ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Campus di Cesena
Scuola di Scienze

Corso di Laurea in
Ingegneria e Scienze Informatiche

Implementazione ottimizzata dell'operatore di Dirac su GPGPU

Tesi in

High Performance Computing

RELATORE:
Prof. **Moreno Marzolla**

CANDIDATO:
Nicolas Farabegoli

CORRELATORE:
Dr. **Enrico Calore**

Anno Accademico 2019/2020

Alla mia famiglia

Indice

Introduzione

L'obiettivo di questa tesi consiste nello studio dell'architettura dei Tensor Core, ovvero moduli che consentono di eseguire moltiplicazioni di matrici in hardware e come possono essere utilizzati per aumentare le prestazioni del calcolo dell'operatore di Dirac. Questo operatore viene utilizzato nelle simulazioni lattice QCD e rappresenta, nel complesso, l'operazione che impiega il maggior tempo per la computazione; ottimizzarne i tempi di esecuzione si riflette in un incremento generale delle prestazioni dell'intero algoritmo lattice QCD.

L'operatore di Dirac svolge come operazione principale la moltiplicazione tra matrici 3×3 e vettori 3×1; questa operazione può essere ottimizzata mediante l'uso dei Tensor Core, ovvero moduli hardware presenti nelle più recenti GPU NVIDIA che effettuano direttamente in hardware l'operazione chiamata **MMA** (Matrix Multiply and Accumulate) che effettua una moltiplicazione di matrici e somma di una terza.

NVIDIA consente di sfruttare i Tensor Core mediante diverse librerie: cuBlas, cuDNN e WMMA API. Si è deciso di utilizzare le WMMA API in quanto garantiscono un controllo a più basso livello dei Tensor Core affinché se ne possa capire meglio il funzionamento. Durante il tirocinio mi sono concentrato sullo studio dell'architettura dei Tensor Core, analizzando i possibili scenari di utilizzo realizzando piccoli programmi di esempio che li sfruttano e misurandone le prestazioni. Sono poi passato a studiare come è possibile adattare il calcolo dell'operatore di Dirac su Tensor Core analizzando l'algoritmo e la struttura dati preesistente. Infine sono state testate alcune soluzioni per incrementare le prestazioni dell'operatore di Dirac con Tensor Core.

Questa tesi si suddivide in tre capitoli: nel primo si illustrano le motivazioni storico-tecnologiche che hanno portato a sviluppare architetture parallele per velocizzare sempre più algoritmi di simulazione e non solo, analizzando le prime architetture fino ad arrivare alle moderne GPU che consentono di sfruttare un parallelismo massivo. Nel secondo capitolo si studia il funzionamento dell'architettura CUDA concentrando particolare attenzione al funzionamento dei Tensor Core e come questi ultimi possono garantire un vantaggio in termini di prestazioni nella moltiplicazione tra matrici. Infine nel terzo capitolo si introduce l'operatore di Dirac mostrando la struttura dati con cui opera l'algoritmo e le soluzione sviluppate per aumentarne le prestazioni con i Tensor Core. Vengono analizzate nel dettaglio le tre soluzioni proposte, evidenziandone vantaggi e svantaggi di ognuna oltre a fornire nel dettaglio la misura delle prestazioni ottenute da ognuna di esse.

Capitolo 1

Calcolo parallelo e GPU

1.1 Importanza del calcolo parallelo

Dal 1986 al 2002, le prestazioni dei microprocessori aumentavano di circa il 50% ogni anno [1]. Dal 2002 l'incremento delle prestazioni di un microprocessore single-core ha subito un drastico calo: dal 50% al 20%. Quindi se con un aumento del 50% in 10 anni si incrementavano le prestazioni di 60 volte, attualmente incrementandole del 20%, in 10 anni abbiamo un aumento di solo 6 volte le prestazioni iniziali.

Questa differenza nell'aumento delle prestazioni è associata a un drastico cambiamento nella progettazione delle CPU: dal 2005 i maggiori produttori di microprocessori hanno deciso di virare in direzione del parallelismo, cercando quindi di inserire all'interno di un singolo chip più microprocessori piuttosto che incrementare le prestazioni di un singolo microprocessore monolitico.

Questa scelta ha non poche conseguenze per gli sviluppatori di software: aggiungendo semplicemente più core all'interno di un chip non si migliorano le prestazioni di programmi seriali. Il programmatore si deve quindi prendere carico di riscrivere il software (o parti di esso) affinché si sfrutti il parallelismo messo a disposizione dall'hardware: in quanto il software non è a conoscenza della presenza degli altri core. Quindi, lo stesso software eseguito su un processore multi-core ha le stesse prestazioni dello stesso software eseguito su un processore single-core.

1.1.1 I sistemi paralleli

L'incremento delle prestazioni su processori single-core ha portato ad aumentare sempre più la densità di transistor all'interno del chip: al decrescere della dimensione di un singolo transistor, la sua velocità aumenta e quindi l'intero circuito integrato aumenta di prestazioni. Oltre all'aumentare delle prestazioni, riducendo la dimensione dei transistor aumentano anche i consumi: gran parte di questi consumi vengono dissipati sotto forma di calore e il surriscaldamento del chip degrada velocemente le sue prestazioni. Nel primo decennio degli anni duemila i circuiti integrati hanno raggiunto i loro limiti fisici per dissipare il calore con raffreddamento ad aria [2]. Per questo motivo è impossibile continuare a cercare di aumentare la velocità di un circuito integrato, ciononostante l'incremento della densità dei transistor continuerà ancora per un po'.

Analizzando la storia dell'High Performance Computing in termini di evoluzioni tecnologiche si osserva un notevole incremento nello sviluppo di nuovi processori e l'impatto che hanno nella comunità scientifica. Vengono suddivisi in tre grandi epoche questi sviluppi tecnologici:

- **Epoca 1**: CRAY-1, il primo super computer che disponeva di un'architettura vettoriale e raggiungeva una potenza computazionale pari a 160 MegaFLOP.
- **Epoca 2**: la barriera dei MegaFLOP è stata infranta muovendosi da processori single-core verso processori multi-core. Il CRAY-2 disponeva di 4 core vettoriali che gli consentivano di raggiungere un potenza computazionale di 2 GigaFLOPs.
- **Epoca 3**: Per oltrepassare la barriera del GigaFLOP occorreva costruire nodi computazionali interconnessi tra loro mediante una rete di comunicazione ad alte prestazioni. CRAY-T3D è stato il primo a raggiungere una potenza computazionale di 1 TeraFLOP. La rete di interconnessione era un toroiode a tre dimensioni che consentiva di ottenere una banda di trasmissione fino a 300 MB/Sec.

Dopo queste epoche per almeno 20 anni, non si sono rilevate innovazioni rilevanti. Le innovazioni tecnologiche si sono focalizzate principalmente su tre aspetti:

- Passare da un set di istruzioni a 8-bit, poi a 16-bit, poi a 32-bit ed infine a 64-bit.
- ILP (Instruction Level Parallelism) ovvero istruzioni che possono essere eseguite in parallelo.
- Incrementare il numero di core.

In figura 1.1 si può osservare uno schema che riassume le principali evoluzioni architetturali dei sistemi High Performance. Inizialmente si sviluppavano sistemi single-core dove la memoria centrale comunicava con una sola CPU; si è passati poi ad avere sistemi multi-core dove esiste una sola memoria centrale che però viene contesa dalle varie CPU. Infine si è passati ad una architettura a memoria distribuita, dove ogni CPU ha una sua memoria dedicata ed esiste un bus di interconnessione tra le varie CPU che consente il trasferimento dei dati e la comunicazione tra le varie CPU.

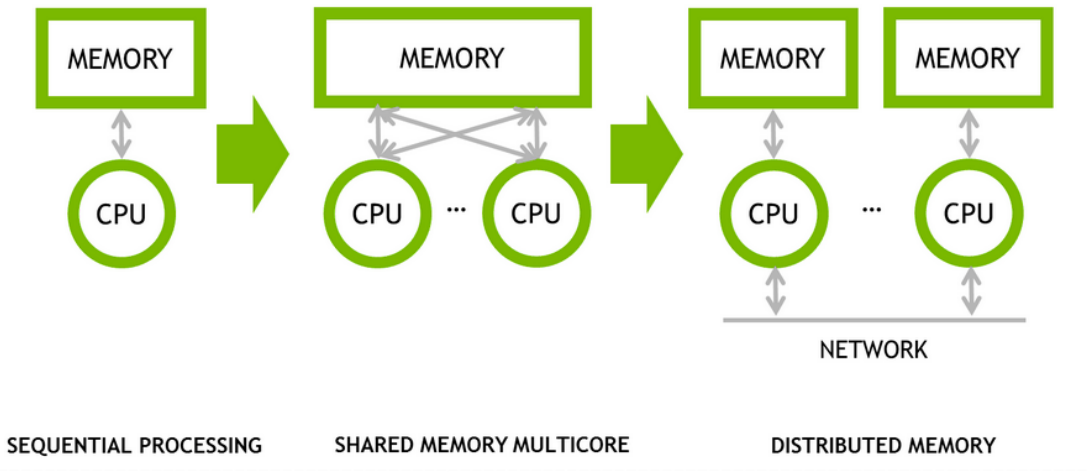


Figura 1.1: Evoluzione architetturale dei sistemi HPC

1.1.2 Il vantaggio dei programmi paralleli

Molti dei programmi scritti per sistemi a singolo processore non riescono a sfruttare appieno la presenza di più core. L'esecuzione di più istanze di un programma, ma non sarebbe l'obiettivo desiderato: se per esempio vengono eseguite più istanze di un videogioco, non si ottiene una maggiore fluidità oppure un maggiore dettaglio della grafica, si desidera invece che il videogioco sfrutti al massimo la presenza dei core del processore per avere una grafica più realistica, ad esempio. Talvolta per ottenere questo risultato occorre riscrivere il programma seriale affinché diventi un programma parallelo che sfrutti il parallelismo dell'architettura.

Non esistano strumenti o programmi che convertono programmi seriali in programmi paralleli, attualmente non esiste nulla di tutto ciò. Esistono diversi limiti che impediscono di effettuare questa conversione: dal momento che si possono scrivere programmi che riconoscono strutture comuni in programmi seriali e automaticamente traducono queste strutture in strutture parallele, la sequenza di costrutti paralleli può essere piuttosto inefficiente. Per questo motivo risulta molto difficile o talvolta impossibile "automatizzare" il processo di conversione di un programma da seriale a parallelo.

1.2 Il ruolo delle GPU

Ad oggi le GPU sono uno dei componenti cruciali in un computer. Inizialmente lo scopo di una scheda video (GPU) era quello di prendere un flusso di dati binari dalla CPU e renderizzarli su un display. Le moderne GPU sono in grado di effettuare calcoli complessi, come ricerche su enormi quantità di dati (big data research), machine learning ed Intelligenza artificiale. Le schede video si sono evolute nel tempo, passando da essere schede video single-core con una funzione ben specifica, ad essere schede programmabili con un numero molto elevato di core in grado di operare in parallelo.

1.2.1 Storia delle GPU

All'inizio del 1951, il MIT costruì *Whirlwind*, un simulatore per la marina militare statunitense. Viene considerato il primo sistema grafico 3D, ad oggi rappresenta la base di tutte le GPU. Nel 1976 RCA costruì il chip video *Pixie*, il quale era in grado di fornire in uscita un segnale video ad una risoluzione di 62x128 pixel. Hardware in grado di supportare la codifica RGB, sprite multicolor e tilemap backgroud risalgono al 1979.

Nel 1981 IBM iniziò ad usare un adattatore monocromatico e a colori (MDA/CDA) nei suoi computer. Non è ancora una GPU moderna ma è una componente progettata per un solo scopo: visualizzare video. All'inizio era in grado di gestire 80 colonne e 25 righe di caratteri o simboli. *ISBX* 275 creata da Intel, rappresenta la successiva rivoluzione: era in grado di gestire una profondità di colore pari a 8-bit con una risoluzione di 256x256 pixel, oppure 512x512 in gamma monocromatica.

Nel 1985 venne fondata la *ATI Technologies* questa azienda fu leader di mercato per anni con la sua linea di prodotti *Wonder*. Dal 1995 tutti i maggiori produttori di schede video introdussero un acceleratore 2D nelle proprie GPU. Il livello di integrazione delle schede video fu significativamente incrementato con la possibilità di sfruttare le Application Programming Interface (API) in modo da consentire agli sviluppatori di sfruttare a pieno le potenzialità dell'hardware.

Gli anni novanta hanno rappresentato l'era delle GPU: molte compagnie vengono fondate o acquistate. Tra le maggiori aziende leader del mercato in quegli anni troviamo NVIDIA la quale alla fine del 1997 deteneva circa il 25% del mercato mondiale per le schede video.

1.2.2 La rivoluzione del 3D

La storia delle moderne GPU inizia nel 1995 con l'introduzione del primi acceleratore 3D all'interno di una scheda video. In precedenza l'industria si era focalizzata sull'accelerazione 2D e le schede video avevano costi talvolta inaccessibili ai più.

La scheda video 3DFx's Voodoo, lanciata nel mercato a fine 1996, ebbe un successo tale che prese circa l'85% del mercato. Schede video che erano in grado di supportare solo l'accelerazione 2D divennero obsolete in breve tempo; *Voodoo1* non implementò nessun acceleratore 2D all'interno della scheda video, tanto è vero che era necessario affiancare una scheda video separata che si occupasse della parte 2D. Ciononostante fu un grande passo avanti per i videogiocatori. *Voodoo2* disponeva di tre chip a bordo e fu una delle prime GPU a supportare in parallelo il lavoro di due schede video in un singolo computer.

Con il progresso tecnologico nella costruzione di chip video, gli acceleratori 2D e 3D vennero integrati nello stesso chip. Nel 1999 apparse la prima vera GPU: NVIDIA lanciò sul mercato la *GeForce 256*. NVIDIA definì il termine **Graphics Processing Unit** come "*a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.* [3]"

1.2.3 General Purpose GPU (GPGPU)

L'era delle GPU general purpose inizia nel 2007 quando NVIDIA e ATI iniziano ad aggiungere alle proprie schede video molte più funzioni. Le due aziende però intraprendono due strade differenti per quanto concerne le GPGPU: nel 2007 NVIDIA rilascia l'ambiente di sviluppo *CUDA* [4], il primo (e maggiormente adottato) modello di programmazione per le computazioni con GPU. Due anni più tardi *OpenCL* [5] divenne largamente supportato nelle varie schede video. Questo framework consente di sviluppare codice sia per GPU che CPU con una particolare enfasi sulla portabilità. Così le GPU sono diventate dei dispositivi su cui si possono effettuare computazioni più generali.

Ad oggi le GPU non sono solo utili a fini grafici ma hanno trovato larga applicazione in diversi ambiti quali: machine learning, elaborazione di immagini scientifiche, algebra lineare, ricostruzioni 3D, ricerche mediche e molto altro. La tecnologia delle GPU tende ad aggiungere sempre più programmabilità e parallelismo ad una architettura che evolve sempre più verso a quella delle CPU.

1.2.4 L'utilizzo delle GPGPU

Sono molte le applicazioni, sopratutto in ambito scientifico, che richiedono di effettuare moltissime operazioni identiche su molti dati diversi. Per questo motivo le GPU possono venire in aiuto per migliorare le prestazioni: dal momento che le schede video sono state progettate per operare su migliaia di pixel di un immagine o video in parallelo, la loro architettura si è evoluta negli anni aggiungendo un numero sempre più elevato di core che potessero operare concorrentemente in modo da velocizzare le computazioni dei nuovi pixel dell'immagine. Oggi non è difficile trovare schede video che dispongono di centinaia se non migliaia di questi core; questi ultimi però differiscono non poco dai core presenti nelle CPU, infatti i core delle GPU sono meno complessi e anche meno prestazionali rispetto a quelli presenti in un processore. Il vantaggio nell'utilizzare i core della GPU sta nel loro parallelismo massivo. Infatti per molte applicazioni avere a disposizione un numero molto elevato di core con ridotte prestazioni, ma ottimizzati per lavorare in parallelo, si rivela più vincente che avere a disposizione un numero ridotto di core ad alte prestazioni, ottimizzati per eseguire sequenzialmente le operazioni. Questa affermazione non è sempre vera: tipicamente per problemi di ridotte dimensioni la CPU riesce a produrre risultati utili in un tempo inferiore, ma dal momento in cui si deve lavorare a un problema di grandi dimensioni, sfruttare il parallelismo massivo delle GPU può portare a un notevole vantaggio.

Con gli anni i produttori di schede video stanno sviluppando prodotti studiati appositamente per essere molto più efficienti nell'effettuare calcoli a scopo generale piuttosto che essere ottimizzate per il rendering, ray tracing ecc. Questi prodotti si collocano in

una fascia differente di mercato: quella riservata ai datacenter ad esempio. L'architettura di queste schede video si discosta da quelle tradizionali, in quanto si predilige la velocità in certi tipi di calcoli oltre ad avere una velocità di banda di memoria molto elevata, in quanto è richiesto trasferire moltissimi dati dalla GPU alla CPU (e viceversa) in tempi molto brevi.

In figura 1.2 possiamo osservare la differenza architetturale tra una CPU, che dispone di un numero limitato di core e una GPU che invece dispone di un numero molto più elevato di core.

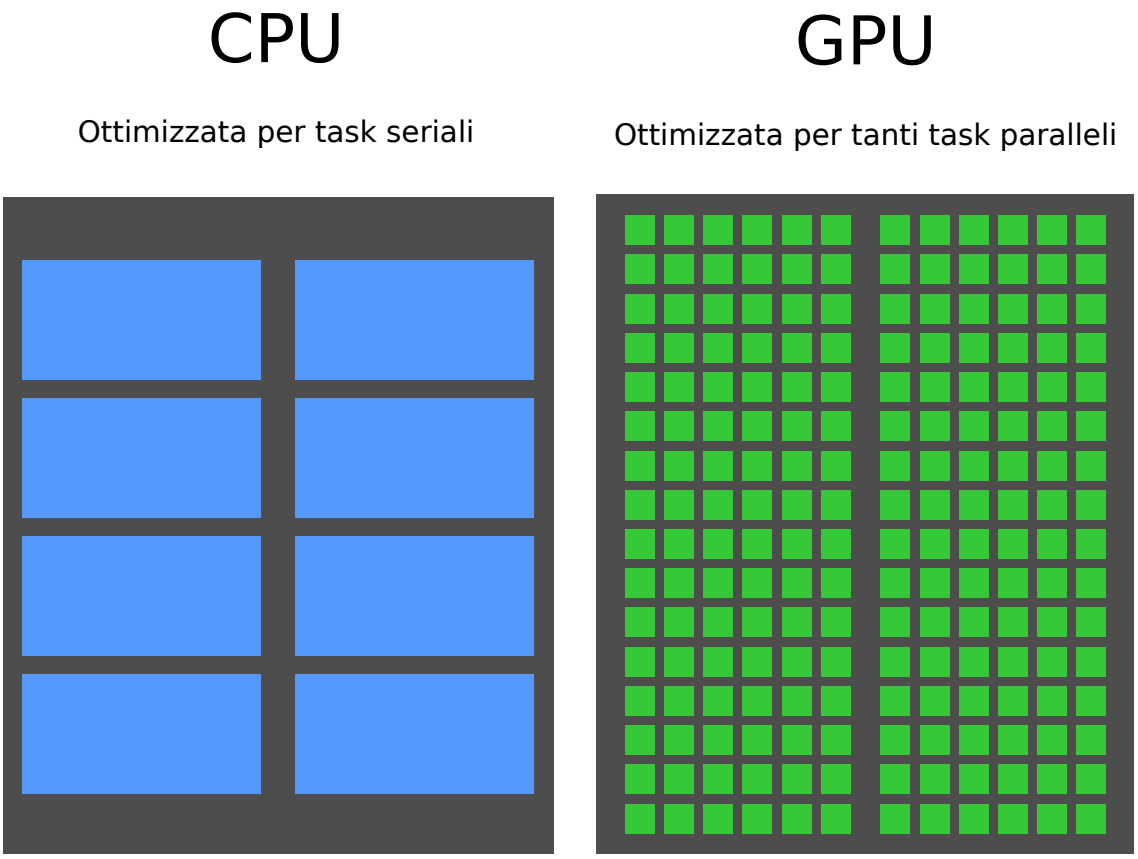


Figura 1.2: Confronto architettura CPU e GPU. I rettangoli di colore blu rappresentano i core di una CPU, i quadrati di colore giallo rappresentano i core di una GPU.

1.2.5 La fine della legge di Moore e l'inizio dell'HPC

La società moderna dipende sempre più fortemente dalla rapida crescita delle prestazioni dei calcolatori in grado di migliorare vari aspetti della mostra vita quotidiana e lavorativa. La crescita delle capacità computazionali è regolata principalmente dalla legge di Moore: si tratta di un modello tecnico-economico che ha predetto (sin dal 1968, anno in cui è stata pubblicata) che l'industria informatica avrebbe raddoppiato le prestazioni e le funzionalità di un prodotto (nel caso specifico si parla di microchip) ogni 18 mesi circa, tenendo fissi il costo, l'energia consumata e la superficie di ingombro [6]. Per quanto riguarda ingombro e consumi non è più così: con l'avvento dei dispositivi mobili come smartphone, laptop ecc. i consumi e le dimensioni hanno iniziato ad avere un ruolo estremamente rilevante, a tal punto che le aziende hanno deciso di investire molto su questi aspetti ottenendo risultati soddisfacenti. Possiamo quindi semplificare la legge di Moore dicendo che ogni 18 mesi la densità dei transistor all'interno di un chip raddoppiava (di conseguenza anche le prestazioni tendevano a raddoppiare) ad un costo che non variava. In figura 1.3 viene mostrato il livello di integrazione di transistor per ogni anno.

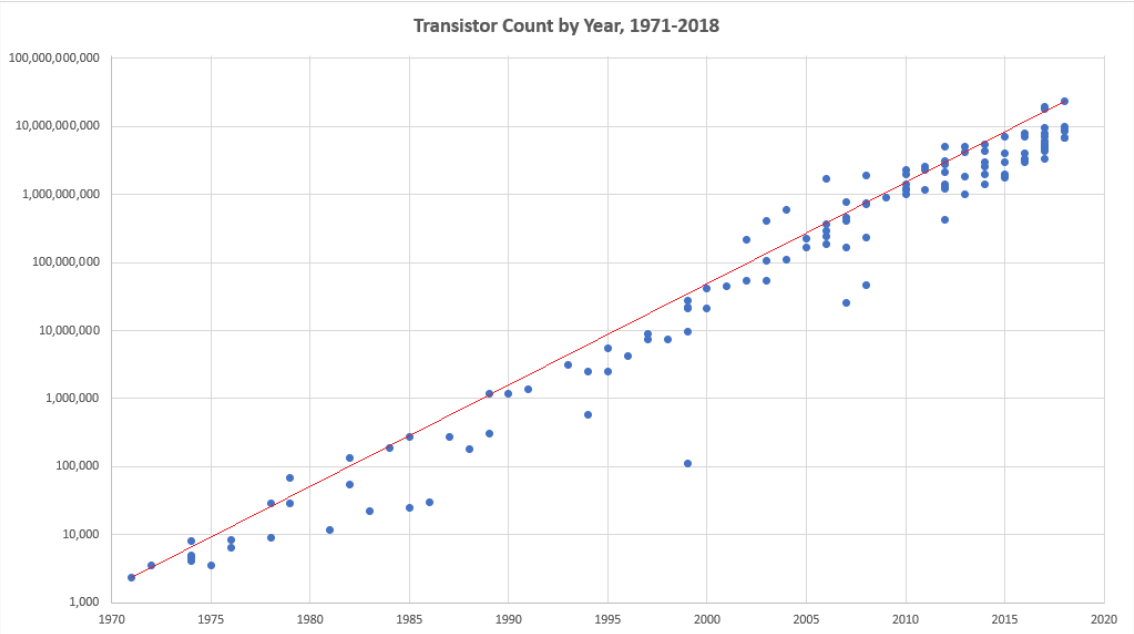


Figura 1.3: Andamento legge di Moore dal 1971 fino al 2018.
Fonte: https://medium.com/predict/moores-law-is-alive-and-well-eaa49a450188

Questa legge ha portato una sorta di stabilità nell'ecosistema tecnologico: compilatori, simulatori, emulatori ecc. vennero costruiti per processori ad uso generale come x86, ARM e Power PC. Ciononostante però, da circa un decennio il valore di questo modello inizia sempre più a perdere valore: l'aumento dell'integrazione dei transistor nei microchip, riducendone sempre più le dimensioni, ha portato a raggiungere la dimensione atomica per i transistor[7]. Questo significa che risulta molto difficile creare litografie in grado di produrre dispositivi con transistor di dimensioni pari a 3-5 nm, poiché questa dimensione corrisponde a qualche dozzina (o anche meno) di atomi di silicio rendendo molto difficile

controllare le cariche che fluiscono tra gli atomi, processo che sta alla base del funzionamento del transistor. Il classico processo tecnologico osservato fino ad oggi e che ha sempre seguito la legge di Moore per i 50 anni passati, sembra non rispecchiare più la realtà dei fatti a tal punto che nel 2025 si pensa che questo modello non sarà più valido.

Senza una nuova tecnologia di transistor in grado di soppiantare l'attuale **CMOS** (Complementary Metal-Oxide Semiconductor), l'opportunità principale per continuare ad incrementare le prestazioni per i dispositivi elettronici, sistemi HPC ecc. è quella di sfruttare in maniera più efficiente le architetture che abbiamo a disposizione. Per questo motivo si rende necessario ottimizzare i programmi affinché sfruttino al massimo le potenzialità offerte dall'architettura: si può pensare di programmare sistemi che meglio sfruttano il trasferimento dei dati in memoria, in quanto ad oggi la memoria centrale rappresenta un collo di bottiglia che penalizza fortemente le prestazioni delle applicazioni.

Alcune delle cause che portano a dover ottimizzare i programmi invece che aspettare architetture più prestazionali sono elencate di seguito:

- La fine della crescita delle frequenze di clock: questo limite (raggiunto ormai da parecchi anni) ha contribuito a sfruttare in maniera massiva il parallelismo offerto dalle architetture che lo supportano.
- La fine della riduzione della dimensione dei transistor mediante il processo fotolitografico: questo ha portato ad un grande aumento delle architetture eterogenee per poter incrementare le prestazioni (un esempio è l'uso di GPU per eseguire calcoli a scopo generale).
- Gestione aggressiva della potenza e contesa delle risorse con conseguenti tassi di esecuzione eterogenei: questo produce eterogeneità delle prestazioni che è in contrapposizione ai modelli di programmazione attuali).

In conclusione il progresso tecnologico nel campo della costruzione dei microchip mediante fotolitografia non è ai livelli di qualche decennio fa. Questo porta gli sviluppatori di software a dover sviluppare programmi che sfruttino al massimo le potenzialità delle architetture di cui dispongono per raggiungere prestazioni sempre più elevate; in tal senso stanno nascendo sempre più architetture eterogenee che consentono di migliorare le prestazioni delle applicazioni, ma il cui utilizzo richiede uno sforzo notevolmente più elevato rispetto a quello a cui si era abituati per architetture omogenee.

Fino a quando non verranno scoperte nuove tecniche per la costruzione di microchip in grado di superare le prestazioni dei chip attuali, l'unico modo di migliorare le prestazioni delle applicazioni è quello di ottimizzare al massimo le risorse a disposizione.

Capitolo 2

CUDA e Tensor Core

In questo capitolo introduciamo CUDA e l'architettura tipica di una GPU. Vedremo quindi il ruolo dei Tensor Core all'interno della scheda video e quali campi di utilizzo possono avere. Infine mostreremo qualche esempio di utilizzo dei Tensor Core e le prestazioni che riescono a raggiungere. Si precisa che in questo contesto i termini GPU, GPGPU e scheda video rappresentano la medesima cosa.

2.1 Architettura di una CUDA-capable GPU

CUDA è l'acronimo di *Compute Unified Device Architecture*, una piattaforma per il calcolo parallelo sviluppata da NVIDIA, la quale mette a disposizione del programmatore API che consentono la programmazione di GPU compatibili con CUDA per realizzare programmi ad uso generale e non specifici per la grafica. La piattaforma CUDA è uno strato che consente di accedere direttamente al set virtuale di istruzioni della GPU e agli elementi computazionali paralleli, per l'esecuzione di kernel.

L'architettura CUDA è stata progettata per lavorare con linguaggi come *C, C++ e Fortran* rendendo più semplice agli esperti in programmazione parallela l'uso delle risorse della GPU. L'uso di un dialetto dei linguaggi sopracitati consente di sviluppare codice più agevolmente, piuttosto che apprendere un nuovo linguaggio dedicato per la programmazione delle GPU. Le GPU NVIDIA predisposte per CUDA si possono programmare anche in altri modi: famosi framework come OpenACC [8] e OpenGL sono comunque supportati dalle GPU che prevedono CUDA. Questa compatibilità con altri framework non limita le GPU NVIDIA ad essere programmate solo con CUDA.

In figura 2.1 è schematizzato il collegamento tra CPU e GPU; tutte le comunicazioni avvengono mediante il bus PCI, il che consente di ottenere una discreta banda di trasmissione dei dati.
Un esempio di flusso di un programma CUDA è il seguente:

1. Si copiano i dati dalla memoria centrale (RAM) alla memoria della GPU.
2. La CPU si occupa di inizializzare i kernel che devono essere eseguiti sulla GPU.
3. I CUDA core eseguono in parallelo il kernel.
4. Viene copiato il risultato dalla memoria della GPU alla memoria centrale (RAM).

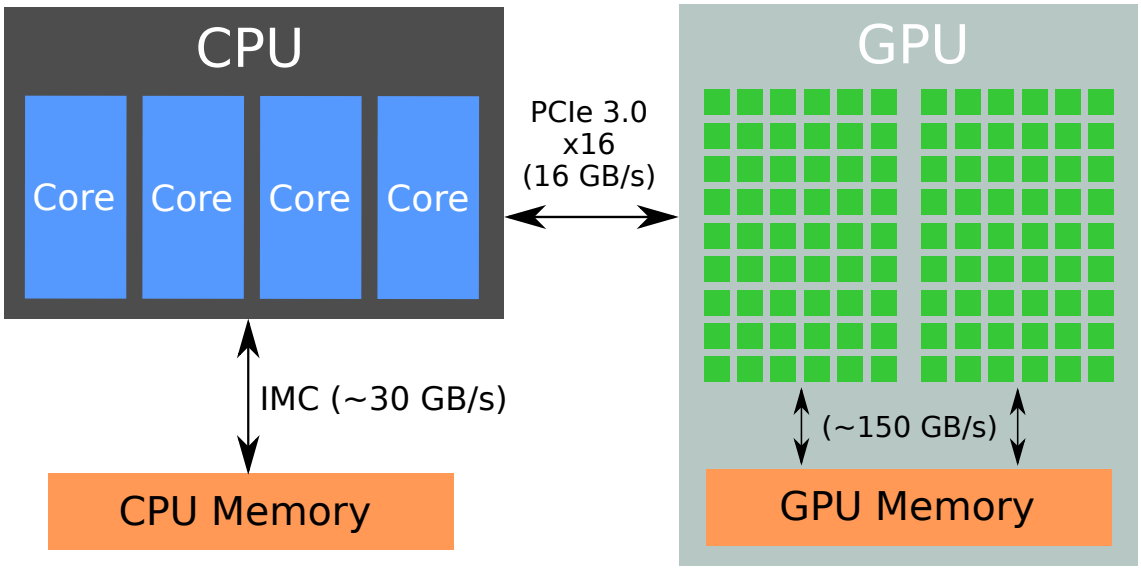


Figura 2.1: Flusso dati da CPU a GPU e viceversa

Architetturalmente una GPU CUDA è composta da uno o più *Streaming Multiprocessor* (SMs), ovvero un'astrazione dell'hardware: ogni SM è a sua volta formato da più *Streaming Processor* anche chiamati **CUDA Core**. Ogni CUDA Core ha accesso ad una cache chiamata anche memoria condivisa (shared memory), questa memoria è più veloce della memoria globale (global memory); la memoria condivisa impone alcune limitazioni: ad esempio può essere acceduta solo dai thread dello stesso *Streaming Multiprocessor* e quindi il suo contenuto non può essere visto da altri thread che non appartengono allo stesso SM.

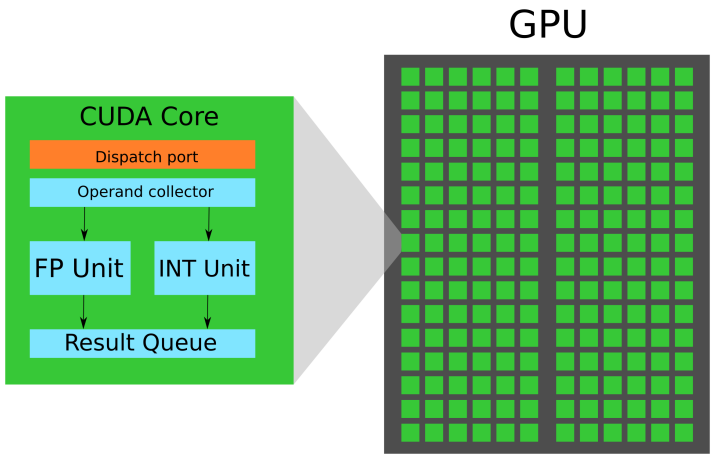


Figura 2.2: Architettura interna di un singolo CUDA core.

Il numero di SP/CUDA Core (in figura 2.2 viene mostrata l'architettura interna di un CUDA Core) dipende strettamente dal dispositivo utilizzato, è però importante ricordare che esistono modelli con qualche migliaia di core all'interno di una GPU. Ad esempio una scheda video NVIDIA Tesla V100 dispone di 5120 CUDA Core, una banda di memoria fino a 900 GB/s e una memoria globale di 16 GB. Ogni CUDA Core dispone di una ALU e una FPU, utilizza lo standard *IEEE 754-2008* per la rappresentazione dei numeri in virgola mobile (quindi pienamente compatibile con la rappresentazione adottata dalle CPU odierne), dispone di un modulo chiamato FMA (Fuse Multiply Add) in grado di operare su numeri in virgola mobile sia a singola che a doppia precisione. Il modulo FMA migliora l'operazione chiamata Multiply/Add (MAD) in quanto effettua la moltiplicazione e a seguito la somma con un solo arrotondamento finale, questo evita di perdere precisione nella somma; quindi il modulo FMA è più accurato ed è in grado di operare separatamente i calcoli. L'ALU nelle prime CUDA-capable GPU erano limitate a 24-bit per le moltipli-

cazioni ma, a partire dall'architettura *Fermi*, venne ridisegnata l'ALU e resa a 32-bit per tutte le istruzioni aderendo di conseguenza agli standard dei linguaggi di programmazione. L'ALU è stata poi ulteriormente migliorata prevedendo l'estensione a 64-bit per le operazioni aritmetiche.
Sono inoltre supportate tutte le operazioni booleane, shifting, confronti, conversioni di tipo, bit-field extract, bit-reverse insert e population count.

Ogni SM dispone di 16 load/store unit, consentendo di calcolare gli indirizzi sorgente/destinazione per sedici thread ad ogni ciclo di clock. Queste unità sono in grado di operare sia sui dati della DRAM che sulla cache. Ogni scheda video dispone anche di almeno quattro *Special Function Unit* (SFUs) le quali eseguono istruzioni trascendenti come seno, coseno, reciproco e radice quadrata. Ogni SFU esegue un istruzione per thread per ciclo di clock; la pipeline della SFU è disaccoppiata dalla dispatch unit consentendo a quest'ultima di servire altre unità mentre la SFU è occupata. La *dispatch unit* è quella unità responsabile di verificare che i thread siano mandati in esecuzione: prende l'istruzione da eseguire, il warp ID e la thread mask, quindi calcola l'ID del thread su cui mandare in esecuzione l'istruzione. Terminata l'esecuzione dell'istruzione, il dispacher invia ogni istruzione del thread ad una unità funzionale libera; nel caso in cui non ci siano unità funzionali libere, allora la dispatch unit mette in coda l'istruzione del thread per il ciclo successivo. Esistono almeno 4 unità funzionali: **Load/Store memory unit**, **Double precision floating-point unit**, **Special function unit** (utilizzata per approssimare le funzioni trascendenti) e unità **"cores"** che si occupa di gestire tutte le restanti operazioni.

Ogni SM organizza i thread in gruppi di 32 thread paralleli chiamati **warp**. Ogni SM dispone di almeno due scheduler per i warp e almeno due instruction dispatcher unit, consentendo quindi ad almeno due warp di essere preparati e mandati in esecuzione parallelamente. Dal momento che i warp eseguono le operazioni indipendentemente tra loro, lo scheduler non si deve preoccupare di controllare dipendenze all'interno del flusso di istruzioni. L'utilizzo di questa tecnica consente all'hardware di raggiungere prestazioni pressoché massime.

Una delle più grandi innovazioni architetturali è la memoria condivisa: si tratta di una memoria che solo i thread di uno stesso thread block possono vedere in modo da facilitare il riuso dei dati in quanto non è necessario ogni volta rileggerli dalla memoria centrale, pratica che creerebbe un elevato traffico nel bus della memoria. In termini pratici questa memoria (shared memory) è una sorta di cache abbastanza grande e quindi con velocità di lettura/scrittura più elevate rispetto alla memoria centrale; quello che solitamente si fa è: copiare una volta sola all'inizio del programma i dati che servono dalla memoria centrale alla memoria condivisa, quindi i thread eseguono i calcoli andando a leggere i dati dalla shared memory (anche più volte se necessario) memorizzano i dati sempre nella shared memory Infine i risultati vengono copiati in massa dalla memoria condivisa alla memoria centrale. La shared memory è molte volte la chiave per il successo in molte applicazioni HPC.

L'uso della shared memory può risultare vantaggiosa per molte applicazione, ma non per tutte. Alcuni algoritmi si adattano bene alla presenza della memoria condivisa, altri richiedono la cache, mentre alcuni necessitano una combinazione di entrambi. La struttura di memoria (schematizzata in figura 2.3) ottimale dovrebbe offrire i benefici sia della shared memory che della cache e consentire al programmatore di scegliere quale usare e come.

Con le nuove generazioni di GPU tutto ciò è possibile: l'architettura Fermi implementa un unico percorso per le load/store nella memoria con una cache L1 per SM e una cache L2 unificata per servire tutte le operazioni (load, store e texture). La cache L1 presente in ogni SM può essere configurata per supportare sia la shared memory che la cache. I 64 KB di memoria possono essere configurati per essere distribuiti in 48 KB di memoria condivisa e 16 KB per la cache L1; viceversa si può configurare affinché 16 KB siano destinati alla memoria condivisa e 48 KB alla cache L1.

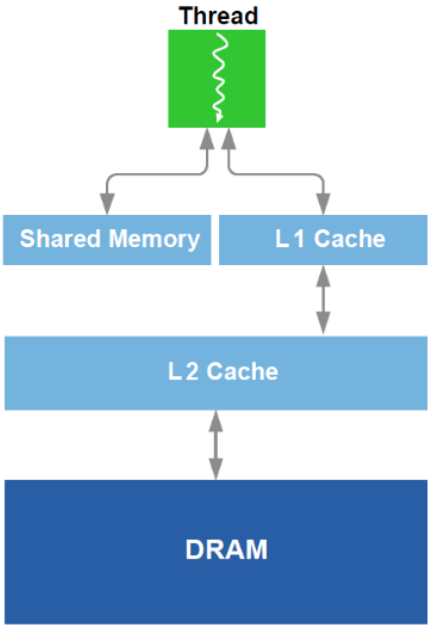


Figura 2.3: Gerarchia delle memorie all'interno di una GPU NVIDIA.

Come le CPU, le GPU supportano il multitasking attraverso l'uso del cambio di contesto, ovvero viene riservato alle applicazioni un intervallo di tempo detto *time slice* dove possono usufruire delle risorse. Con il tempo le pipeline delle GPU si sono evolute in modo da ridurre il tempo che occorre per ripristinare il contesto qualora si renda necessario cambiarlo (*cambio di contesto*); questo tempo è al di sotto dei 20 microsecondi, che però, comparato con i tempi impiegati dalla CPU (circa 1.5 microsecondi), è ben più lungo. Questo divario viene mitigato dal fatto che, come dicevamo in precedenza, coesistono più scheduler all'interno di uno SM ed essendo indipendenti tra loro aumentano il parallelismo migliorando le prestazioni.

A partire dall'architettura Fermi è possibile mandare in esecuzione kernel in parallelo, ovvero kernel diversi della stessa applicazione possono essere eseguiti parallelamente. L'esecuzione concorrente di più kernel consente al programma di sfruttare al massimo la potenzialità della GPU nel caso in cui siano implementati kernel di piccole dimensioni o che effettuano calcoli molto semplici. Se per esempio prendiamo un simulatore fisico che deve invocare un kernel per risolvere il comportamento di un fluido su un corpo rigido, se si eseguissero i kernel sequenzialmente si utilizzerebbe solo la metà dei thread disponibili nel processore; con questa tecnica quindi è possibile eseguire in parallelo kernel differenti

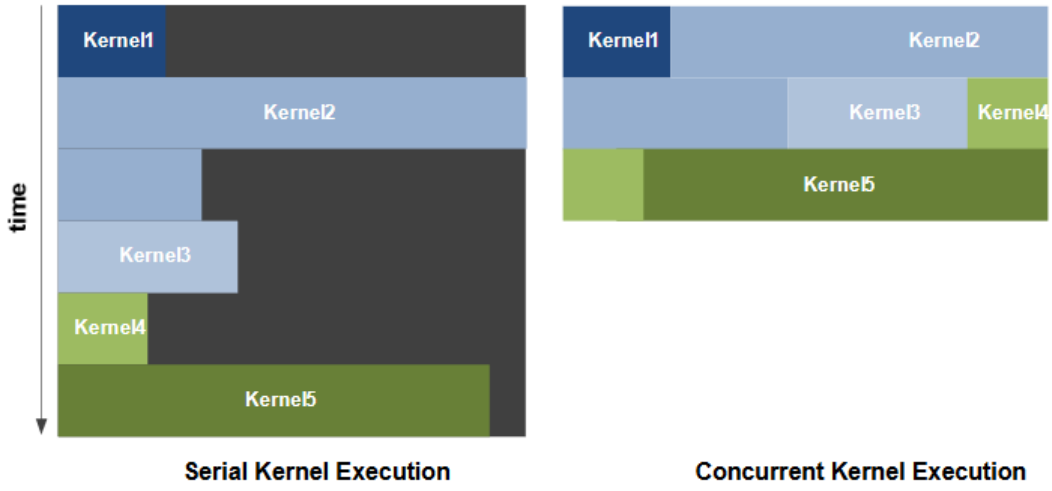


Figura 2.4: Confronto tempi di esecuzione di uno stesso programma in cui a destra viene mostrata l'esecuzione seriale dei kernel, mentre a sinistra l'esecuzione parallela.

per il calcolo del risultato finale. In figura 2.4 è mostrato lo schema che riassume questo concetto.

2.1.1 Griglie e blocchi in CUDA

Il framework CUDA prevede tre astrazioni fondamentali [9]:

- Una gerarchia di gruppi di thread.
- Memoria condivisa (Shared memory)
- Barriere per la sincronizzazione

Queste tre astrazioni sono disponibili al programmatore sotto forma di estensioni del linguaggio di programmazione.

CUDA prevede un parallelismo a grana fine a livello di dati e thread, il tutto incapsulato in un parallelismo a grana grossa per quanto riguarda il parallelismo dei task. Con questo metodo si spinge il programmatore a suddividere il problema in "grandi" sotto problemi che possono essere risolti indipendentemente in parallelo da blocchi di thread e ogni sotto problema può essere risolto in modo cooperativo in parallelo da tutti i thread dello stesso blocco.

Un kernel è eseguito in parallelo da un array di thread:

- Tutti i thread eseguono lo stesso identico codice.
- Ogni thread ha un identificativo che si usa per l'indirizzamento di memoria e prendere decisioni.

I thread sono organizzati come una griglia di blocchi (figura 2.5):

- Kernel diversi possono avere una configurazione diversa di griglie/blocchi.
- Thread dello stesso blocco hanno accesso alla stessa memoria condivisa e la loro esecuzione può essere sincronizzata.

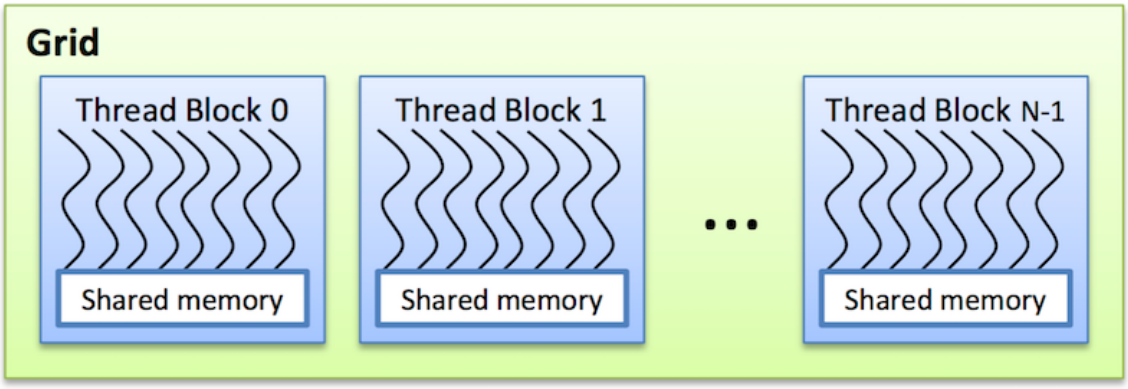


Figura 2.5: Organizzazione dei thread in blocchi e griglie.

2.1.2 Tensor Core

Come visto nelle sezione precedente, i CUDA Core effettuano una singola istruzione ben precisa per ciclo di clock della GPU: questo significa che la frequenza del clock gioca un ruolo fondamentale nelle prestazioni di CUDA (ovviamente anche il numero di CUDA Core influenza le prestazioni). I **Tensor Core**, dall'altro lato, possono effettuare una moltiplicazione di una matrice 4×4 in un singolo ciclo di clock con dati in virgola mobile a 16-bit di precisione (fp16) e sommare una terza matrice (sempre di dimensioni 4×4) con precisione in virgola mobile pari a 32-bit o 16-bit (vedi figura 2.6), in quanto l'accumulatore opera in *"mixed-precision"*, il che significa che può prendere in input un numero a 16-bit e restituire come risultato un numero a 32-bit. Mettendo quindi a confronto CUDA Core con Tensor Core possiamo affermare che i CUDA Core sono più lenti ma offrono una precisione maggiore; al contrario i Tensor Core sono sensibilmente più veloci ma hanno una precisione sensibilmente inferiore (con precisione in questo caso ci riferiamo all'ampiezza dei calcoli in virgola mobile che sono in grado di effettuare).

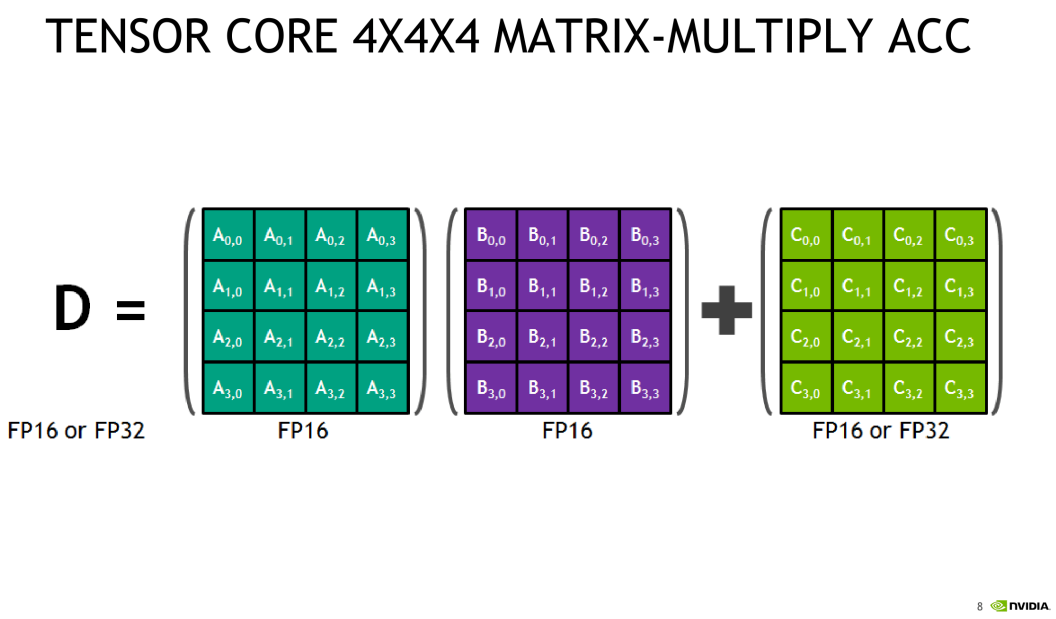


Figura 2.6: Funzionamento architettura Tensor Core.
Fonte: NVIDIA.

L'architettura *Turing* ha introdotto un miglioria ai Tensor Core: è prevista l'operabilità con dati di tipo INT4 e INT8 per operazioni di inferenza che possono tollerare errori di quantizzazione. L'introduzione di questa novità consente di accelerare ulteriormente le applicazioni basate su *Intelligenza Artificiale*. Quindi, anziché usare molti CUDA Core e molti cicli di clock, per ottenere lo stesso task con Tensor Core occorre solamente un ciclo di clock e questo causa un drammatico incremento delle prestazioni. NVIDIA stima che lo stesso calcolo effettuato con Tensor Core risulta essere più veloce di circa 12 volte rispetto ai CUDA Core. Le applicazioni di Machine Learning hanno tratto un enorme vantaggio da questa nuova architettura ma, se ci spostiamo nell'ambito dell'High Performance Computing, questa architettura può avere alcune limitazioni. NVIDIA rende accessibile al programmatore delle API per poter sfruttare i Tensor Core:

- **WMMA API**: API di basso livello che consentono di accedere ai Tensor Core.
- **cuBLAS**: API blas-compliant che sfruttano i Tensor Core per operazioni di algebra lineare.
- **cuDNN**: API che sfruttano i Tensor Core per calcoli come la convoluzione ecc. utili per reti neurali.

Per l'HPC le prime due rappresentano una soluzione pratica; la prima libreria è stata progettata appositamente per poter realizzare algoritmi che sfruttano la moltiplicazione di matrici come operazione base, mentre la seconda (cuBLAS) è stata progettata ad un livello di astrazione più alto e specifica per operazioni di algebra lineare come moltiplicazioni vettore/vettore, matrice/vettore, matrice/matrice ecc prestando particolare attenzione all'efficienza e alle prestazioni. Una comparativa sulle prestazioni della moltiplicazione

tra matrici con uso di Tensor Core e con diverse librerie la si può trovare nell'articolo "*NVIDIA Tensor Core Programmability, Performance & Precision*"[10].

2.1.3 Architettura dei Tensor Core

L'uso dei Tensor Core è determinante in tutte quelle applicazioni in cui si devono moltiplicare matrici di grandi dimensioni; se invece occorre moltiplicare matrici di piccole dimensioni allora questa architettura potrebbe avere prestazioni non ottimali [11].

Vediamo ora in pratica come operano i Tensor Core: come accennato prima, effettuano una sola operazione di moltiplicazione tra due matrici 4×4 e accumulo di una terza. In pratica però i Tensor Core lavorano su matrici 16×16 in cui le operazioni vengono gestite su due Tensor Core alla volta, questa particolarità sembra essere dovuta alla nuova architettura *Volta* e più specificatamente come questi Tensor Core siano collocati in uno SM (Streaming Multiprocessor). Per l'architettura *Volta* gli SM sono stati suddivisi in quattro blocchi di elaborazione o sub-core (vedi figura 2.7):

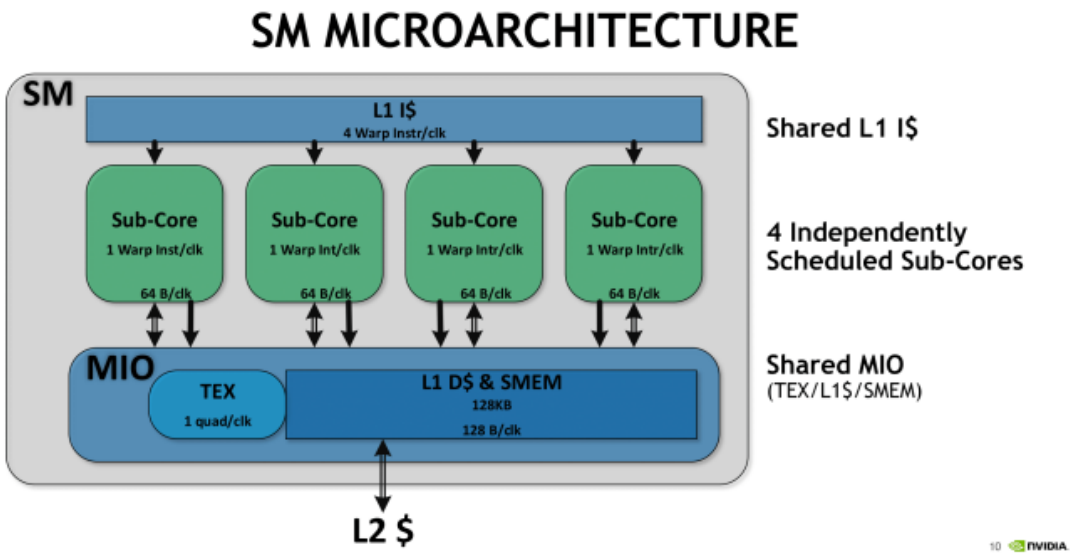


Figura 2.7: Architettura di uno Streaming Multiprocessor nell'architettura Volta.

per ogni sub-core lo scheduler invia un istruzione di un warp alla **Local Branch Unit** per ciclo di clock, all'array di **Tensor Core**, alla **Math Dispatcher Unit** o alla unità **MIO** condivisa. Se avessimo un solo Tensor Core per sub-core, ci precluderemo la possibilità di eseguire l'operazione su Tensor Core e simultaneamente altre operazioni matematiche. Utilizzando due Tensor Core il warp-scheduler invia l'operazione di moltiplicazione di matrici e, dopo aver ricevuto le matrici in input dal registro, effettua la moltiplicazione. Una volta terminata la moltiplicazione il Tensor Core scrive il risultato nel registro. In figura 2.8 viene mostrata l'architettura interna di un sub-core in cui si evidenziano i moduli che lo compongono e la loro interazione con gli altri.

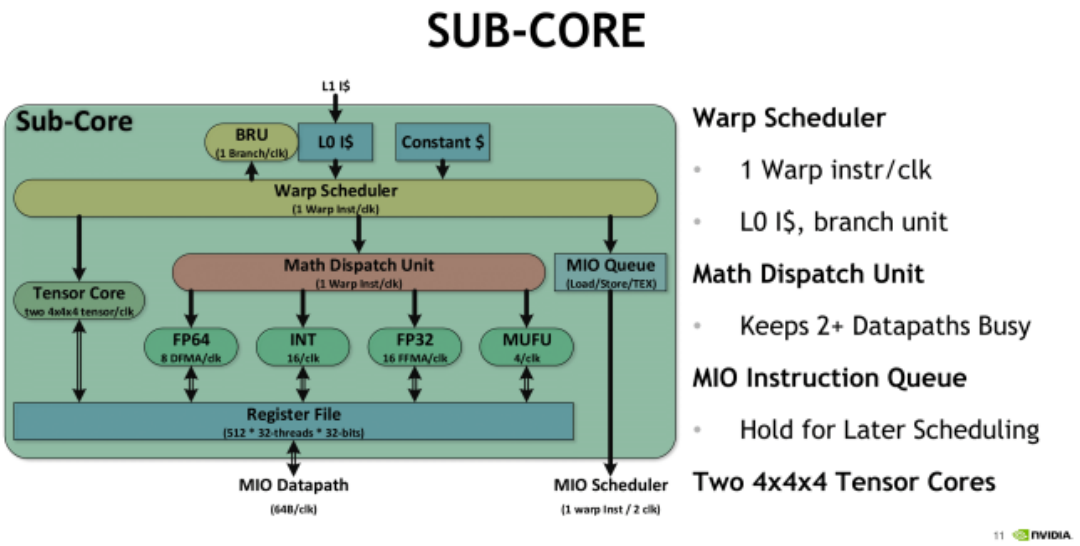


Figura 2.8: Struttura interna di un sub-core nell'architettura Volta in cui si mostrano le unità fondamentali che lo compongono.

Un analisi delle prestazioni a basso livello [12] effettuata da un team della Citadel LLC ha rivelato un numero di dettagli sull'architettura Volta, incluse le operazioni dei Tensor Core, i fragment, le locazioni dei registri e l'identità dei warp che sono coinvolti nelle matrici di input. Attualmente i programmatori CUDA possono usare delle primitive warp-level per effettuare la moltiplicazione del tipo 16x16x16 in half-precision con i Tensor Core. Andremo quindi ad analizzare in dettaglio il meccanismo di computazione che i Tensor Core adottano; in questi esempi consideriamo il caso in cui le matrici siano memorizzate in column-major order. Nell'esempio consideriamo le matrici di dimensione 16x16. Gli elementi delle matrici A e B sono float in half-precision (FP16), mentre in C abbiamo un float in single-precision (FP32).

Prima di invocare la moltiplicazione, il programmatore deve caricare i dati dalla memoria (DRAM) ai registri con la primitiva wmma::load_matrix_sync. Il compilatore NVCC traduce questa primitiva in una serie di istruzioni per il caricamento dei dati dalla memoria. A tempo di esecuzione ogni thread carica 16 elementi dalla matrice A e 16 elementi dalla matrice B.

Nell'articolo redatto dal team della Citadell LLC viene descritta la relazione tra le posizioni interne di ciascuna matrice; hanno inoltre scoperto che gli indici dei thread che caricano i corrispondenti dati nei registri sono fissi. In figura 2.9 e in figura 2.10 sono riportati gli indici dei thread che operano rispettivamente sulle matrici A e B. A tempo di esecuzione due thread differenti caricano i dati da una posizione in A e in B all'interno dei propri registri.

0 (0,8)	32 (1,9)	64 (2,10)	96 (3,11)	128 (0,8)	160 (1,9)	192 (2,10)	224 (3,11)	256 (0,8)	288 (1,9)	320 (2,10)	352 (3,11)	384 (0,8)	416 (1,9)	448 (2,10)	480 (3,11)
2 (0,8)	34 (1,9)	66 (2,10)	98 (3,11)	130 (0,8)	162 (1,9)	194 (2,10)	226 (3,11)	258 (0,8)	290 (1,9)	322 (2,10)	354 (3,11)	386 (0,8)	418 (1,9)	450 (2,10)	482 (3,11)
4 (0,8)	36 (1,9)	68 (2,10)	100 (3,11)	132 (0,8)	164 (1,9)	196 (2,10)	228 (3,11)	260 (0,8)	292 (1,9)	324 (2,10)	356 (3,11)	388 (0,8)	420 (1,9)	452 (2,10)	484 (3,11)
6 (0,8)	38 (1,9)	70 (2,10)	102 (3,11)	134 (0,8)	166 (1,9)	198 (2,10)	230 (3,11)	262 (0,8)	294 (1,9)	326 (2,10)	358 (3,11)	390 (0,8)	422 (1,9)	454 (2,10)	486 (3,11)
8 (16,24)	40 (17,25)	72 (18,26)	104 (19,27)	136 (16,24)	168 (17,25)	200 (18,26)	232 (19,27)	264 (16,24)	296 (17,25)	328 (18,26)	360 (19,27)	392 (16,24)	424 (17,25)	456 (18,26)	488 (19,27)
10 (16,24)	42 (17,25)	74 (18,26)	106 (19,27)	138 (16,24)	170 (17,25)	202 (18,26)	234 (19,27)	266 (16,24)	298 (17,25)	330 (18,26)	362 (19,27)	394 (16,24)	426 (17,25)	458 (18,26)	490 (19,27)
12 (16,24)	44 (17,25)	76 (18,26)	108 (19,27)	140 (16,24)	172 (17,25)	204 (18,26)	236 (19,27)	268 (16,24)	300 (17,25)	332 (18,26)	364 (19,27)	396 (16,24)	428 (17,25)	460 (18,26)	492 (19,27)
14 (16,24)	46 (17,25)	78 (18,26)	110 (19,27)	142 (16,24)	174 (17,25)	206 (18,26)	238 (19,27)	270 (16,24)	302 (17,25)	334 (18,26)	366 (19,27)	398 (16,24)	430 (17,25)	462 (18,26)	494 (19,27)
16 (4,12)	48 (5,13)	80 (6,14)	112 (7,15)	144 (4,12)	176 (5,13)	208 (6,14)	240 (7,15)	272 (4,12)	304 (5,13)	336 (6,14)	368 (7,15)	400 (4,12)	432 (5,13)	464 (6,14)	496 (7,15)
18 (4,12)	50 (5,13)	82 (6,14)	114 (7,15)	146 (4,12)	178 (5,13)	210 (6,14)	242 (7,15)	274 (4,12)	306 (5,13)	338 (6,14)	370 (7,15)	402 (4,12)	434 (5,13)	466 (6,14)	498 (7,15)
20 (4,12)	52 (5,13)	84 (6,14)	116 (7,15)	148 (4,12)	180 (5,13)	212 (6,14)	244 (7,15)	276 (4,12)	308 (5,13)	340 (6,14)	372 (7,15)	404 (4,12)	436 (5,13)	468 (6,14)	500 (7,15)
22 (4,12)	54 (5,13)	86 (6,14)	118 (7,15)	150 (4,12)	182 (5,13)	214 (6,14)	246 (7,15)	278 (4,12)	310 (5,13)	342 (6,14)	374 (7,15)	406 (4,12)	438 (5,13)	470 (6,14)	502 (7,15)
24 (20,28)	56 (21,29)	88 (22.30)	120 (23,31)	152 (20,28)	184 (21,29)	216 (22.30)	248 (23,31)	280 (20,28)	312 (21,29)	344 (22.30)	376 (23,31)	408 (20,28)	440 (21,29)	472 (22.30)	504 (23,31)
26 (20,28)	58 (21,29)	90 (22.30)	122 (23,31)	154 (20,28)	186 (21,29)	218 (22.30)	250 (23,31)	282 (20,28)	314 (21,29)	346 (22.30)	378 (23,31)	410 (20,28)	442 (21,29)	474 (22.30)	506 (23,31)
28 (20,28)	60 (21,29)	92 (22.30)	124 (23,31)	156 (20,28)	188 (21,29)	220 (22.30)	252 (23,31)	284 (20,28)	316 (21,29)	348 (22.30)	380 (23,31)	412 (20,28)	444 (21,29)	476 (22.30)	508 (23,31)
30 (20,28)	62 (21,29)	94 (22.30)	126 (23,31)	158 (20,28)	190 (21,29)	222 (22.30)	254 (23,31)	286 (20,28)	318 (21,29)	350 (22.30)	382 (23,31)	414 (20,28)	446 (21,29)	478 (22.30)	510 (23,31)

Figura 2.9: Relazione tra le posizione nella matrice A (column major) e gli indici dei thread nel caricamento dei dati. Ogni numero prima della parentesi è relativo all'indirizzo nella matrice A (espresso in byte), ogni numero dentro la parentesi è l'indice del thread che carica il dato dalla corrispondente posizione.

0 (0,8)	32 (1,9)	64 (2,10)	96 (3,11)	128 (0,8)	160 (1,9)	192 (2,10)	224 (3,11)	256 (0,8)	288 (1,9)	320 (2,10)	352 (3,11)	384 (0,8)	416 (1,9)	448 (2,10)	480 (3,11)
2 (0,8)	34 (1,9)	66 (2,10)	98 (3,11)	130 (0,8)	162 (1,9)	194 (2,10)	226 (3,11)	258 (0,8)	290 (1,9)	322 (2,10)	354 (3,11)	386 (0,8)	418 (1,9)	450 (2,10)	482 (3,11)
4 (0,8)	36 (1,9)	68 (2,10)	100 (3,11)	132 (0,8)	164 (1,9)	196 (2,10)	228 (3,11)	260 (0,8)	292 (1,9)	324 (2,10)	356 (3,11)	388 (0,8)	420 (1,9)	452 (2,10)	484 (3,11)
6 (0,8)	38 (1,9)	70 (2,10)	102 (3,11)	134 (0,8)	166 (1,9)	198 (2,10)	230 (3,11)	262 (0,8)	294 (1,9)	326 (2,10)	358 (3,11)	390 (0,8)	422 (1,9)	454 (2,10)	486 (3,11)
8 (16,24)	40 (17,25)	72 (18,26)	104 (19,27)	136 (16,24)	168 (17,25)	200 (18,26)	232 (19,27)	264 (16,24)	296 (17,25)	328 (18,26)	360 (19,27)	392 (16,24)	424 (17,25)	456 (18,26)	488 (19,27)
10 (16,24)	42 (17,25)	74 (18,26)	106 (19,27)	138 (16,24)	170 (17,25)	202 (18,26)	234 (19,27)	266 (16,24)	298 (17,25)	330 (18,26)	362 (19,27)	394 (16,24)	426 (17,25)	458 (18,26)	490 (19,27)
12 (16,24)	44 (17,25)	76 (18,26)	108 (19,27)	140 (16,24)	172 (17,25)	204 (18,26)	236 (19,27)	268 (16,24)	300 (17,25)	332 (18,26)	364 (19,27)	396 (16,24)	428 (17,25)	460 (18,26)	492 (19,27)
14 (16,24)	46 (17,25)	78 (18,26)	110 (19,27)	142 (16,24)	174 (17,25)	206 (18,26)	238 (19,27)	270 (16,24)	302 (17,25)	334 (18,26)	366 (19,27)	398 (16,24)	430 (17,25)	462 (18,26)	494 (19,27)
16 (4,12)	48 (5,13)	80 (6,14)	112 (7,15)	144 (4,12)	176 (5,13)	208 (6,14)	240 (7,15)	272 (4,12)	304 (5,13)	336 (6,14)	368 (7,15)	400 (4,12)	432 (5,13)	464 (6,14)	496 (7,15)
18 (4,12)	50 (5,13)	82 (6,14)	114 (7,15)	146 (4,12)	178 (5,13)	210 (6,14)	242 (7,15)	274 (4,12)	306 (5,13)	338 (6,14)	370 (7,15)	402 (4,12)	434 (5,13)	466 (6,14)	498 (7,15)
20 (4,12)	52 (5,13)	84 (6,14)	116 (7,15)	148 (4,12)	180 (5,13)	212 (6,14)	244 (7,15)	276 (4,12)	308 (5,13)	340 (6,14)	372 (7,15)	404 (4,12)	436 (5,13)	468 (6,14)	500 (7,15)
22 (4,12)	54 (5,13)	86 (6,14)	118 (7,15)	150 (4,12)	182 (5,13)	214 (6,14)	246 (7,15)	278 (4,12)	310 (5,13)	342 (6,14)	374 (7,15)	406 (4,12)	438 (5,13)	470 (6,14)	502 (7,15)
24 (20,28)	56 (21,29)	88 (22.30)	120 (23,31)	152 (20,28)	184 (21,29)	216 (22.30)	248 (23,31)	280 (20,28)	312 (21,29)	344 (22.30)	376 (23,31)	408 (20,28)	440 (21,29)	472 (22.30)	504 (23,31)
26 (20,28)	58 (21,29)	90 (22.30)	122 (23,31)	154 (20,28)	186 (21,29)	218 (22.30)	250 (23,31)	282 (20,28)	314 (21,29)	346 (22.30)	378 (23,31)	410 (20,28)	442 (21,29)	474 (22.30)	506 (23,31)
28 (20,28)	60 (21,29)	92 (22.30)	124 (23,31)	156 (20,28)	188 (21,29)	220 (22.30)	252 (23,31)	284 (20,28)	316 (21,29)	348 (22.30)	380 (23,31)	412 (20,28)	444 (21,29)	476 (22.30)	508 (23,31)
30 (20,28)	62 (21,29)	94 (22.30)	126 (23,31)	158 (20,28)	190 (21,29)	222 (22.30)	254 (23,31)	286 (20,28)	318 (21,29)	350 (22.30)	382 (23,31)	414 (20,28)	446 (21,29)	478 (22.30)	510 (23,31)

Figura 2.10: Relazione tra le posizione nella matrice B (column major) e gli indici dei thread nel caricamento dei dati. Ogni numero prima della parentesi è relativo all'indirizzo nella matrice B (espresso in byte), ogni numero dentro la parentesi è l'indice del thread che carica il dato dalla corrispondente posizione.

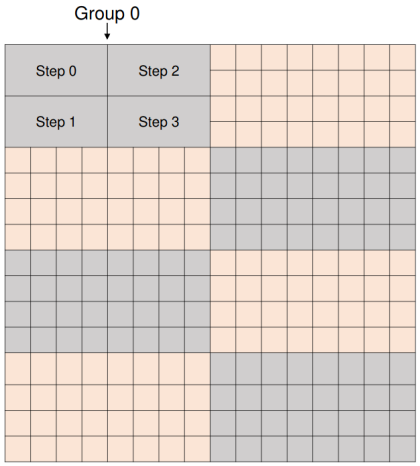
```
1  ; set 0:
2  HMMA.884.F32.F32.STEP0 R8, R26.reuse.T, R16.reuse.T, R8;
3  HMMA.884.F32.F32.STEP1 R10, R26.reuse.T, R16.reuse.T, R10;
4  HMMA.884.F32.F32.STEP2 R4, R26.reuse.T, R16.reuse.T, R4;
5  HMMA.884.F32.F32.STEP3 R6, R26.T, R16.T, R6;
6
7  ; set 1:
8  HMMA.884.F32.F32.STEP0 R8, R20.reuse.T, R18.reuse.T, R8;
9  HMMA.884.F32.F32.STEP1 R10, R20.reuse.T, R18.reuse.T, R10;
10 HMMA.884.F32.F32.STEP2 R4, R20.reuse.T, R18.reuse.T, R4;
11 HMMA.884.F32.F32.STEP3 R6, R20.T, R18.T, R6;
12
13 ; set 2:
14 HMMA.884.F32.F32.STEP0 R8, R22.reuse.T, R12.reuse.T, R8;
15 HMMA.884.F32.F32.STEP1 R10, R22.reuse.T, R12.reuse.T, R10;
16 HMMA.884.F32.F32.STEP2 R4, R22.reuse.T, R12.reuse.T, R4;
17 HMMA.884.F32.F32.STEP3 R6, R22.T, R12.T, R6;
18
19 ; set 3:
20 HMMA.884.F32.F32.STEP0 R8, R2.reuse.T, R14.reuse.T, R8;
21 HMMA.884.F32.F32.STEP1 R10, R2.reuse.T, R14.reuse.T, R10;
22 HMMA.884.F32.F32.STEP2 R4, R2.reuse.T, R14.reuse.T, R4;
23 HMMA.884.F32.F32.STEP3 R6, R2.T, R14.T, R6;
```

Listato 2.1: Codice assembly generato per l'istruzione wmma::mma_sync dal compilatore NVCC

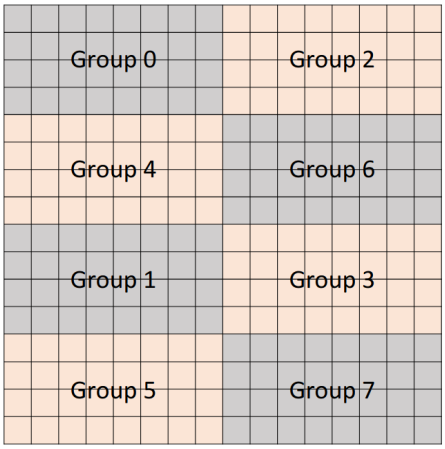
Dopo il caricamento dei dati dalla memoria ai registri, il programmatore deve usare la primitiva wmma::mma_sync per operare la moltiplicazione con i Tensor Core. Il compilatore NVCC traduce la singola primitiva in 4 set di istruzioni HMMA, come mostrato nel listato 2.1.

Ogni set è costituito da quattro istruzioni HMMA con flag che va dallo "SETP0" allo "STEP3". Il registro di destinazione è lo stesso per tutte e quattro le istruzioni dei set; i flag da "STEP0" fino a "STEP3" corrispondono a diverse posizione della matrice C (vedi figura 2.11a).

A tempo di esecuzione, l'architettura Volta divide i 32 thread di un warp in 8 gruppi ($group\_id = thread\_id/4$) e condivide i valori del registro tra tutti i thread dello stesso gruppo e tra i gruppi. Ogni gruppo di thread computa $4 \times 8 = 32$ elementi nella matrice C. La figura 2.11b mostra la realazione tra la posizione della matrice C e gli indici dei gruppi. Per un gruppo di thread tutte le istruzioni HMMA contribuiscono al calcolo della stessa posizione nella matrice C e ognuno di essi accumula e moltiplica gli elementi da parti differenti delle matrici A e B (vedi figura 2.12).



(a) Quattro step di una istruzione HMMA per la computazione di elementi differenti nella matrice C.



(b) Mapping tra la posizione nella matrice C e gli indici dei gruppi di thread.

Figura 2.11: Schema operazione istruzione HMMA e posizioni degli indici nella matrice C.

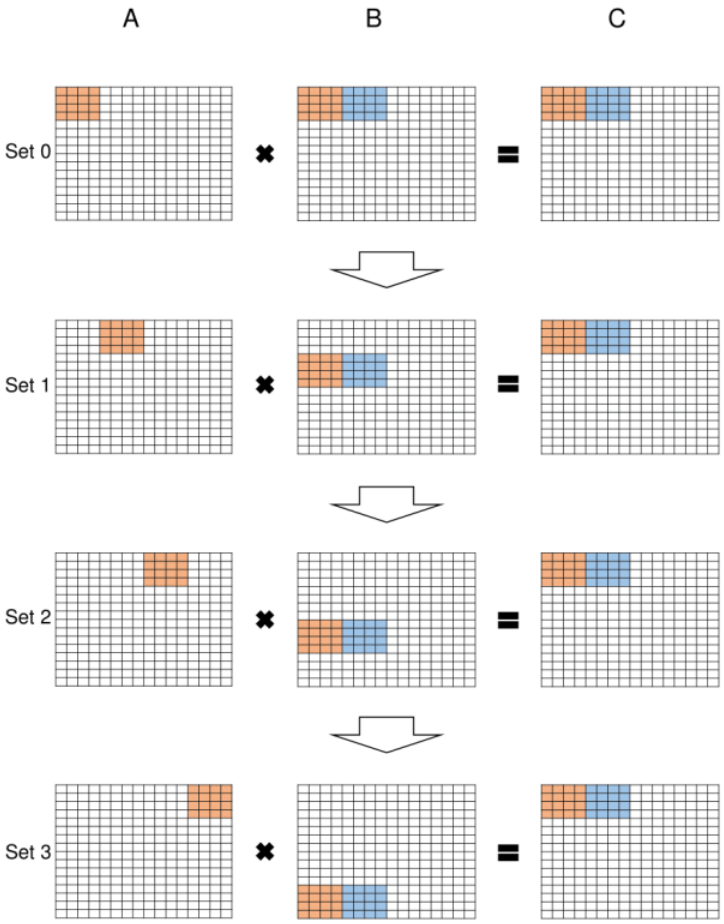


Figura 2.12: Quattro set di istruzioni HMMA completano il risultato parziale 4x8 nella matrice C del gruppo 0. Set differenti usano elementi differenti in A e B. L'istruzione nel set 0 viene eseguita per prima, poi il set 1, il set 2 e infine il set 3. In questo modo l'istruzione HMMA può eseguire correttamente i 4×8 elementi della matrice C.

2.1.4 Studio prestazioni Tensor Core

In questa sezione analizzeremo le prestazioni di varie librerie che fanno uso dei Tensor Core; le librerie prese in considerazione sono: **WMMA API** e **cuBLAS**.

Partendo ad analizzare la prima libreria, possiamo osservare l'uso dei fragment cioè istruzioni che consentono di partizionare la matrice in blocchi 16×16. In questo caso la funzione presente nel listato 2.2 moltiplica una matrice 16x16, quindi non è stato necessario andare a scomporre la matrice in quanto era già delle dimensioni supportate dal fragment. Con le operazioni wmma::load_matrix_sync si carica dalla memoria nei registri i dati delle matrici. Con wmma::mma_sync effettuiamo la moltiplicazione $A \times B + C$ e infine con wmma::store_matrix_sync copiamo il risultato della moltiplicazione dal registro alla memoria centrale.

```
1  __global__ void dot_wmma16x16(half *a, half *b, half *c)
2  {
3    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
4    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
5    wmma::fragment<wmma::accumulator, 16, 16, 16, half> c_frag;
6    wmma::load_matrix_sync(a_frag, a, 16);
7    wmma::load_matrix_sync(b_frag, b, 16);
8    wmma::fill_fragment(c_frag, 0.0f);
9    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
10   wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
11 }
```

Listato 2.2: Uso delle WMMA API per la moltiplicazione di due matrici 16x16.

Se si volesse moltiplicare una matrice di dimensioni maggiori occorre tener presente che la dimensione della matrice (numero colonne e numero righe) deve essere un multiplo di 16, in caso contrario la moltiplicazione produce risultati errati. Nel listato 2.3 è indicata una possibile implementazione di un kernel che fa uso delle **WMMA API** per moltiplicare due matrici di qualunque dimensione purché valgano le condizioni sopra citate.

```
1  __global__ void simple_wmma_gemm(half *a, half *b, float *c, float *d,
2                                   int m_ld, int n_ld, int k_ld,
3                                   float alpha, float beta) {
4    // Leading dimensions. Packed with no transpositions.
5    int lda = m_ld;
6    int ldb = k_ld;
7    int ldc = n_ld;
8
9    // Tile using a 2D grid
10   int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
11   int warpN = (blockIdx.y * blockDim.y + threadIdx.y);
12
13   // Declare the fragments
14   wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half,
15                  wmma::row_major> a_frag;
16   wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half,
17                  wmma::col_major> b_frag;
18   wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float>
19                  acc_frag;
```

```
20    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float>
21                   c_frag;
22    wmma::fill_fragment(acc_frag, 0.0f);
23
24    // Loop over k
25    for (int i = 0; i < k_ld; i += WMMA_K) {
26      int aCol = i;
27      int aRow = warpM * WMMA_M;
28      int bCol = i;
29      int bRow = warpN * WMMA_N;
30
31      // Bounds checking
32      if (aRow < m_ld && aCol < k_ld && bRow < k_ld && bCol < n_ld) {
33        // Load the inputs
34        wmma::load_matrix_sync(a_frag, a + aCol + aRow * lda, lda);
35        wmma::load_matrix_sync(b_frag, b + bCol + bRow * ldb, ldb);
36
37        // Perform the matrix multiplication
38        wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
39      }
40    }
41
42    // Load in the current value of c, scale it by beta,
43    // and add this our result
44    // scaled by alpha
45    int cCol = warpN * WMMA_N;
46    int cRow = warpM * WMMA_M;
47
48    if (cRow < m_ld && cCol < n_ld) {
49      wmma::load_matrix_sync(c_frag, c + cCol + cRow * ldc, ldc,
50                             wmma::mem_row_major);
51
52      for (int i = 0; i < c_frag.num_elements; i++) {
53        c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
54      }
55
56      // Store the output
57      wmma::store_matrix_sync(d + cCol + cRow * ldc, c_frag, ldc,
58                              wmma::mem_row_major);
59    }
60  }
```

Listato 2.3: Implementazione di un kernel che fa uso delle WMMA API per moltiplicare matrici di qualsiasi dimensioni.

Il test condotto è il seguente: si sono scelte diverse implementazioni del calcolo della moltiplicazione tra matrici e si sono misurate i tempi di esecuzione prodotti da ogni singola implementazione andando a variare la dimensione delle matrici; i test sono stati eseguiti con valori piuttosto elevati in termini di dimensioni delle matrici in quanto l'architettura che fa uso di Tensor Core raggiunge le prestazioni migliori quando gran parte, o meglio tutti i Tensor Core sono attivati. Quello che possiamo osservare dalla figura 2.13 è che, prendendo come riferimento la versione CUDA, la libreria **cuBLAS** è quella che ha ottenuto le prestazioni migliori in qualsiasi circostanza; questo non ci sorprende dal

momento che è una libreria studiata e ottimizzata per le GPU NVIDIA, per far uso dei Tensor Core e per eseguire calcoli di algebra lineare tra cui la moltiplicazione tra matrici. Potrebbe sorprendere invece la libreria che fa uso delle **WMMA API** in quanto il degrado delle loro prestazioni rispetto a **cuBLAS** è quasi sempre di 8 volte inferiore e sapendo che anche questa implementazione fa uso dei Tensor Core potrebbe stupire. Se pensiamo però che l'esempio proposto non fa uso di Shared Memory, ne di tecniche particolari per l'ottimizzazione dei calcoli o meglio ancora tecniche studiate appositamente per eseguire i calcoli più velocemente sulle GPU, questo risultato è quantomeno plausibile. Se considerassimo soltanto l'uso della Shared Memory (che possiamo osservare nell'implementazione **CUDA (SHM)** senza Tensor Core), allora possiamo vedere che utilizzandola si ottiene uno speedup di quasi due volte.

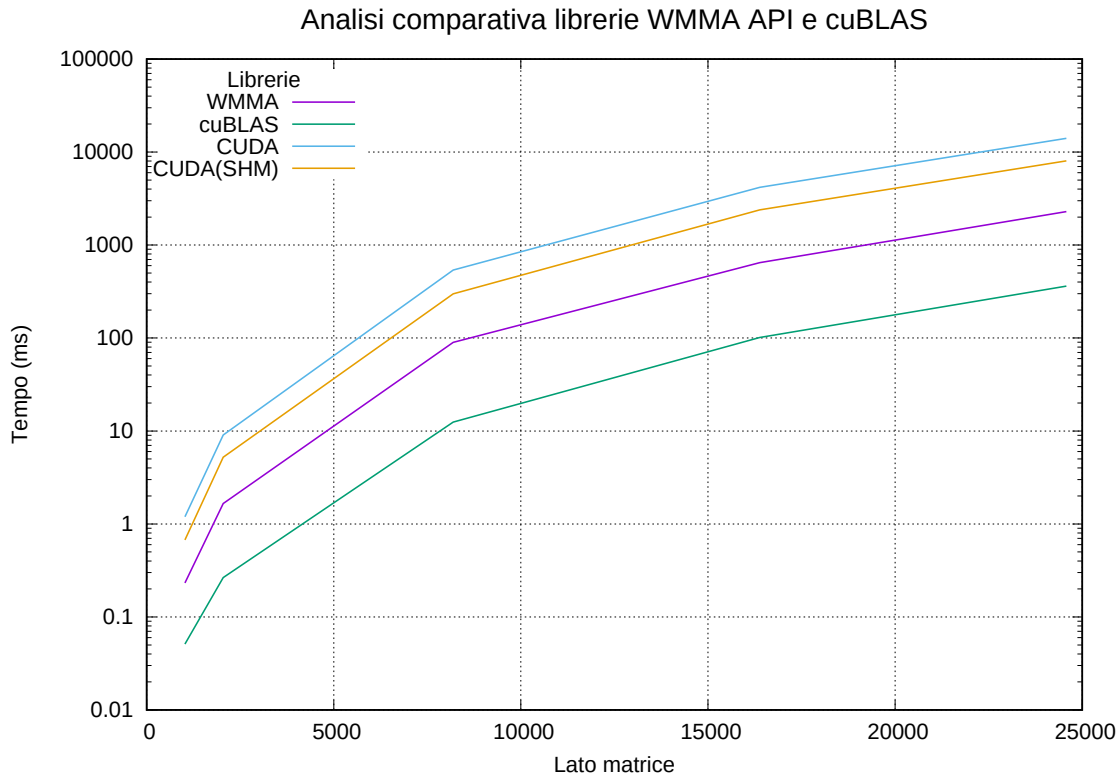


Figura 2.13: Confronto tra librerie per la moltiplicazione tra matrici.

In definitiva possiamo affermare che qualora sia necessario moltiplicare matrici di grandi dimensioni, la soluzione migliore risulta essere la libreria **cuBLAS**. Dal momento che questa libreria viene sviluppata e distribuita da NVIDIA, non è possibile conoscerne i dettagli implementativi: infatti NVIDIA non rilascia i relativi codici sorgenti, quindi risulta molto difficile capire in che modo questa libreria operi e che ottimizzazioni siano state eseguite.

Prendendo quindi come base di partenza per la misurazione delle prestazioni la versione CUDA base, possiamo osservare che **cuBLAS** ottiene le prestazioni migliori avendo uno speedup che varia dalle 30 alle 40 volte rispetto alla versione CUDA.

Capitolo 3

Operatore di Dirac e Tensor Core

In questo capitolo viene analizzato il funzionamento dell'algoritmo per il calcolo dell'operatore di Dirac. Questo operatore viene utilizzato in simulazioni Lattice QCD e rappresenta l'operazione principale poiché il tempo impiegato per il suo calcolo va dal 40% all'80% del tempo totale impiegato dalla simulazione [13]. Ottimizzarne l'efficienza permette di velocizzare le prestazioni dell'intero algoritmo di simulazione.

3.1 Funzionamento operatore di Dirac

L'operatore di Dirac effettua una serie di moltiplicazioni tra matrici 3×3 e vettori 3×1, entrambi composti da numeri complessi. Le matrici e i vettori (anche detti *fermioni*) rappresentano uno schema ben preciso identificato da un ipercubo a quattro dimensioni, o *lattice*, le cui dimensioni sono: $nx \times ny \times nz \times nt$.

Questo ipercubo presenta due componenti fondamentali: i siti e i collegamenti tra essi. Ogni sito è collegato a tutti i suoi 8 vicini (due per dimensione). Un sito è rappresentato da un fermione (ovvero un vettore 3×1), mentre ogni collegamento è rappresentato dalla matrice 3×3. In figura 3.1 viene mostrato un esempio di lattice in tre dimensioni poiché un lattice quadridimensionale è difficile da rendere graficamente. I siti che risiedono sul bordo sono vicini ai siti presenti sul bordo opposto, definendo quindi un dominio ciclico (vedi figura 3.2). Le due operazioni principali che l'operatore di Dirac effettua sono Deo (Dirac even odd) e Doe (Dirac odd even). Tali operazioni definiscono come devono essere letti i valori dei siti: Deo legge i valori dei siti la cui somma delle coordinate è un numero pari e scrive il risultato sui siti la cui somma delle coordinate è un numero dispari; Doe effettua l'operazione opposta.

Le due operazioni appena citate agiscono sulla metà dei dati del dominio. Nel listato 3.1 possiamo osservare una possibile implementazione di come possano essere iterati i siti.

```
1 for (unsigned t=0; t<nt; t++)
2   for (unsigned z=0; z<nz; z++)
3     for (unsigned y=0; y<ny; y++)
4       for (unsigned hx=0; hx<nxh; hx++) {
5           //Compute output vector of site idxh
6       }
```

Listato 3.1: Codice per l'iterazione dei siti del lattice.

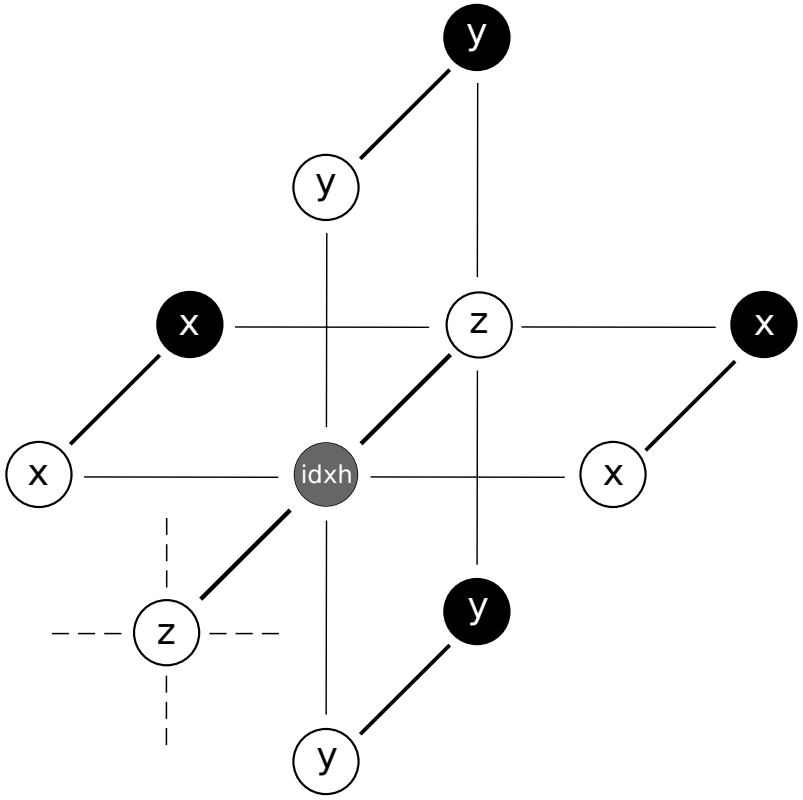


Figura 3.1: Esempio di un lattice in tre dimensioni. idxh identifica il fermione che si sta valutando.

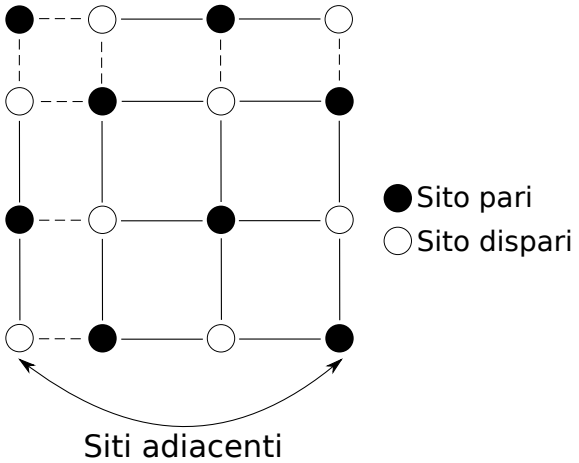


Figura 3.2: Esempio di un lattice in due dimensioni in cui si evidenzia l'adiacenza dei siti sul bordo.

La coordinata x, come appena detto, scorre solamente metà dei dati in quanto solo i siti pari e i siti dispari vengono valutati rispettivamente dalle funzioni Deo e Doe. Per questo motivo il ciclo for più interno va da zero fino a nxh, che rappresenta la metà del dominio di x.

Il ciclo più interno presente in Deo e Doe serve a selezionare i vicini di un sito. Per ogni vicino v_i si moltiplicare la matrice rappresentata dal collegamento di v_i al corrente con il fermione rappresentato da v_i (vedi listati 3.2 e 3.3).

```
1  x = 2*hx
2
3  if (((y+z+t) % 2) != 0) x++
4
5  aux  = A0[x,y,z,t] x V[x+1,y,z,t]
6  aux += A2[x,y,z,t] x V[x,y+1,z,t]
7  aux += A4[x,y,z,t] x V[x,y,z+1,t]
8  aux += A6[x,y,z,t] x V[x,y,z,t+1]
9
10 aux -= CA1[x-1,y,z,t] x V[x-1,y,z,t]
11 aux -= CA3[x,y-1,z,t] x V[x,y-1,z,t]
12 aux -= CA5[x,y,z-1,t] x V[x,y,z-1,t]
13 aux -= CA7[x,y,z,t-1] x V[x,y,z,t-1]
14
15 endFremion[x,y,z,t] = aux/2
```

Listato 3.2: Pseudo codice ciclo interno di Deo.

```
1  x = 2*hx
2
3  if (((y+z+t) % 2) == 0) x++
4
5  aux  = A1[x,y,z,t] x V[x+1,y,z,t]
6  aux += A3[x,y,z,t] x V[x,y+1,z,t]
7  aux += A5[x,y,z,t] x V[x,y,z+1,t]
8  aux += A7[x,y,z,t] x V[x,y,z,t+1]
9
10 aux -= CA0[x-1,y,z,t] x V[x-1,y,z,t]
11 aux -= CA2[x,y-1,z,t] x V[x,y-1,z,t]
12 aux -= CA4[x,y,z-1,t] x V[x,y,z-1,t]
13 aux -= CA6[x,y,z,t-1] x V[x,y,z,t-1]
14
15 endFremion[x,y,z,t] = aux/2
```

Listato 3.3: Pseudo codice ciclo interno di Doe.

Sia in Deo che in Doe vengono svolte le stesse operazioni, la principale differenza sta negli indici delle matrici e dei vettori da moltiplicare. Le variabili Ax rappresentano le matrici 3×3, mentre le variabili V rappresentano i vettori 3×1. La variabile aux è anch'essa un vettore 3×1 in cui si memorizza il risultato. La variabile CAx indica il complesso coniugato della matrice 3×3.

L'operazione che viene eseguita maggiormente è la moltiplicazione tra matrice 3×3 e vettore 3×1, quindi se si ottimizzasse questa operazione si ridurrebbero i tempi per il calcolo del fermione risultante.

3.1.1 Complessità

La complessità dell'algoritmo è data dai quattro cicli innestati nelle funzioni **Deo** e **Doe**. Le operazioni all'interno dei cicli hanno tutte complessità costante, quindi possiamo definire la complessità dell'algoritmo come:

$$\Theta(\text{NITER} \times nx \times ny \times nz \times nt)$$

Dove NITER indica in numero di chiamate a Deo e Doe, nx, ny, nz e nt rappresentano le dimensioni del lattice. In una simulazione completa NITER è dell'ordine delle migliaia.

3.2 Strutture dati

Analizziamo ora la struttura dati in cui sono memorizzate le informazioni. Sia le matrici che i vettori sono memorizzati in Strutture di Array (SoA). Per quanto riguarda le matrici, queste sono suddivise in un array di 8 elementi, ognuno dei quali memorizza tutte le matrici di tutti i sizeh siti. Ad esempio definiamo u il vettore di 8 elementi, allora u[0] è una struttura che contiene la matrice A0 dei siti $0, 1, 2, \ldots sizeh - 1$ (vedi figura 3.3).

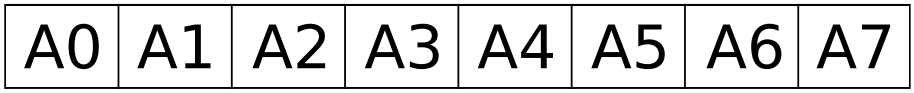


Figura 3.3: Array di 8 elementi contenenti le matrici dei siti.

All'interno di ogni elemento di u sono allocati tre vettori: r0, r1 ed r2, questi rappresentano le 3 righe di tutte le sizeh matrici A_i, dove i rappresenta l'indice dell'elemento di u. Nel listato 3.4 si riporta la dichiarazione della struttura dati su3_soa.

```
1  typedef struct vec3_soa_t {
2    d_complex c0[sizeh];
3    d_complex c1[sizeh];
4    d_complex c2[sizeh];
5  } vec3_soa;
6
7  typedef struct su3_soa_t {
8    vec3_soa r0;
9    vec3_soa r1;
10   vec3_soa r2;
11 } su3_soa;
```

Listato 3.4: Dichiarazione struttura dati su3_soa

Ad esempio u[0]->r0 è una struttura dati che ospita la prima riga (composta da tre numeri complessi) delle matrici dei sizeh siti. In figura 3.4 viene mostrata la rappresentazione di una matrice; nella figura r0, r1 e r2 sono indicati uno sotto all'altro, ma è bene precisare che in memoria le righe sono adiacenti.

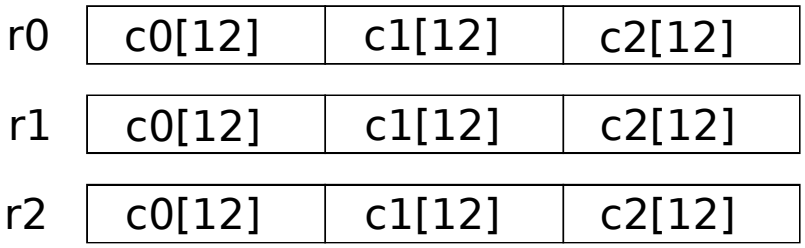


Figura 3.4: Struttura dati contenuta in ogni elemento di u. In questo esempio abbiamo riportato la matrice di indice 12. Le righe sono adiacenti in memoria.

Ad esempio, accedendo ad u[2]->r0.c2[34] viene letto il valore della terza colonna (c2), prima riga (r0) della matrice A2 del sito 34.

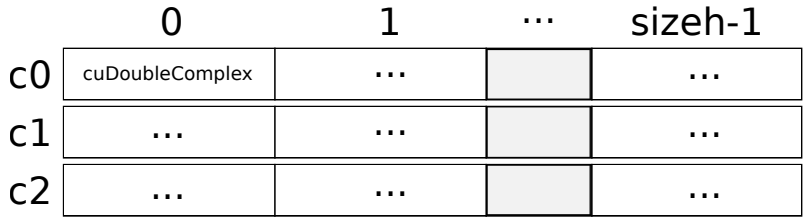


Figura 3.5: Rappresentazione grafica della struttura dati vec3_soa.

Per quanto riguarda i vettori invece, la struttura dati che viene usata è simile a quella delle matrici: si utilizza sempre una struttura di tipo SoA (Struttura di Array) in cui sono presenti 3 componenti rappresentate da un vettore lungo sizeh. La struttura vec3_soa, analizzata in precedenza nel listato 3.4 è la stessa struttura che viene utilizzata per rappresentare i vettori. L'utilizzo dei vettori nella struttura delle matrici è dovuto al fatto che una matrice 3×3 può essere rappresentata come una serie di 3 vettori colonna 3×1 adiacenti, formando quindi una matrice 3×3. In figura 3.5 viene mostrata la struttura dati vec3_soa.

3.2.1 Tensor Core

Vediamo come poter utilizzare i Tensor Core per migliorare le prestazioni dell'operatore di Dirac. Come visto nella sezione precedente l'algoritmo si occupa principalmente di effettuare delle moltiplicazioni matrice-vettore; questo tipo di operazione differisce leggermente dall'operazione che i Tensor Core sono in grado di effettuare, ovvero moltiplicazioni tra matrici. Possiamo però vedere una moltiplicazione matrice-vettore come una moltiplicazione tra matrici supponendo di avere il vettore disposto lungo la prima colonna della seconda matrice e le celle restanti della seconda matrice poste a 0. In questo modo gli zeri che compaio nella seconda matrice non alterano il risultato della moltiplicazione e, sapendo che una moltiplicazione matrice-vettore produce un vettore, esso sarà presente nella prima colonna della matrice risultato. In figura 3.6 è espresso graficamente il concetto appena illustrato.

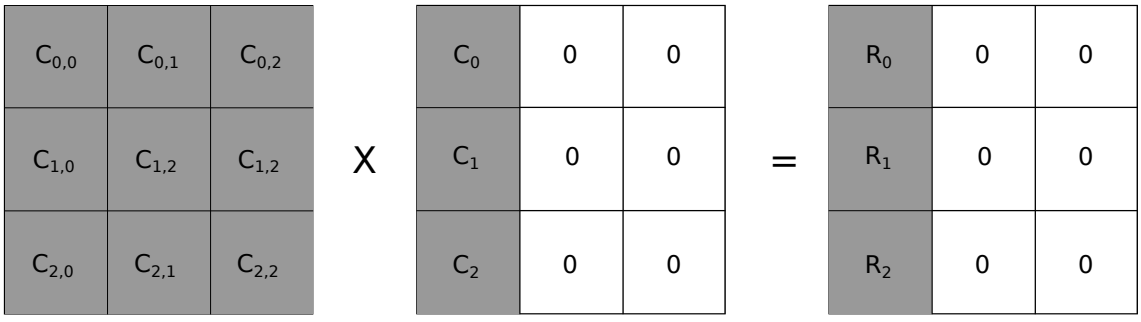


Figura 3.6: Esempio di moltiplicazione matrice-vettore utilizzando una matrice come secondo operando.

Ricordiamo che i Tensor Core possono moltiplicare matrici con dimensione minima

pari a 16×16, quindi occorre inserire le matrici 3×3 e i vettori 3×1 nella matrice più grande 16×16; inoltre i Tensor Core effettuano calcoli su dati di tipo half, ovvero numeri in virgola mobile a 16-bit invece i calcoli di nostro interesse sono con numeri complessi.

Come accennato in precedenza, l'operatore di Dirac prevede l'uso di numeri complessi, quindi le matrici e i vettori con cui abbiamo a che fare sono di natura complessa; questo rende più complicati i calcoli: la moltiplicazione di due matrici di numeri complessi avviene scomponendo la matrice (o il vettore) in due matrici della stessa dimensione, in cui una contiene solo la parte reale e l'altra la parte immaginaria.

$$\mathbf{M}_c = \mathbf{M}_r + i\mathbf{M}_i$$

Dove M_r rappresenta la matrice della parte reale e M_i rappresenta la matrice della parte immaginaria. Quindi per moltiplicare due matrici di numeri complessi occorre tener presente che il calcolo del prodotto tra due numeri complessi si effettua come segue:

$$\begin{aligned} z_1 &= a + ib \\ z_2 &= c + id \\ z_1 \cdot z_2 &= (ac - bd) + i(ad + bc) \end{aligned} \tag{3.1}$$

Portiamo questo esempio con matrici. Consideriamo m_1 e m_2 due matrici di numeri complessi, quindi operiamo i calcoli definendo t_1, t_2, t_3 e t_4 i risultati parziali dei soli prodotti:

$$\begin{aligned} \mathbf{T}_1 &= \mathbf{M}_{1r} \cdot \mathbf{M}_{2r} \\ \mathbf{T}_2 &= \mathbf{M}_{1i} \cdot \mathbf{M}_{2i} \\ \mathbf{T}_3 &= \mathbf{M}_{1r} \cdot \mathbf{M}_{2i} \\ \mathbf{T}_4 &= \mathbf{M}_{1i} \cdot \mathbf{M}_{2r} \end{aligned} \tag{3.2}$$

Ottenuti i prodotti ora andiamo a svolgere le somme e le differenze:

$$\mathbf{M}_1 \cdot \mathbf{M}_2 = (\mathbf{T}_1 - \mathbf{T}_2) + i(\mathbf{T}_3 + \mathbf{T}_4) \tag{3.3}$$

In conclusione, otteniamo due matrici (una per la parte reale e una per la parte immaginaria) che rappresentano il risultato finale della moltiplicazione. In questo esempio si è fatto uso di due matrici invece che matrice e vettore, ma il concetto rimane il medesimo, ovvero si trasforma il vettore in una matrice (come spiegato in precedenza), si effettua la moltiplicazione e infine si prende solo il vettore colonna della matrice risultante.

Un possibile modo di operare questo calcolo con i Tensor Core è quello di inserire la matrice in alto a sinistra e riempire il resto della matrice con zeri (vedi figura 3.7). Questo ragionamento lo si applica ai numeri complessi scomponendo la parte reale e quella immaginaria in due matrici distinte. In questo modo si spreca molta memoria oltre a far eseguire calcoli non utili ai fini del risultato.

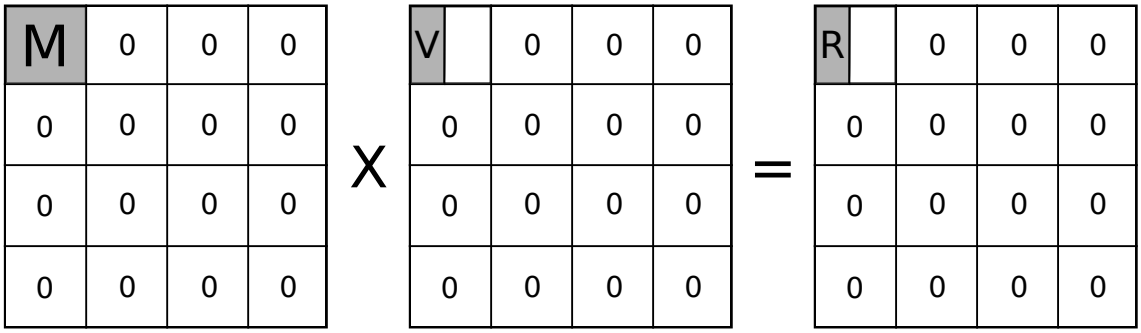


Figura 3.7: Prima soluzione per utilizzare i Tensor Core per moltiplicazione matrice-vettore.

Un altro possibile modo di operare è quello di inserire le matrici 3×3 (e anche i relativi vettori 3×1) lungo la diagonale della matrice 16×16 (vedi figura 3.8). In questo modo gli sprechi in termini di memoria vengono leggermente ridotti e con una sola moltiplicazione di una matrice (16×16) otteniamo 4 vettori come risultato sempre disposti lungo la diagonale della matrice. In figura 3.8 viene illustrata la soluzione appena descritta.

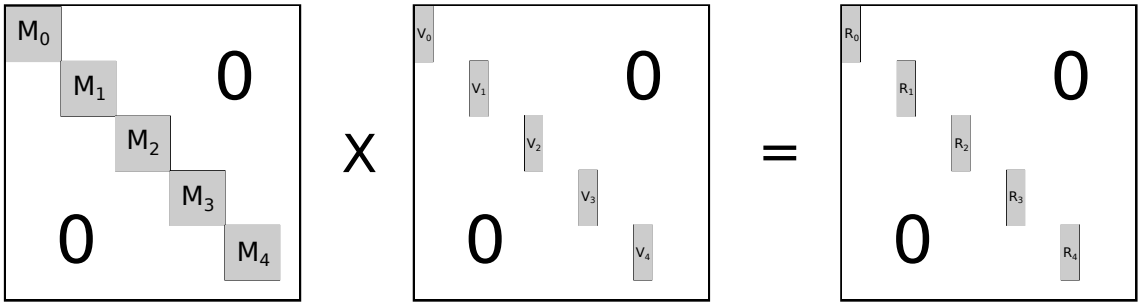


Figura 3.8: Seconda soluzione per utilizzare i Tensor Core per moltiplicazione matrice-vettore.

Seppur l'ultima soluzione proposta riduca lo spreco di memoria, il numero di matrici rappresentate all'interno di matrici 16×16 usate dai Tensor Core è ancora piuttosto basso. Nella sezione 3.2.2 analizzeremo nel dettaglio le prestazioni di ogni soluzione, evidenziando i punti favorevoli e sfavorevoli di ognuna.

Le soluzioni appena proposte, oltre ad usare molta memoria, necessitano di quattro moltiplicazioni tra matrici, una somma e una sottrazione e questo potrebbe influire negativamente sulle prestazioni dell'algoritmo. Occorre trovare un modo in cui con una sola moltiplicazione effettuata dai Tensor Core venga prodotto già un risultato completo, ovvero comprendente sia parte reale che parte immaginaria.

L'ultimo modo che analizziamo ha uno spreco minimo in termini di memoria, ma con una sola moltiplicazione si ottengono 8 vettori come risultato. Partiamo analizzando solo la matrice che contiene le matrici 3×3: queste ultime possiamo inserirle in una matrice 4×4 allineandole in alto a sinistra (vedi figura 3.9)

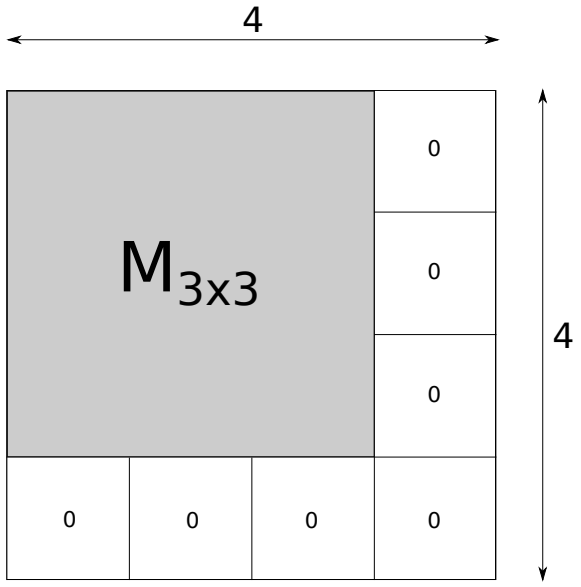


Figura 3.9: Matrice 3×3 inserita in una matrice 4×4 con padding di zeri per l'allineamento.

A questo punto avendo una matrice 4×4 possiamo inserire esattamente 16 matrici in quella 16×16, avendo uno spreco di memoria irrisorio. Lavorando con i numeri complessi possiamo disporre le matrici come rappresentato in figura 3.10 in modo da poter effettuare 8 prodotti matrice-vettore e ottenere direttamente il risultato sotto forma di numero complesso.

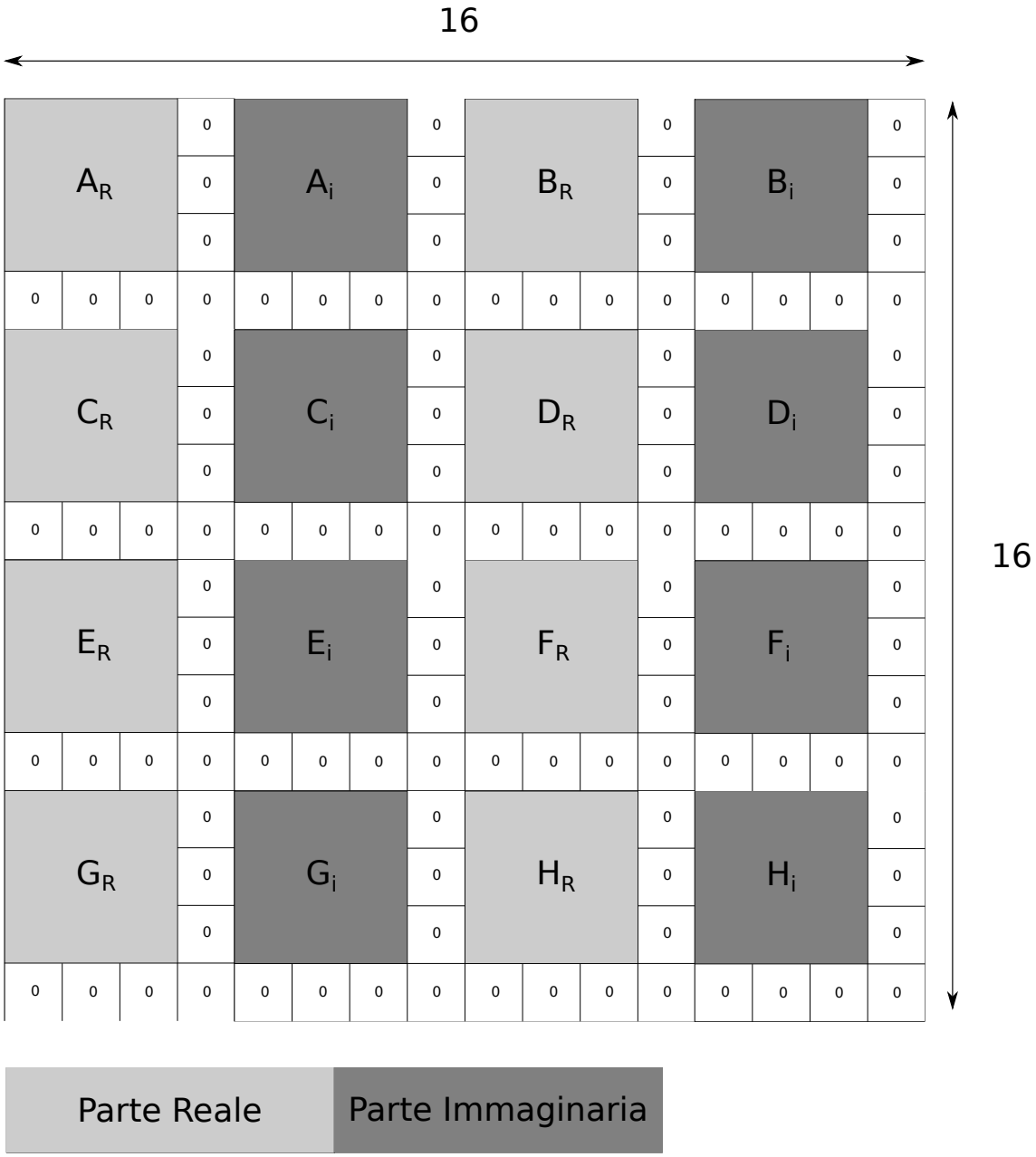


Figura 3.10: Matrice 16×16 in cui sono presenti 16 matrici 3×3 disposte per il calcolo con Tensor Core.

Analizzando la matrice illustrata in figura 3.10 possiamo osservare che lungo le righe della matrice sono presenti quattro matrici 3×3 che rappresentano la parte reale e immaginaria. In questo caso non serve scomporre parte reale e parte immaginaria in due matrici separate, ma la separazione coesiste nella stessa matrice. Da qui possiamo ottenere il risultato direttamente in un unica matrice senza dover "comporre" le varie matrici come illustrato nei precedenti modi.

Analizziamo ora la disposizione dei vettori 3×1 nella matrice 16×16. La loro disposizione segue uno schema che consente di effettuare le quattro moltiplicazioni matrice-vettore, la somma e la sottrazione sfruttando le proprietà della moltiplicazione tra matrici.

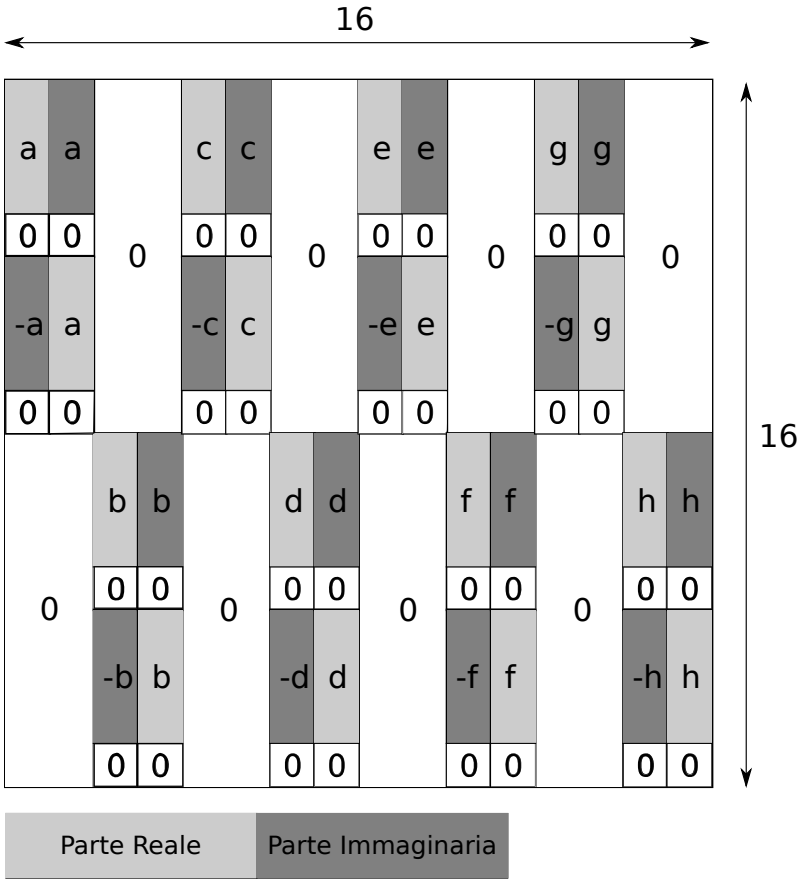


Figura 3.11: Disposizione dei vettori 3×1 nella matrice 16×16.

Moltiplicando le due matrici appena illustrate otteniamo come risultato una matrice in cui lungo la diagonale sono presenti i vettori ottenuti come risultato dalla moltiplicazione già suddivisi in parte reale e parte immaginaria. In figura 3.12 viene illustrato il risultato della moltiplicazione matrice-vettore.

Con questa particolare disposizione delle matrici e dei vettori all'interno della matrice 16×16 è possibile effettuare otto moltiplicazioni matrice-vettore in una sola moltiplicazione tra matrici effettuata dai Tensor Core. Questo modo semplifica di molto la gestione dei dati in memoria in quanto non è necessario combinare risultati parziali come succede con i metodi illustrati precedentemente; infine la memoria non utilizzata risulta essere molto inferiore rispetto alle soluzioni illustrate in precedenza.

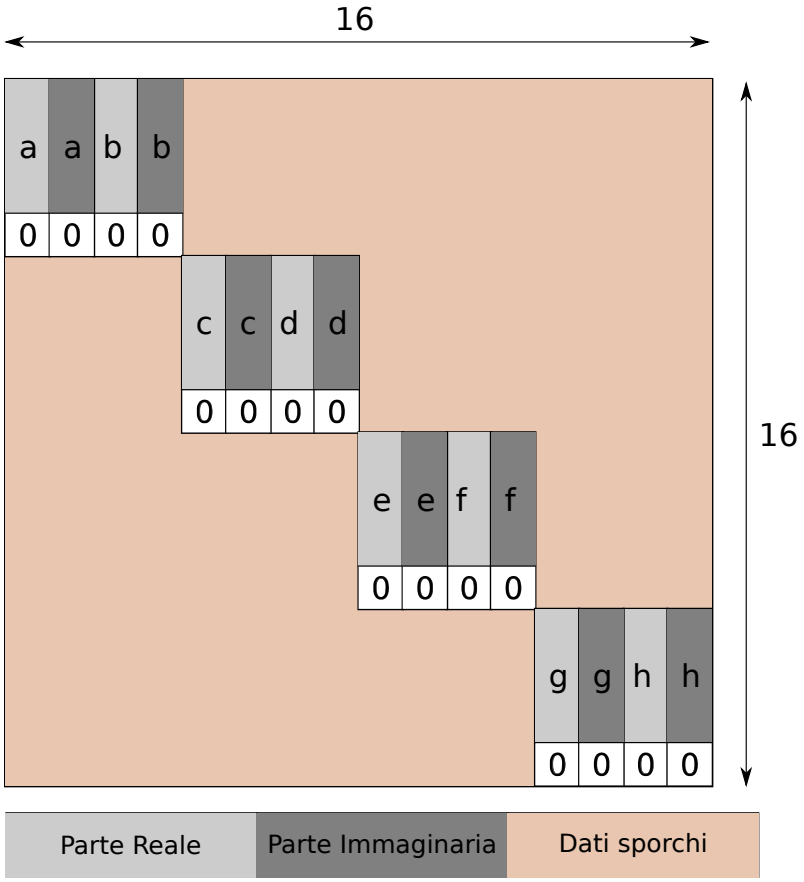


Figura 3.12: Matrice rappresentante il risultato contenente gli otto vettori complessi.

Nel listato 3.5 possiamo osservare la moltiplicazione degli otto vicini al sito corrente. Data una coordinata (x,y,z,t) le variabili xp, yp, zp e tp rappresentano la coordinata di indice +1, ovvero la coordinata che rappresenta il sito adiacente; le coordinate xm, ym, zm e **tm** indicano la coordinata di indice −1. Queste otto coordinate servono per il calcolo della moltiplicazione matrice-vettore con gli otto siti adiacenti lungo le quattro dimensioni del lattice.

```
1  ...
2  mat_vec_mul( &u[1], idxh, eta, in, snum(xp,y,z,t), &aux_tmp );
3  aux = aux_tmp;
4
5  eta = ETA_UPDATE(x);
6  mat_vec_mul( &u[3], idxh, eta, in, snum(x,yp,z,t), &aux_tmp );
7  aux = sumResult(aux, aux_tmp);
8
9  eta = ETA_UPDATE(x+y);
10 mat_vec_mul( &u[5], idxh, eta, in, snum(x,y,zp,t), &aux_tmp );
11 aux = sumResult(aux, aux_tmp);
12
13 eta = ETA_UPDATE(x+y+z);
14 mat_vec_mul( &u[7], idxh, eta, in, snum(x,y,z,tp), &aux_tmp );
```

```
15  aux = sumResult(aux, aux_tmp);
16  ////////////////////////////////////////////////////////////////////////
17  eta = 1;
18  conjmat_vec_mul( &u[0], snum(xm,y,z,t), eta, in, snum(xm,y,z,t), &aux_tmp );
19  aux = subResult(aux, aux_tmp);
20
21  eta = ETA_UPDATE(x);
22  conjmat_vec_mul( &u[2], snum(x,ym,z,t), eta, in, snum(x,ym,z,t), &aux_tmp );
23  aux = subResult(aux, aux_tmp);
24
25  eta = ETA_UPDATE(x+y);
26  conjmat_vec_mul( &u[4], snum(x,y,zm,t), eta, in, snum(x,y,zm,t), &aux_tmp );
27  aux = subResult(aux, aux_tmp);
28
29  eta = ETA_UPDATE(x+y+z);
30  conjmat_vec_mul( &u[6], snum(x,y,z,tm), eta, in, snum(x,y,z,tm), &aux_tmp );
31  aux = subResult(aux, aux_tmp)
32  ...
```

Listato 3.5: Moltiplicazione siti adiacenti.

Possiamo sfruttare i Tensor Core per effettuare otto moltiplicazioni matrice-vettore per ogni sito adiacente: per ognuna delle otto moltiplicazioni, invece di eseguire solo una moltiplicazione matrice-vettore ne eseguiamo otto facendo uso dei Tensor Core; in questo modo garantiamo il fatto che effettuiamo moltiplicazioni matrice-vettore di siti adiacenti sia sul lattice che in memoria, aspetto fondamentale che necessitano i Tensor Core per un corretto funzionamento.

Il listato 3.6 mostra lo pseudo-codice di come possono essere usati i Tensor Core per effettuare la moltiplicazione di matrici e vettori adiacenti.

```
1   ...
2
3   t =   (blockIdx.z * blockDim.z + threadIdx.z) / nz;
4   z =   (blockIdx.z * blockDim.z + threadIdx.z) % nz;
5   y =   (blockIdx.y * blockDim.y + threadIdx.y);
6   x = 2*(blockIdx.x * blockDim.x + threadIdx.x) + ((y+z+t+1) & 0x1)
7
8   for (unsigned i=0; i<NUM*2; i+=2)
9       idxh[i/2] = snum(x+i,y,z,t);
10
11  // Calcolo xp, yp, zp, tp
12  // Calcolo xm, ym, zm, tm
13
14  tensor_mat_vec_mul( &u[1], idxh, eta, in, snum(xp,y,z,t), &aux_tmp );
15  aux = aux_tmp;
16
17  eta = ETA_UPDATE(x);
18  tensor_mat_vec_mul( &u[3], idxh, eta, in, snum(x,yp,z,t), &aux_tmp );
19  aux = sumResult(aux, aux_tmp);
20
21  eta = ETA_UPDATE(x+y);
22  tensor_mat_vec_mul( &u[5], idxh, eta, in, snum(x,y,zp,t), &aux_tmp );
23  aux = sumResult(aux, aux_tmp);
```

```
24
25  // Altre moltiplicazioni...
26  ...
```

Listato 3.6: Pseudo-codice funzionamento operatore di Dirac con Tensor Core.

Il calcolo degli indici **y**, **z** e **t** viene effettuato nello stesso modo della versione che non fa uso dei Tensor Core ad eccezione di **x** che assume valori multipli di 8 in quanto i Tensor Core eseguono otto moltiplicazioni matrice-vettore alla volta. Allo stesso modo idxh assume valori multipli di 8 per la stessa ragione.

In figura 3.13 viene mostrato graficamente come i Tensor Core elaborano i siti del lattice lungo una coordinata. Nell'immagine il sito evidenziato di rosso scuro indica il sito corrente, mentre i siti in rosso chiaro indicano i siti adiacenti che i Tensor Core elaborano in una sola moltiplicazione. In questo caso è stato rappresentato un lattice in due dimensioni, ma il concetto è espandibile anche per il lattice di quattro dimensioni.

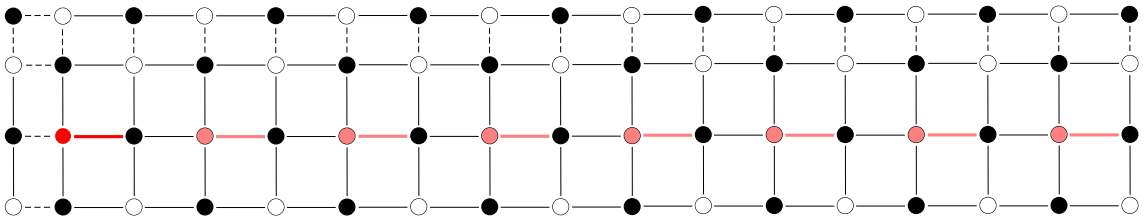


Figura 3.13: Esempio di elaborazione di 8 siti adiacenti lungo una coordinata con Tensor Core.

Possiamo osservare che i risultati parziali di ogni iterazione del lattice vengono accumulati nella variabile aux. In questa variabile, al termine della computazione, vi è il vettore risultato che verrà poi salvato in memoria. Con i Tensor Core è possibile ottimizzare questa operazione sfruttando l'operazione di accumulo prevista nella moltiplicazione tra le matrici. Si ricorda che l'operazione che i Tensor Core eseguono è la seguente:

$$\mathbf{C} = \mathbf{A} \times \alpha\mathbf{B} + \beta\mathbf{C}$$

Fino a questo momento abbiamo analizzato come sfruttare la moltiplicazione con i Tensor Core, vediamo ora come è possibile sfruttare l'accumulo (somma di **C**). Dal momento che dobbiamo sommare o sottrarre i risultati parziali delle moltiplicazioni possiamo sfruttare l'accumulo in **C** nei vari passaggi delle moltiplicazioni. Nel listato 3.5 vengono eseguite le prime quattro moltiplicazioni in cui i risultati vengono sommati ad aux, mentre nelle restanti quattro moltiplicazioni i risultati parziali vengono sottratti ad aux; possiamo quindi sfruttare la somma della matrice **C** nei primi quattro casi ponendo $\beta = 1$ in modo che il risultato della moltiplicazione $\mathbf{A} \times \mathbf{B}$ venga sommato a **C**. Per le restanti quattro operazioni, in cui deve essere sottratto il risultato parziale, si pone $\beta = -1$ in modo da sottrarre i risultati. Al termine della computazione otterremmo nella matrice **C** gli otto vettori ottenuti dalla moltiplicazione con Tensor Core.

Così facendo sfruttiamo la somma nel calcolo con Tensor Core senza eseguirla esplicitamente, riducendo le operazioni. Dal momento che i Tensor Core, indipendentemente

dal valore di β, eseguono anche l'operazione di accumulo, risulta vantaggioso sfruttare questa operazione piuttosto che implementare una procedura che effettui solo la somma (o sottrazione) dei risultati parziali. Sfruttare questa operazione può portare ad un ulteriore incremento delle prestazioni generali dell'algoritmo.

3.2.2 Analisi delle prestazioni

Esaminiamo le prestazioni delle soluzioni proposte nella sezione precedente, analizzandone efficienza e tempi stimati di esecuzione. Si prende come riferimento il tempo per moltiplicare una matrice 3×3 con un vettore 3×1, nonché l'operazione che viene effettuata dall'operatore di Dirac e questo tempo è circa di $10ns$.

Analisi soluzione 1

Questa soluzione prevede di inserire una singola matrice 3×3 e il vettore 3×1 rispettivamente nelle matrici 16×16 allineate in alto a sinistra. Il kernel è stato eseguito con un blocco di dimensioni pari a 1024 e una griglia di dimensioni pari a 60. Le prestazioni ottenute con questa soluzione per moltiplicare 1920 matrici sono:

- Versione CUDA: $23.6\mu s$, oppure $\sim 11ns$/matrice.
- Versione Tensor Core: $130ms$, oppure $\sim 68\mu s$/matrice.

I risultati ottenuti per questa soluzione sono alquanto deludenti: esiste un divario di più di un ordine di grandezza tra la versione con Tensor Core e CUDA, rendendo impraticabile questa soluzione.

Analisi Soluzione 2

Con questa soluzione disponiamo cinque matrici 3×3 e quattro vettori 3×1 lungo la diagonale delle matrici 16×16. Il kernel è stato eseguito con un blocco di dimensioni pari a 1920 e una griglia di dimensioni pari a 32. Le prestazioni ottenute da questa soluzione per moltiplicare 1920 matrici sono:

- Versione CUDA: $27.5\mu s$, oppure $\sim 12ns$/matrice.
- Versione Tensor Core: $67.86\mu s$, oppure $\sim 35.3ns$/matrice.

Come si evince dai tempi il kernel che fa uso dei Tensor Core non produce nessun miglioramento ma peggiora di più del doppio le prestazioni rispetto alla versione CUDA.

Questo peggioramento è dovuto al fatto che nella matrice 16×16 disponiamo le 5 matrici 3×3 lungo la diagonale, occupando solamente il 17.57% del totale della matrice, mentre occupiamo solo il 5.86% dell'intera matrice per i vettori. Questo, oltre a produrre uno spreco di memoria in fase di allocazione, fa si che molti Tensor Core lavorino su dati che non sono utili ai fini del calcolo dell'algoritmo.

Nella matrice 16×16 agiscono 16 Tensor Core Unit: questo significa che solamente 4 dei Tensor Core Unit effettuano calcoli utili per il problema (ovvero i quattro che operano lungo la diagonale) mentre i restanti 12 non effettuano calcoli utili, quindi l'utilizzo di Tensor Core Unit utile è inferiore al 25%. Con questa analisi si spiega come questo modo di operare non risulti efficiente.

Analisi soluzione 3

Quest'ultima soluzione rappresenta quella che meglio si comporta sia in termini di occupazione di memoria che di prestazioni. Il kernel è stato eseguito con un blocco di dimensioni pari a 1024 e una griglia di dimensioni pari a 10. Le prestazioni ottenute con questo metodo moltiplicando 2560 matrici sono le seguenti:

- Versione CUDA: $23.86\mu s$, oppure $\sim 9.92ns$/matrice.
- Versione Tensor Core: $13.57\mu s$, oppure $\sim 5.48ns$/matrice.

Come si osserva dai tempi ottenuti la versione con Tensor Core produce un significativo miglioramento delle prestazioni. Il vantaggio di questo metodo è dovuto al fatto che con una sola moltiplicazione di due matrici 16×16 si ottengono 8 vettori in forma complessa. In questo caso l'uso di Tensor Core produce uno speedup di circa il doppio.

In figura 3.14 vengono mostrati i tempi impiegati dalle soluzioni appena descritte; per una migliore leggibilità del grafico si è esclusa la soluzione 1 in quanto non significativa sia in termini di efficienza che di prestazioni.

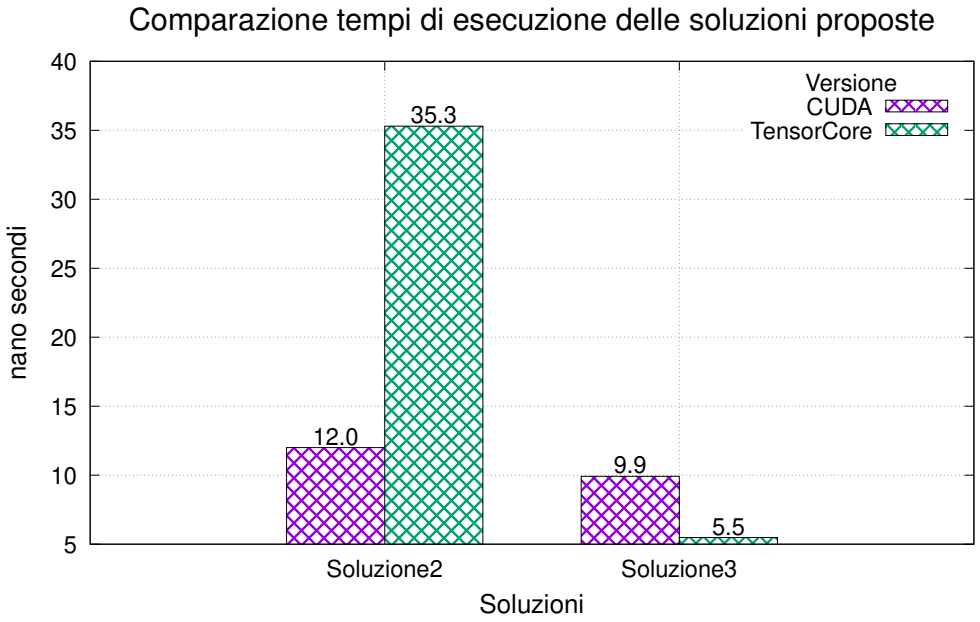


Figura 3.14: Analisi comparativa dei tempi di esecuzione delle soluzioni proposte per l'ottimizzazione dell'operatore di Dirac.

Conclusioni e sviluppi futuri

In questa tesi si è analizzata l'architettura dei Tensor Core e come questi ultimi possono essere utilizzati nell'operatore di Dirac. Nella prima parte del lavoro si è analizzato come le GPU siano diventate uno strumento rilevante per il calcolo parallelo. Si è quindi analizzata nel dettaglio l'architettura interna della GPU in particolare ai Tensor Core, studiando il loro funzionamento e i vantaggi che possono offrire nella moltiplicazione di matrici. Nell'ultima parte, descritta nel capitolo tre, si è studiato il funzionamento dell'operatore di Dirac contestualizzandone un possibile utilizzo nelle simulazioni LQCD; si è poi descritta la struttura dati utilizzata per rappresentare i collegamenti e i fermioni, enfatizzandone i vantaggi che possono offrire in termini di efficienza di calcolo dell'operatore.

Come contributo per un possibile miglioramento delle prestazioni dell'operatore di Dirac sono state formulate alcune soluzioni che prevedono l'uso dei Tensor Core. Di queste soluzioni sono state analizzate nel dettaglio le prestazioni che offrono e si è identificata in particolare una soluzione che, applicata all'operatore di Dirac, migliora di circa il doppio le prestazioni misurate con i soli CUDA Core.

Se da un lato i Tensor Core possono offrire un modo per aumentare le prestazioni dell'algoritmo di Dirac, il fatto che operino con tipi di dato a mezza precisione (half/FP16) può portare ad una eccessiva approssimazione dei valori ottenuti.

A fronte del lavoro svolto si è appreso che i Tensor Core nascono per accelerare algoritmi di machine learning e non principalmente per l'High Performance Computing. Questo rende l'architettura molto specifica per un tipo di operazione con vincoli molto rigidi: il lato delle matrici deve avere come dimensione un multiplo di otto, oltre a rappresentare i dati a mezza precisione. Questo rende molto difficile adattare i problemi di High Performance Computing a questi vincoli, infatti in questo caso è stato necessario disporre i dati nelle matrici in modo piuttosto complesso, che però ha portato a dei miglioramenti nelle prestazioni dell'operatore di Dirac.

Un possibile sviluppo del progetto è quello di modificare leggermente la struttura dati per consentire la creazione di un'area attorno al dominio in modo da ottenere un contorno ciclico affinché non sia necessario controllare che il sito corrente sia di bordo, potendo leggere direttamente i dati in memoria, aumentando ulteriormente le prestazioni dell'algoritmo. Un altro possibile sviluppo consiste nell'analizzare l'impatto che la rappresentazione a mezza precisione dei dati ha nella simulazione, ad esempio calcolando la massima variazione percentuale tra i dati ottenuti con rappresentazione a doppia precisione e i dati ottenuti con rappresentazione a mezza precisione.

Bibliografia

[1] Maurice Herlihy e Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[2] John L Hennessy e David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[3] *NVIDIA GeForce 256*. https://www.nvidia.co.uk/page/geforce256.

[4] *CUDA Toolkit*. https://developer.nvidia.com/cuda-zone.

[5] *OpenCL: the open standard for parallel programming of heterogeneous systems*. https://www.khronos.org/opencl/.

[6] Gordon E. Moore. «Cramming more components onto integrated circuits». In: *Electronics* 38.8 (apr. 1965).

[7] J. M. Shalf e R. Leland. «Computing beyond Moore's Law». In: *Computer* 48.12 (dic. 2015), pp. 14–23. ISSN: 1558-0814. DOI: 10.1109/MC.2015.374.

[8] *OpenACC: more science. Less programming*. https://www.openacc.org/.

[9] *Introduction to GPUs: CUDA*. https://nyu-cds.github.io/python-gpu/02-cuda/.

[10] S. Markidis et al. «NVIDIA Tensor Core Programmability, Performance Precision». In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.

[11] A. Abdelfattah, S. Tomov e J. Dongarra. «Fast Batched Matrix Multiplication for Small Sizes Using Half-Precision Arithmetic on GPUs». In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 111–122. DOI: 10.1109/IPDPS.2019.00022.

[12] Zhe Jia et al. «Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking». In: *CoRR* abs/1804.06826 (2018). arXiv: 1804.06826. URL: http://arxiv.org/abs/1804.06826.

[13] Claudio Bonati et al. «Portable LQCD Monte Carlo code using OpenACC». In: *EPJ Web of Conferences* 175 (gen. 2018), p. 09008. DOI: 10.1051/epjconf/201817509008.

Ringraziamenti

Ringrazio la mia famiglia per il sostegno e il supporto offertomi durante il percorso di studi. Desidero ringraziare il professore Moreno Marzolla per la disponibilità, la professionalità e per avermi seguito durante lo sviluppo della tesi. Un grazie è rivolto a Enrico Calore e Fabio Schifano che mi hanno concesso l'opportunità di poter lavorare a questa tesi fornendo strumenti, consigli e suggerimenti. Ringrazio tutti coloro che hanno contribuito alla stesura di questa tesi offrendo consigli, osservazioni e critiche. Inoltre ringrazio amici e colleghi che hanno reso questa esperienza più piacevole oltre a sostenermi nei momenti più difficili. Desidero ringraziare l'amica e collega Martina Cavallucci per il sostegno e l'aiuto offerto durante tutto il percorso affrontato. Infine ringrazio i colleghi del gruppo *CeSeNA Security* che hanno contribuito a fornirmi conoscenze e metodologie che si sono rivelate fondamentali per gli studi, in particolare ringrazio Fabrizio Margotta il quale mi ha supportato e aiutato in ogni momento del percorso universitario.