l'**intensità aritmetica**. Generalmente, data la complessità delle operazioni in virgola mobile, le prestazioni sono misurate nel numero di operazioni algebriche Floating Point eseguite al secondo (**FLOPs**, FLoating-Point OPerations per second). L'intensità aritmetica (*AI*) è invece il rapporto tra FLOP da eseguire e byte di dati da trasferire da/a memoria. Quando questo valore è basso, sono eseguite più operazioni di memoria piuttosto che calcoli, ed il processore spende tempo idle aspettando che i dati siano trasferiti dalla memoria. Al contrario, quando il valore è alto, viene speso più tempo nella computazione, e la banda di memoria costituisce meno probabilmente un impedimento.

In Tabella 3.9 riportiamo un calcolo di FLOP per sito e *AI* per modelli e benchmark in uso.

Utilizziamo il modello roofline per visualizzare le prestazioni assieme al contesto dei tetti computazionali di Knights Landing.

	Taylor Green	Flusso di Couette
<b>Modello LB</b>		FLOP/sito
<b>D3Q15</b>	553	1024
<b>D3Q19</b>	697	1296
<b>D3Q27</b>	985	1840
	Intensità computazionale (FLOP/byte)	
	2.29	4.26

Figura 3.9: Il modello LB determina il numero di popolazioni ma non influisce sull'AI della `collide`: i FLOP necessari per aggiornare un sito aumentano linearmente con  $NPOP$ , così come i byte da caricare e memorizzare. Aumentando le popolazioni, d'altro canto, la distanza media in memoria tra particelle dello stesso sito cresce e quindi la località del singolo sito tende a diminuire. Ciò che determina l'intensità computazionale di `collide` è il tipo di simulazione LB: la simulazione del flusso di Couette richiede circa il doppio dei calcoli per sito rispetto ai vortici di Taylor Green.

## 3.8 Analisi Consumi e Frequency Scaling

Come accennato, nel calcolo ad alte prestazioni ci si interessa anche dei costi energetici dei sistemi, aspetto critico per sistemi su larga scala. Nelle architetture recenti sono inclusi contatori hardware dedicati al tracking dei consumi energetici dei componenti, ed esistono interfacce che permettono di leggerli e calcolare in tempo reale il consumo della macchina. I processori moderni, inoltre, sono in grado di operare sulla propria frequenza di clock, aumentandola o riducendola a seconda del contesto (*Frequency Scaling*). Alzare il clock della cpu comporta minor tempo di soluzione, alzando il costo per unità di tempo; abbassarlo riduce i consumi, ma se la computazione richiede troppo tempo il costo totale di un'elaborazione può aumentare.

Per quanto riguarda i consumi dell'esecuzione di un codice, è interessante misurare l'andamento delle prestazioni in funzione di del clock. Un'analisi di questo tipo permette di trovare dei **trade-off tra performance e costi**, e suggerisce dove possano trovarsi i bottleneck di elaborazione.

## Impiego

Ci interessiamo all’analisi consumi delle due routine utilizzando un wrapper per l’interfaccia *PAPI* (*Performance API* [18]). Knights Landing offre contatori *RAPL* (*Running Average Power Limit*) per la valutazione dei consumi di package e DDR, e la possibilità di impostare quattro frequenze nel range tra 1GHz e 1.30GHz.

Il tool utilizzato [19] permette la consultazione a runtime dei contatori *RAPL* tramite un sistema di letture periodiche: si inizializza la raccolta dati dei consumi, che genera un processo che periodicamente legge i valori dei contatori e li etichetta con un timestamp; nei punti di interesse del codice si immettono dei marker, istruzioni che al passaggio dell’esecuzione segnano il timestamp; infine si chiama una funzione che scrive su file timestamp di marker e campionamenti.

È poi possibile incrociare i timestamp delle letture periodiche con quelli dei segnali posti per calcolare l’energia consumata nelle regioni delimitate. Ogni contatore riporta il dispendio energetico di un componente dall’ultima sua lettura, per cui i consumi sono calcolati accumulando le singole letture tra un marker e il successivo.

Come mostrato nel Listato 3.10, a inizio simulazione avviamo la raccolta indicando la frequenza di campionamento, e poniamo dei marker a delimitare l’inizio e la fine di *propagate* e *collide*. Per impostare il clock utilizziamo un semplice tool Linux, *cpupower* [20]: `cpupower frequency-set -f clock_freq`.

```

1 float acquisition_frequency = 10.0f;
2 PAPI_reader_Init("output.log", acquisition_frequency, "rapl");
3 PAPI_reader_Start();
4
5 for(iter = 1; iter <= NITER; iter++) {
6     ...
7     PAPI_reader_Marker();
8     propagate(f2, f1, stencil, HX, NX-HX);
9     PAPI_reader_Marker();
10    collide(f1, f2, stencil, HX, NX-HX);
11    PAPI_reader_Marker();
12 }
13
14 PAPI_reader_Stop();

```

Figura 3.10: Analisi consumi di *propagate* e *collide* con *PAPI Power Reader* [19]. Si inizializza il campionamento dei registri *RAPL* con una data frequenza. Dopodiché si impostano i marker, che segnano il tempo di computazione trascorso. Incrociando timestamp di campioni e marker ricaviamo i consumi dei due kernel.

## *Capitolo 4*

---

# *I Risultati*

---

Usiamo `propagate` e `collide` per misurare le prestazioni rispettivamente di memoria (*MCDRAM, DDR4*) e CPU.

`propagate` consiste in sole operazioni di load/store, quindi è fortemente memory-bound: la utilizziamo per valutare le prestazioni della memoria. Effettueremo il tuning utilizzando la MCDRAM, ed in Sezione 4.5 confronteremo le prestazioni con quelle ottenute usando la più lenta DDR. Misuriamo le sue prestazioni approssimando la banda raggiunta come segue:

$$\text{BW} = \frac{\#\text{load} + \#\text{stores}}{\text{elapsed}} = \frac{2 * \#\text{sites}}{\text{elapsed}}$$

La routine di `collide` esegue un certo numero di operazioni FP per ogni sito. Quando i calcoli necessari ad una routine sono tanti rispetto alle operazioni di memoria la banda di trasferimento non costituisce più un collo di bottiglia, e si considera invece la quantità di operazioni eseguite dal processore. In particolare, la metrica principale è il numero di *FLOP* eseguiti al secondo, di cui abbiam già parlato in Sezione 3.7, e che calcoliamo per `collide` nel seguente modo:

$$\text{FLOPs} = \frac{\#\text{sites} * \#\text{FLOP\_per\_site}}{\text{time per iteration}}$$

dove il numero di FLOP necessari per aggiornare un sito dipende dal benchmark e dallo stencil in uso (vedi Tabella 3.9).

Per i metodi reticolari, inoltre, un risultato di interesse è il numero di siti aggiornati al secondo, solitamente misurato in milioni: *MLUPs*, Million Lattice Updates per second. Lo calcoliamo come:

$$\text{MLUPS} = \frac{\#\text{sites} * \#\text{iter}}{\text{global\_elapsed} * 10^6}$$

Di seguito riportiamo i risultati delle ottimizzazioni usando i 4 layout di memoria (vedi Sezione 3.2).

## 4.1 Parallelizzazione

Come mostrato in Tabella 4.1, le prestazioni single-thread dei codici base sono peggiori rispetto a quelle ottenute su *Haswell*, un processore più tradizionale. Questo è un risultato atteso dal momento che la frequenza di clock del singolo core KNL è quasi tre volte inferiore a quella di un processore *Haswell*. Per sfruttare la superiore potenza di calcolo offerta dal KNL è necessario implementare parallelizzazione e la vettorizzazione, come descritto in sezioni 3.3 e 3.4. Nel resto della sezione valutiamo i benefici introdotti dalle varie ottimizzazioni, prendendo come riferimento simulazioni dei vortici di Taylor Green usando il modello *D3Q27* e un lattice cubico di lato 192.

Layout	propagate Bandwidth (GB/s)		collide GFLOPs	
	KNL	HSW	KNL	HSW
AoS	1.21	4.36	0.84	9.17
SoA	6.44	11.72	1.10	4.68
CSoA	7.48	11.70	2.59	5.34
CAoSoA	3.48	8.65	1.88	10.91

Tabella 4.1: Confronto prestazioni single-thread tra Knights Landing e Haswell.

### 4.1.1 Multi-threading e vettorizzazione

Occupando tutti i 64 core e vettorizzando osserviamo uno speed-up importante, che mostra la scalabilità del singolo core del Knights Landing (Fig. 4.1).

Impiegando solo multi-threading (Tabella 4.2), ad esempio, vediamo che Haswell con 8 core migliora le performance di un fattore  $\sim 6x$ , mentre in alcuni casi il KNL ottiene per `collide` uno speed-up quasi lineare con il numero dei core. `collide`, essendo maggiormente cpu-intensive, beneficia più facilmente di elaborazione parallela, ed infatti ottiene un ulteriore miglioramento di un fattore  $\sim 4\text{-}5x$  se si impiegano istruzioni vettoriali (Fig. 4.1).

Layout	propagate Bandwidth		collide GFLOPs	
	KNL	HSW	KNL	HSW
AoS	52.02x (0.81)	5.66x (0.71)	60.29x (0.94)	6.22x (0.78)
SoA	38.24x (0.60)	2.53x (0.32)	59.19x (0.92)	6.81x (0.85)
CSoA	35.38x (0.55)	2.54x (0.32)	50.25x (0.79)	6.67x (0.83)
CAoSoA	43.03x (0.67)	3.20x (0.40)	54.24x (0.85)	5.84x (0.73)

Tabella 4.2: Speed-up ed efficienza parallela dell’impiego di multi-threading su Knights Landing (64 core) e Haswell (8 core). Lo speed-up è misurato rispetto alle prestazioni single-thread di Tab. 4.1

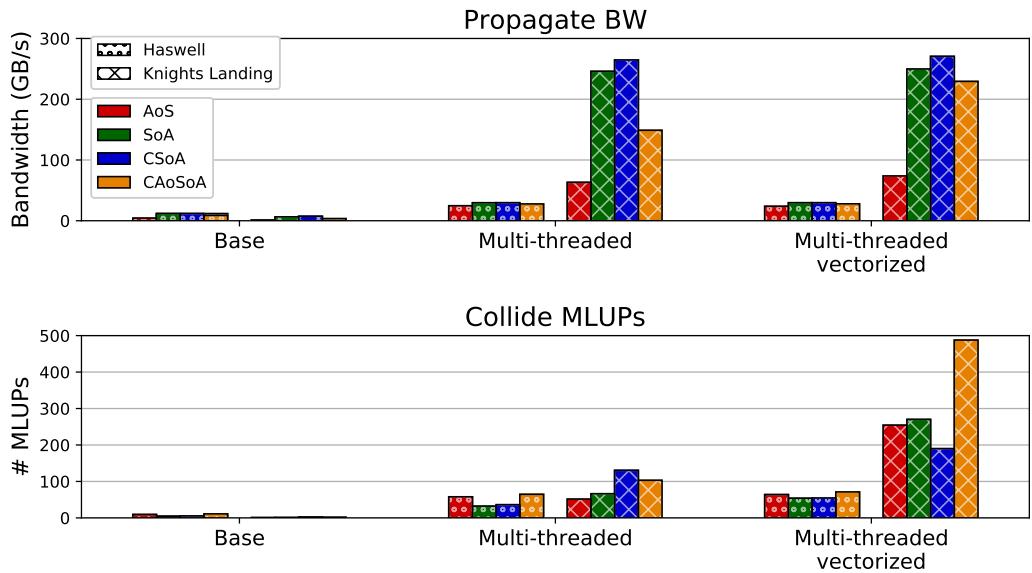


Figura 4.1: Effetto di multi-threading e vettorizzazione su KNL (64 core, 512-bit SIMD) e Haswell (8 core, 256-bit SIMD). Knights Landing esegue con più efficienza codice parallelo: questo si riflette nello speed-up multi-threaded per entrambi i kernel, e nel miglioramento ulteriore dato dalla vettorizzazione di cui beneficia specialmente `collide`, che è più compute-bound di `propagate`.

### 4.1.2 Hyper-Threading

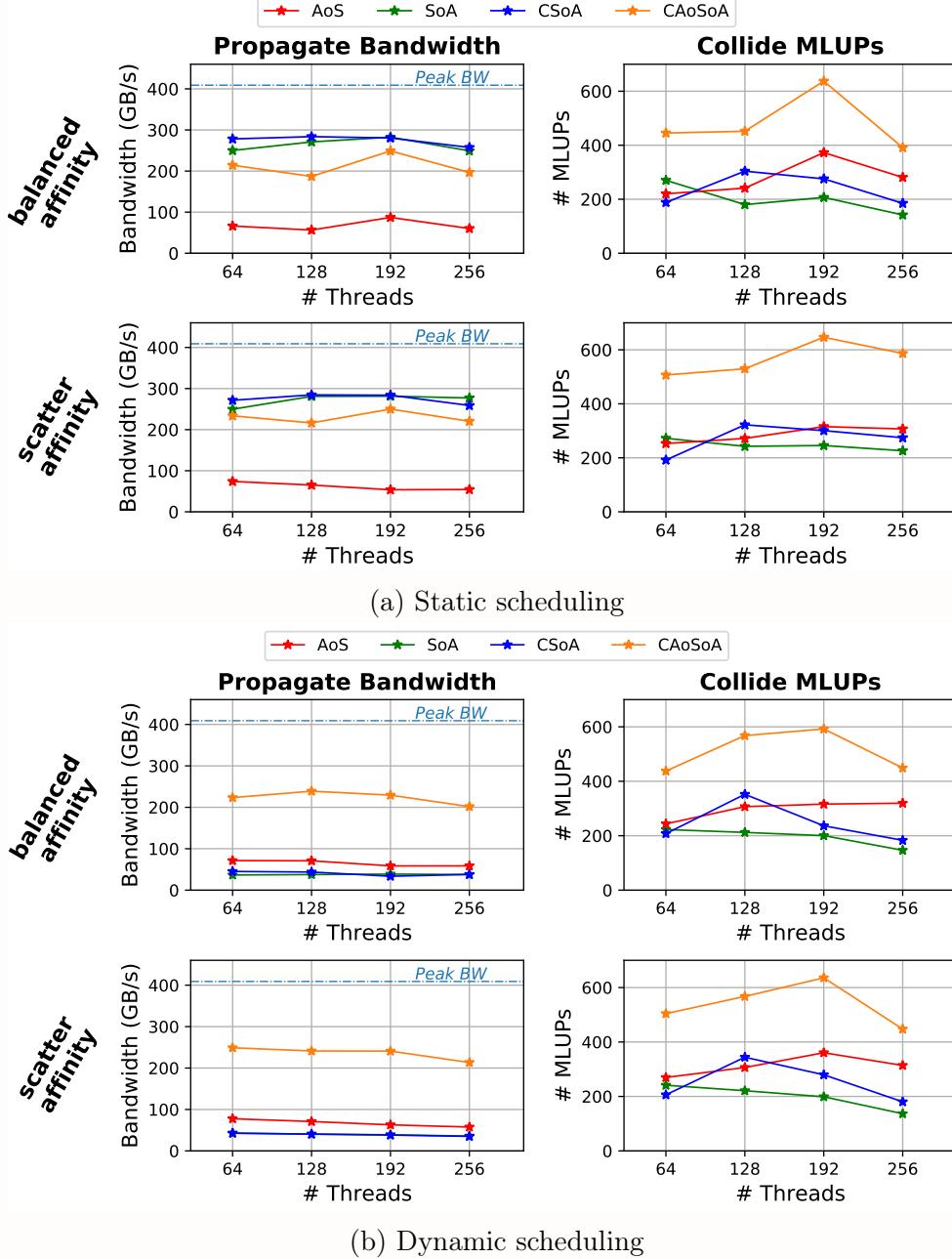


Figura 4.2: **Hyper-Threading + vettorizzazione**: Confrontando i risultati si osserva che in generale la politica di scheduling statico da risultati migliori. In questo caso, la scelta ottimale del numero di thread è 192, corrispondente al caso in cui ad ogni thread è assegnato un piano del lattice. Nel pannello (a) per i casi (C)SoA *propagate* raggiunge prestazioni pari al 75% del picco di banda. Per quanto riguarda *collide* la struttura CAoSoA offre un vantaggio tra il 20 e il 40% rispetto alle altre strutture dati.

Impiegando Hyper-Threading fino ad un fattore 4 e assegnando ad ogni core uno stesso numero di thread, osserviamo variazioni di performance più o meno accentuate, rappresentate in Figura 4.2. Essendo limitata dalla memoria, `propagate` non beneficia dell'*HT*, mentre `collide` mostra variazioni di performance più accentuate all'aumentare dei thread (indice che i core non sono trattenuti dalla banda).

Le prestazioni migliori di `propagate` si hanno per (C)SoA (località di popolazioni di siti vicini), con le quali si raggiunge un 75% del picco di banda.

In generale il numero di thread ottimo dipende dal numero di piani da spartire; lo scopo è quello di distribuire il lavoro in maniera uniforme, per cui numero di thread e la dimensione  $HX$  devono essere in qualche modo correlati (multipli, ad esempio). In figura troviamo dei picchi usando 192 thread per AoS e CAoSoA perché in queste implementazioni il ciclo più esterno conta esattamente 192 iterazioni, spartibili direttamente tra i thread. Di seguito, con prestazioni per lattice diversi troveremo altri valori preferenziali di *HT*.

Variando l'affinità sembrano esserci alcune differenze per `collide`, poiché varia l'associazione che porta i thread operanti su diverse aree di memoria a condividere le cache L2. Per `propagate` le differenze sono lievi: essendo limitati dalla memoria, le cache intra-tile non giocano un ruolo determinante. A causa dell'equa spartizione thread/core, assegnazione compatta e bilanciata si equivalgono.

Data l'omogeneità della divisione del lavoro, lo scheduling dinamico delle iterazioni non comporta particolari miglioramenti: al contrario, i due casi che sfruttano meglio la banda mostrano un calo drastico se si usa binding dinamico. Per quanto riguarda `collide`, CAoSoA costituisce un miglioramento del 20-40% rispetto agli altri tre layout. Questo ci dice che si tratta effettivamente di un buon compromesso tra località di popolazioni di siti vicini (caratteristica di CSoA e SoA) e di popolazioni dello stesso sito (caratteristica di AoS).

## 4.2 Streaming Stores

CSoA e SoA eseguono sequenzialmente gli accessi in lettura, e in maniera sparsa quelli in scrittura, per nel momento in cui la cache ignora questi ultimi cui osserviamo uno speed-up importante, che porta i due layout a raggiungere la banda massima ottenibile con MCDRAM. Lo speed-up per il layout ibrido si spiega allo stesso modo, ma non è altrettanto efficace: passiamo da un miglioramento del  $\sim 37\%$  di (C)SoA ad uno del  $\sim 13\text{-}22\%$  per CAoSoA, anche ordinando gli accessi in maniera diversa non riusciamo a sfruttare completamente la banda.

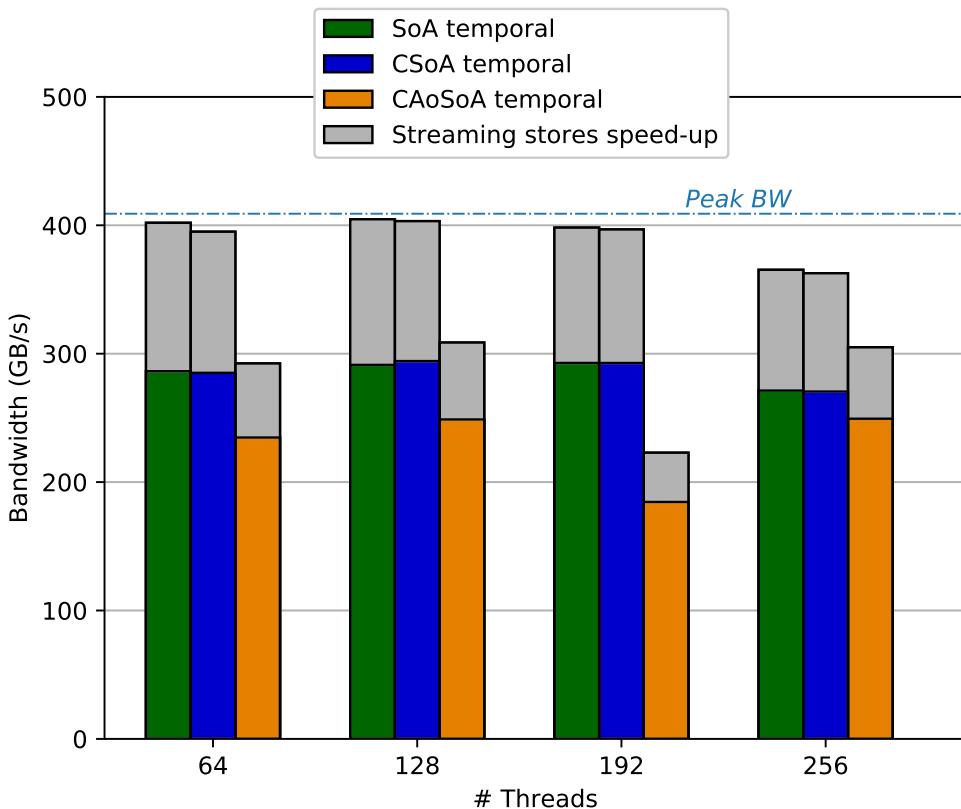


Figura 4.3: **Streaming Stores in propagate** (*D3Q27 Couette Flow, Lattice 256<sup>3</sup>*): considerando il tetto di banda ottenibile con MCDRAM quella misurata per uno *stream copy*, lo speed-up delle store non-temporali è ottimale per CSoA e SoA. CAoSoA mostra un miglioramento meno accentuato.

### 4.3 Vector Length

In Figura 4.4 riportiamo i risultati della variazione della dimensione del cluster per CSoA e CAoSoA.

Con  $VL = 4$  il compilatore vettorizza utilizzando istruzioni a 256 bit, mentre per  $VL \geq 8$  vengono utilizzate istruzioni a 512bit, sfruttando al massimo l'unità vettoriale del KNL. Come risultato sia `propagate` che `collide` possono osservare miglioramenti del 100% passando da vettori di 4 a vettori di 8 particelle.

Per  $VL$  superiori a 8 vengono ancora una volta utilizzate istruzioni vettoriali a 512bit, ma il processo di clusterizzazione produce differenti pattern di accesso alla memoria. Complessivamente `propagate` di CSoA non sembra avere valori preferenziali, mentre CAoSoA per diverse combinazioni di modello, benchmark e reticolo mostra una preferenza per cluster da 16-32 per la propagazione e da 8 per la collisione.

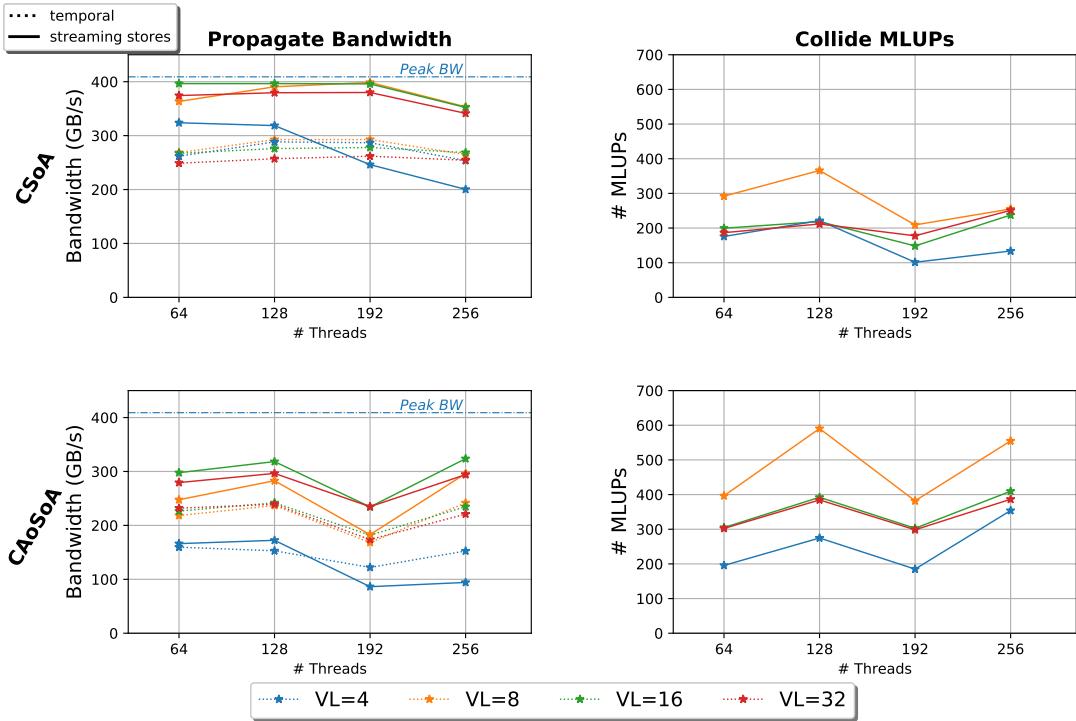


Figura 4.4: Prestazioni dei layout clusterizzati al variare della dimensione del cluster. Una dimensione minore di 8 porterà le *VPU* a non sfruttare a pieno i propri registri vettoriali, abbattendo le performance fino ad un fattore 2.

## 4.4 Prefetch Software

Come visto nella sezione precedente, la struttura dati CAoSoA permette di ottenere prestazioni migliori rispetto alle altre strutture dati per la routine di collide, ma presenta un penalty del 30% confrontato le prestazioni di propagate rispetto ai casi  $C(SoA)$ . Tuttavia, osservando i report di compilazione si nota che nel caso di CAoSoA il compilatore predice dei potenziali stalli della pipeline in corrispondenza delle operazioni di memoria:

```
vmovntpd %zmm0, 64(%rax) #38.11 c7 stall 2
```

Impieghiamo quindi il prefetch come descritto in Sezione 3.6; avendo a disposizione diversi tipologie (e combinazioni) di prefetch e non conoscendo a priori la distanza di prefetching ottimale, facciamo degli scan per valori da 0 a  $NPOP$ , utilizzando i modelli  $D3Q15$ ,  $D3Q19$  e  $D3Q27$ , e lattice  $256^3$  e  $512 \times 128 \times 128$ . Le prestazioni in questa sezione sono state misurate con  $VL = 8$ , scheduling iter/thread statico e affinità *scatter*.

Impiegando il precariamento per `propagate`, non solo riusciamo a guadagnare abbastanza banda da pareggiare le altre strutture dati, ma troviamo pattern molto interessanti per quanto riguarda la scelta delle distanze di prefetch. Indipendentemente dal modello LB, osserviamo picchi di banda associati a distanze di prefetch multiple di  $NPOP$ . L'andamento, visibile in Figura 4.5, è ondulatorio e mostra una qualche periodicità, per cui troviamo diversi picchi locali, tuttavia è evidente che si tratta di una tecnica da usare con attenzione, poiché è molto facile peggiorare le prestazioni. Il prefetch vincente per `propagate` è quello che porta i dati da memoria alla cache L2, condivisa dai 2 core dello stesso tile. Le prestazioni con più di 64 thread mostrano maggiori fluttuazioni ( 4.5d- 4.5b): il prefetch sembra dare risultati più instabili con Hyper-Threading.

In Figura 4.6 osserviamo il trend di `collide`: è meno prevedibile, ma delinea andamenti curvilinei che variano a seconda della geometria del reticolo, dello stencil, e del modello. Tendenzialmente, senza Hyper-Threading sembra esserci sempre spazio di miglioramento, poiché utilizzando prefetch  $T0$  o  $T1$  guadagnamo almeno un 15% di performance. Con Hyper-Threading, l'impiego del prefetch diminuisce di efficacia, infatti laddove guadagnamo performance saturando le risorse computazionali con più thread, il prefetch in `collide` diventa controproducente, poiché nel migliore dei casi lascia il throughput invariato. Questo si vede confrontando

le Fig. 4.6e- 4.6f e 4.6c- 4.6d , che inoltre mostrano come i guadagni impiegando HT oppure prefetch siano comparabili, tra il 10% ed il 20%. Concludiamo che, per la collisione, poiché vediamo con HT le stesse prestazioni guadagnate con il prefetch, conviene impiegare quest'ultimo solo se non troviamo un fattore di HT fortunato. Nella sezione successiva vedremo che, essendo limitati dalla banda della MCDRAM, le prestazioni raggiunte presentano pochissimo spazio di miglioramento.

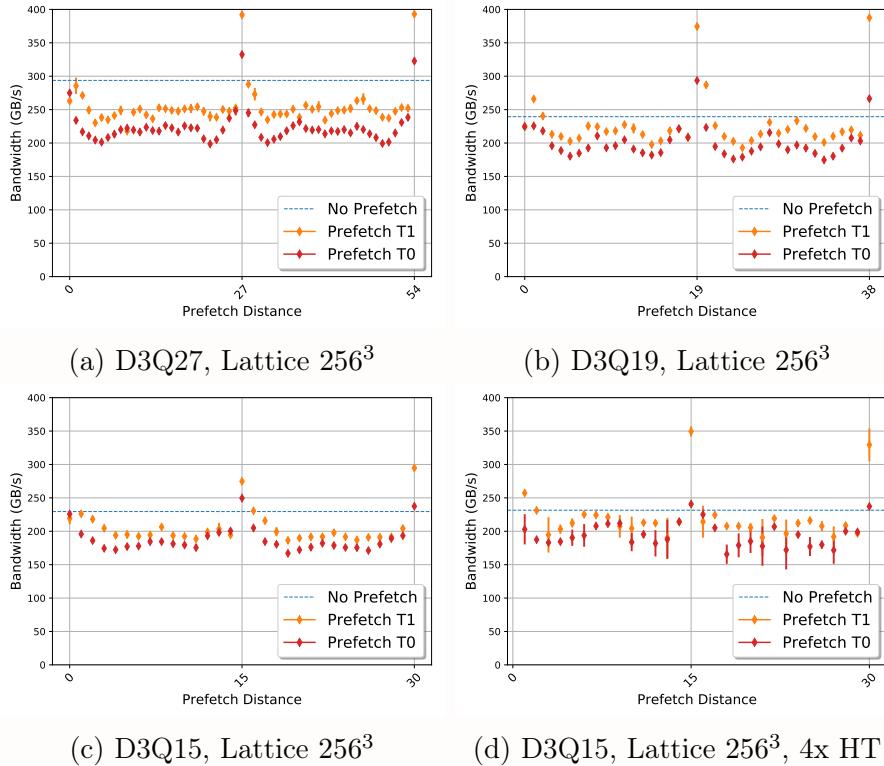


Figura 4.5: Banda di propagate con prefetch  $T0$  e  $T1$ , variando la prefetch distanze tra 0 e  $2 * NPOP$ . Per ogni combinazione sono stati considerati due campioni: un marker segna il loro valore medio, e una barra la loro dispersione. (a), (b), (c) mostrano un confronto senza Hyper-Threading tra diversi modelli, che rivela un andamento ondulatore e picchi per prefetch distance multiple di  $NPOP$ . Nel pannello (d) vediamo un caso con 4 thread per core che mostra una dispersione maggiore.

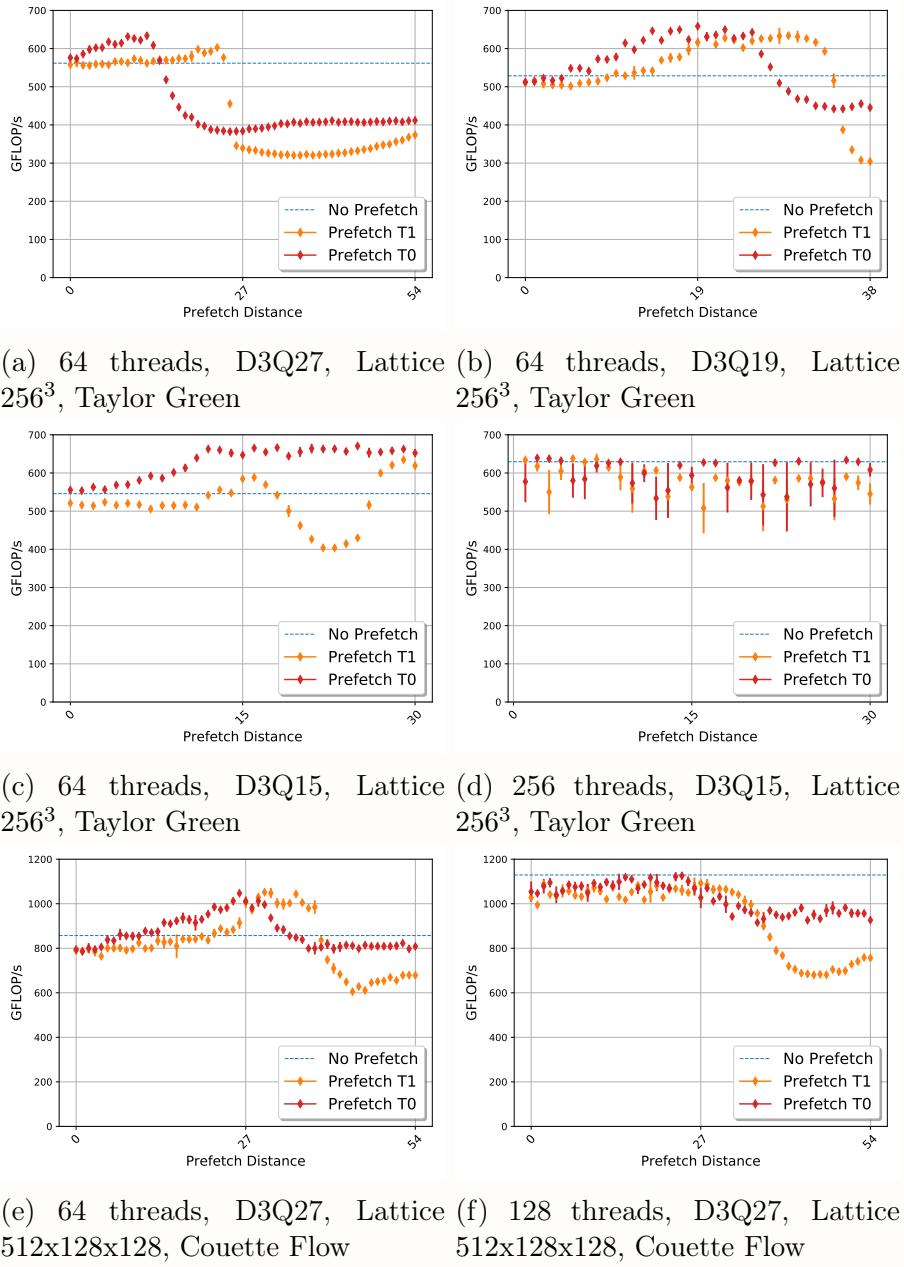


Figura 4.6: Effetto del prefetch su `collide`: senza Hyper-Threading si possono migliorare leggermente le performance. La distanza di prefetch va scelta accuratamente, e, da quello che abbiamo visto valori, vantaggiosi sembrano stare tra 10 e 30 byte. Se invece si trova un fattore di HT che dà buoni risultati, il prefetch al più lascia il throughput invariato.

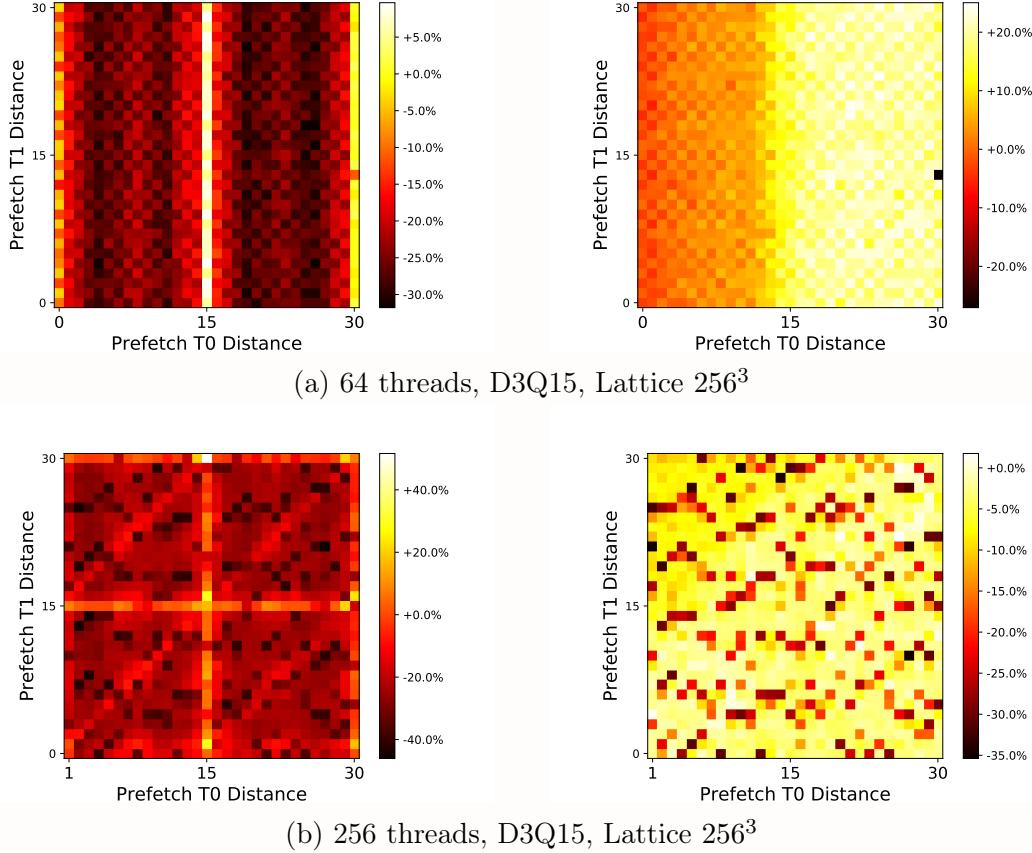


Figura 4.7: Prefetch combinato  $T0 + T1$ . A sinistra speed-up ottenuti su *propagate*, a destra su *collide*. I guadagni sono in linea con quelli ottenuti con prefetch singolo (Figure 4.5c, 4.5d, 4.6c, 4.6d).

Infine, testando l’uso di due prefetch a distanze diverse non rileviamo grossi guadagni: per entrambi i kernel le prestazioni riflettono quelle ottenute con prefetch singoli (Figura 4.7). A questo punto, per quanto riguarda le prestazioni, CAoSoA costituisce una buona di disposizione dei dati sia per *propagate* che *collide*.

## 4.5 Flat Mode vs. Cache Mode

Fino ad ora abbiamo effettuato il tuning utilizzando la MCDRAM, che offre una banda molto alta, ma è poco capiente. Figura 4.8 confronta le prestazioni in ter-

mini di MLUPs ottenute usando MCDRAM e DDR in *Flat Mode*, utilizzando un reticolo cubico di lato 192. Il fattore 4 di differenza tra la banda delle due memorie si riflette direttamente in `propagate`, poiché passiamo da un massimo di  $\sim 920$  MLUPs ad uno di  $\sim 170$ . Per quanto riguarda la `collide` vediamo che il peggioramento della banda tende ad annullare le differenze tra i layout. Inoltre, nella panoramica dei risultati raggiunti leggiamo un miglioramento progressivo passando attraverso i layout di memoria. Di seguito mostriamo le prestazioni ottenute con lattice più grandi di 16GB: questi necessitano dell'utilizzo della DDR, poiché può contenere fino a  $\sim 380$ GB. Figura 4.9 mostra in un grafico roofline (Sez. 3.7) le prestazioni computazionali di `collide`, confrontando i progressi raggiunti usando la MCDRAM con risultati ottenuti impostando il KNL in *Cache Mode* (Sez. 1.3.2). Innanzitutto osserviamo che per entrambi i benchmark il throughput è prossimo al picco teorico ottenibile sull'architettura; il massimo si ottiene con CAoSoA, eventualmente impiegando il prefetch. Conoscendo l'intensità aritmetica di `collide` (Tab. 3.9) stimiamo per il solo uso della MCDRAM una banda di memoria effettiva tra 120GB/s e 280GB/s. In modalità cache, lattice più piccoli di 16GB possono essere contenuti completamente nella MCDRAM senza “spillare” nella memoria principale. Come risultato, osserviamo che le performance rimangono invariate rispetto alla *flat mode*.

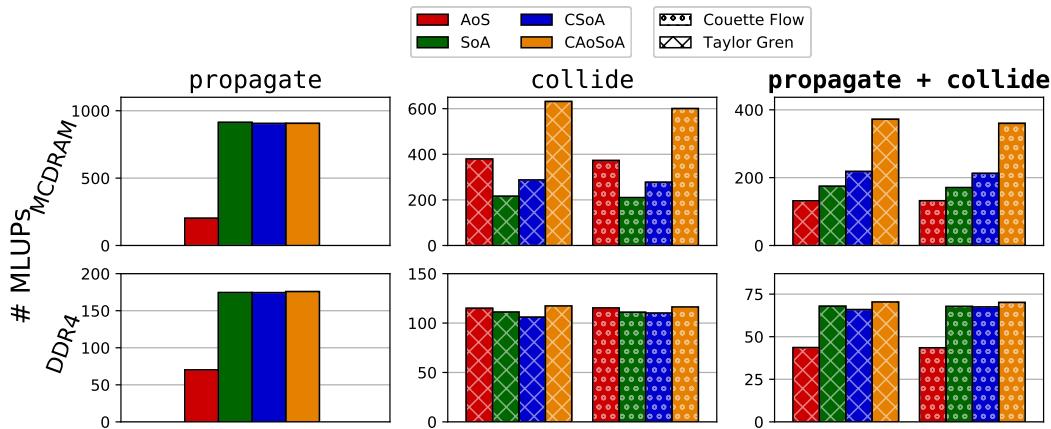


Figura 4.8: Prestazioni complessive per MCDRAM e DDR in modalità *flat* (modello D3Q27). Sono stati usati reticolni cubici di lato 192 (1.42GB)

D'altra parte, se il reticolo è più grande della dimensione della MCDRAM, la computazione si ritrova limitata dai trasferimenti DDR-MCDRAM: nel grafico

vediamo che le performance cadono al di sotto della banda offerta dalla DDR ( $\sim 90\text{GB/s}$ ), riducendo drasticamente l'efficienza di utilizzo del picco prestazionale del KNL. Per superare il problema si possono pensare strategie di loop-tiling (cache blocking, unroll-and-jam: [8], p. 521) che sfruttino meglio la MCDRAM in quanto cache; Oppure si possono minimizzare i tempi complessivi di esecuzione con una soluzione multi-nodo, dividendo il reticolo in porzioni non più grandi di 16GB e facendoli processare su diverse cpu Knights Landing.

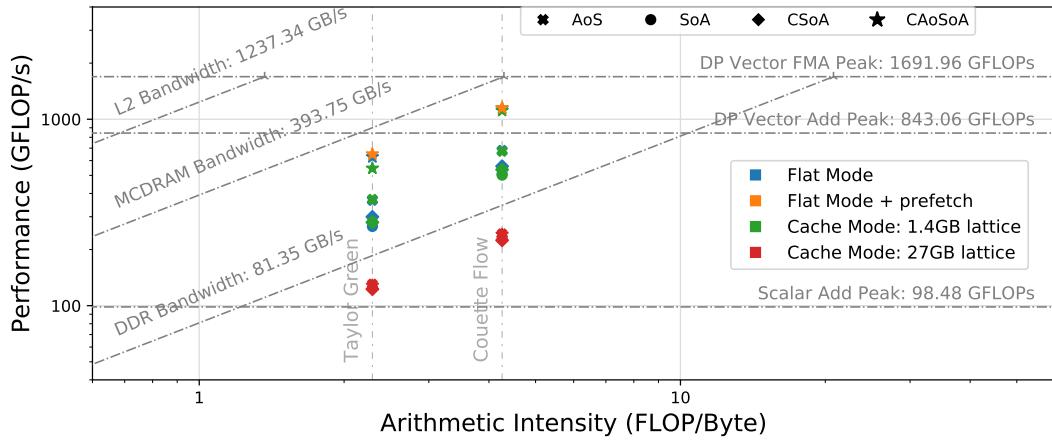


Figura 4.9: Roofline analysis della routine di `collide` (modello *D3Q27*). Con reticolii più piccoli di 16GB le prestazioni sono limitate dalla banda della MCDRAM sia in modalità *flat* che in modalità *cache*. Per reticolii più grandi, impostando il KNL in cache mode osserviamo un calo di performance, poiché la banda offerta della DDR è quattro volte minore. Sono stati usati reticolii cubici di lato 192 (1.4 GB) e 512 (27GB).

## 4.6 Frequency Scaling

Mentre le GPU NVIDIA offrono una ampia gamma di frequenze per il *frequency scaling* [21], che permette di trovare trade-off ragionevoli tra consumi e prestazioni, nel caso del KNL abbiamo solo 4 frequenze disponibili, per cui questo tipo di analisi è più limitata. In Figura 4.10 sono riportati tempi di soluzione e consumi energetici dei due kernel al variare della frequenza di clock, misurati nelle modalità descritte in Sez. 3.8. Anche per questo studio è stato usato un reticolo 192<sup>3</sup>, allocato su MCDRAM in flat mode.

AoS, dati i lunghi tempi di propagazione, mostra consumi quasi di un ordine di grandezza superiore rispetto agli altri layout. Inoltre, osserviamo un andamento importante al variare del clock: complessivamente passando da 1 a 1.3GHz risparmiamo tra il 10% ed il 20% del tempo per iterazione senza perdite in termini di costi. CAoSoA dà le prestazioni migliori sia per performance che per consumi: l'ottimo si ha con la frequenza massima di 1.3GHz. Rispetto al caso migliore (192 thread), diminuendo la frequenza a 1.2GHz, 1.1GHz e 1GHz si perdono rispettivamente 1%, 3% e 7% di prestazioni, risparmiando un ~4-7% di energia.

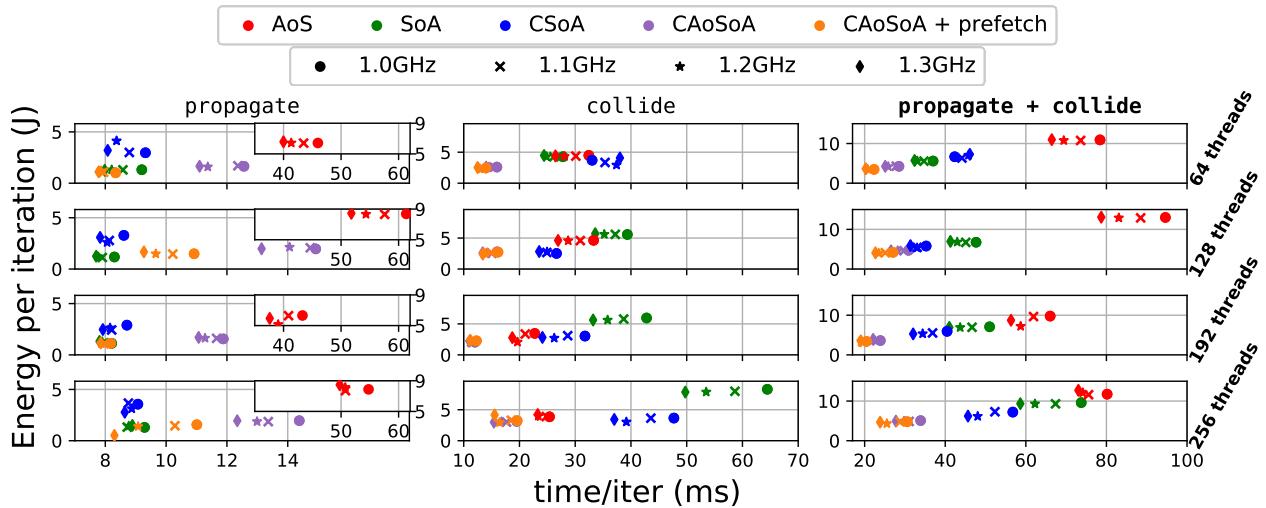


Figura 4.10: Consumi energetici e tempi di soluzione di `propagate` e `collide` al variare della frequenza di clock, per una simulazione *D3Q27 Taylor Green* con reticolo 192<sup>3</sup>. Vediamo la possibilità di guadagnare un 10-20% di performance senza penalty di costi alzando la frequenza del processore fino a 1.3GHz

## *Capitolo 5*

---

### *Le Conclusioni*

---

In questo lavoro di tesi abbiamo descritto e presentato i risultati dell’ottimizzazione di un codice tridimensionale per simulazioni reticolari di Boltzmann. In Tabella 5.1 riportiamo una panoramica dei risultati ottenuti applicando le diverse ottimizzazioni descritte nei capitoli precedenti. Di particolare importanza è stata l’ottimizzazione dei pattern di accesso alla memoria, che ha consentito di massimizzare l’utilizzo della banda del processore, grazie all’implementazione di diversi data-layout.

Partendo dal codice non ottimizzato, abbiamo inizialmente osservato uno speed-up quasi lineare occupando tutti 64 i core. Ottimizzazioni SIMD ci hanno poi permesso un ulteriore miglioramento di un fattore  $\sim 5$  per il kernel cpu-bound, consentendo di operare su più dati in parallelo. La gestione ottimizzata di più thread per core (Hyper-Threading) ha rappresentato un ulteriore strumento per affinare la parallelizzazione thread-level. L’impiego di streaming stores e di prefetching, infine, hanno ottimizzato ulteriormente l’utilizzo della banda di memoria.

Per la routine memory-intensive `propagate` abbiamo misurato una banda di memoria tra 350 e 410GB/s, valori molto vicini al picco di performance della MCDRAM; Per quanto riguarda la routine cpu-intensive `collide` abbiamo raggiunto

picchi di elaborazione tra  $\sim 650$  GFLOP/s e  $\sim 1150$  GFLOP/s, poco più del 30% del picco di 3 TFLOPs in doppia precisione offerto dal *Knights Landing*.

<b>Layout</b>	<b>propagate</b>		<b>collide (TaylorGreen)</b>		<b>collide (CouetteFlow)</b>	
	<i>time/iter</i>	<i>BW (GB/s)</i>	<i>time/iter</i>	<i>MLUPs</i>	<i>time/iter</i>	<i>MLUPs</i>
base						
AoS	2536.07	1.21	8236.78	0.86	0.93	1.21
SoA	474.05	6.44	6312.84	1.12	1.54	6.73
CSoA	413.02	7.48	2688.12	2.63	1.15	7.80
CAoSoA	879.37	3.48	3703.63	1.91	1.08	3.48
Multi-Threading						
AoS	48.39	63.19	137.15	51.61	127.09	55.69
SoA	12.43	245.89	106.85	66.24	84.69	83.57
CSoA	11.55	264.63	54.27	130.42	118.04	59.96
CAoSoA	20.54	148.84	68.67	103.07	123.21	57.45
+ SIMD						
AoS	26.11	77.24	39.59	271.09	39.95	258.95
SoA	25.76	256.16	11.94	274.79	11.95	273.61
CSoA	38.89	280.05	10.92	182.01	11.10	291.12
CAoSoA	13.81	237.00	12.90	512.54	13.20	456.74
+ Hyper-Threading						
AoS	34.34	89.05	18.61	380.25	19.10	370.65
SoA	10.75	284.36	33.45	211.57	25.87	273.61
CSoA	10.74	284.77	24.04	294.37	24.31	291.12
CAoSoA	12.28	248.90	11.20	632.10	11.48	616.74
+ Streaming Stores						
SoA	7.55	404.69	33.19	213.22	34.14	207.33
CSoA	7.58	403.24	24.37	290.46	23.89	296.24
CAoSoA	9.90	308.68	10.95	646.47	11.49	615.78
+ Prefetch						
CAoSoA	7.81	391.57	10.43	634.65	10.28	604.22

Tabella 5.1: Panoramica delle migliori performance ottenute con le diverse ottimizzazioni (modello *D3Q27*). Tutte le simulazioni hanno utilizzato 1000 iterazioni ed un lattice cubico di lato 192.

Attraverso una roofline analysis (Figura 4.9) abbiamo mostrato come questo divario sia da attribuire alla velocità di accesso della memoria MCDRAM. Per simulazioni su griglie di dimensioni superiori a 16GB la limitazione è data dalla banda della DDR4, con un conseguente degrado delle prestazioni di un fattore

quasi 5. Per supportare l'esecuzione efficiente di codici anche per reticoli di grandi dimensioni una possibile soluzione consiste nell'estensione dell'implementazione per supportare l'esecuzione su cluster multi-nodo.

Si rimanda ai posteri questo possibile sviluppo futuro del lavoro.

---

## *Bibliografia*

---

- [1] K. Rupp, “42 Years of Microprocessor Trend Data.” <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>, 2018. [Online; accessed 01-July-2018].
- [2] C. Carvalho, “The gap between processor and memory speeds.” [http://www.jedec.org/sites/default/files/Ricki\\_Dee\\_Williams-Final\\_0.pdf](http://www.jedec.org/sites/default/files/Ricki_Dee_Williams-Final_0.pdf), 2002. [Online; accessed 06-July-2018].
- [3] R. D. Williams, T. Sze, D. Huang, S. Pannala, and C. Fang., “Server memory road map.” [http://www.jedec.org/sites/default/files/Ricki\\_Dee\\_Williams-Final\\_0.pdf](http://www.jedec.org/sites/default/files/Ricki_Dee_Williams-Final_0.pdf), 2012. [Online; accessed 06-June-2018].
- [4] OpenMP, “OpenMP Application Program Interface - version 4.0.” <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>, 2013. [Online; accessed 06-June-2018].
- [5] R. K. (Intel), “Data Alignment to Assist Vectorization.” <https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization>, 2013. [Online; accessed 01-July-2018].
- [6] Wikipedia, “Larrabee — Wikipedia, the free encyclopedia.” [https://en.wikipedia.org/wiki/Larrabee\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Larrabee_(microarchitecture)). [Online; accessed 27-June-2018].

- [7] Top500, “Tianhe-2 (milkyway-2).” <https://www.top500.org/featured/systems/tianhe-2/>. [Online; accessed 02-July-2018].
- [8] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming. Knights Landing Edition.* Morgan Kaufmann, 2016.
- [9] Intel, “Xeon Phi Processor 7230 Specifications.” [https://ark.intel.com/products/94034/Intel-Xeon-Phi-Processor-7230-16GB-1\\_30-GHz-64-core](https://ark.intel.com/products/94034/Intel-Xeon-Phi-Processor-7230-16GB-1_30-GHz-64-core). [Online; accessed 01-July-2018].
- [10] B. B. (Intel), “Multi-Channel DRAM (MCDRAM) and High-Bandwidth Memory (HBM).” <https://software.intel.com/en-us/articles/multi-channel-dram-mcdram-and-high-bandwidth-memory-hbm>, 2016. [Online; accessed 01-July-2018].
- [11] NUMA, “Linux Support for NUMA Hardware.” <http://lse.sourceforge.net/numa/>. [Online; accessed 01-July-2018].
- [12] S. Succi, *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* Oxford: Clarendon Press, 2001.
- [13] “Cache prefetching — Wikipedia, the free encyclopedia.” [https://en.wikipedia.org/wiki/Cache\\_prefetching](https://en.wikipedia.org/wiki/Cache_prefetching). [Online; accessed 27-June-2018].
- [14] “Cpu hardware prefetch — The bios optimization guide.” <https://www.techarp.com/bios-guide/cpu-hardware-prefetch/>. [Online; accessed 27-June-2018].
- [15] F. Cloutier, “PREFETCHW — Prefetch Data into Caches in Anticipation of a Write.” <https://www.felixcloutier.com/x86/PREFETCHW.html>. [Online; accessed 08-July-2018].
- [16] insideHPC, “Prefetching Data for Intel Xeon Phi.” <https://insidehpc.com/2016/02/prefetching/>. [Online; accessed 27-June-2018].
- [17] “Roofline model — Wikipedia, the free encyclopedia.” [https://en.wikipedia.org/wiki/Roofline\\_model](https://en.wikipedia.org/wiki/Roofline_model). [Online; accessed 06-July-2018].

- [18] T. D., J. H., Y. H., and D. J, "*Collecting Performance Data with PAPI-C*", *Tools for High Performance Computing*. 2009.
- [19] E. Calore, "Papi power reader: Simple C library to read PAPI events related to power consumption." <https://baltig.infn.it/COKA/PAPI-power-reader>, 2018. [Online; accessed 06-June-2018].
- [20] Archlinux, "Cpu frequency scaling — Archwiki." [https://wiki.archlinux.org/index.php/CPU\\_frequency\\_scaling#cpupower](https://wiki.archlinux.org/index.php/CPU_frequency_scaling#cpupower). [Online; accessed 06-June-2018].
- [21] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications," *ArXiv e-prints*, Mar. 2017.