ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA E DELLE TELECOMUNICAZIONI PER LO SVILUPPO SOSTENIBILE

TITOLO DELLA TESI

PROGETTO E IMPLEMENTAZIONE DI UN DECODIFICATORE LDPC SU ARCHITETTURA GPU

Tesi in

SISTEMI DI TELECOMUNICAZIONI

Relatore
Alessandro Vanelli Coralli

Co-Relatori:
Stefano Andrenacci
Giorgio Calarco

Presentata da
Luca Feltrin

Sessione III
Anno Accademico 2012/2013

Sommario

INTRODUZIONE **1**

CODICI LDPC **3**

2.1 INTRODUZIONE 3
2.1.1 Definizioni e parametri caratteristici 3
2.1.2 Rappresentazione della matrice di parità 6
2.2 ALGORITMI DI CODIFICA/DECODIFICA 7
2.2.1 Codici Repeat-Accumulate 8
2.3 ALGORITMI DI DECISIONE 13
2.3.1 SPA 14
2.3.2 MSA 17
2.3.3 Considerazioni per l'implementazione di MSA su sistemi many-core 20
2.4 CODICI LDPC NELLO STANDARD DVB-S2 23

CUDA **27**

3.1 INTRODUZIONE ALLA PROGRAMMAZIONE GPGPU 27
3.2 ELEMENTI DI CUDA 28
3.2.1 L'Hardware 29
3.2.2 Modello di esecuzione SIMD e Branch Divergence 31
3.2.3 Organizzazione dei thread 32
3.2.4 Memorie 34
3.3 TECNICHE DI OTTIMIZZAZIONE 36
3.3.1 Occupancy 36
3.3.2 Coalescenza 38
3.3.3 Evitare le Branch 39
3.3.4 Memorie speciali e trasferimenti da Host a Device 40

PROGETTO E IMPLEMENTAZIONE **43**

4.1 INTRODUZIONE 43
4.2 STRUTTURA GENERALE 44
4.2.1 Rappresentazione dei codici LDPC 46
4.2.2 Schema a blocchi e filosofia 50
4.2.3 Le API 52
4.3 I BLOCCHI DI ELABORAZIONE IN DETTAGLIO 59
4.3.1 Generatore di bit Casuali 59
4.3.2 Encoder 59
4.3.3 Simulatore di canale 60
4.3.4 Decoder 61
4.3.5 Calcolatore di Bit Error Rate 62

RISULTATI E MISURE **65**

5.1 INTRODUZIONE E TOOL UTILIZZATI 65
5.2 GLI STRUMENTI UTILIZZATI 65
5.3 FIGURE DI EFFICIENZA 67

5.4 STRATEGIE USATE E LORO VALIDAZIONE 68
5.4.1 Coalescenza *68*
5.4.2 Memoria Texture e Surface *71*
5.5 TUNING DEI PARAMETRI 72
5.5.1 Numero di parole elaborate in parallelo *72*
5.5.2 Configurazione dei thread *72*
5.5.3 Numero di Iterazioni del Decoder *74*
5.6 PRESTAZIONI FINALI E LIMITAZIONI 76
5.6.1 Confronto con lavori precedenti *77*

CONCLUSIONI 81

6.1 CONSIDERAZIONI SUI RISULTATI OTTENUTI 81
6.2 LAVORI FUTURI 81

BIBLIOGRAFIA I

INDICE DELLE FIGURE II

INDICE DELLE TABELLE III

INDICE DEI CODICI III

ALLEGATI V

7.1 SOLUZIONE VISUAL STUDIO 2010 V
7.1.1 Progetti realizzati .... *v*
7.1.2 Organizzazione dei File .... *vi*
7.1.3 Come creare un nuovo progetto .... *vi*
7.2 DESCRIZIONE DEI KERNEL VIII
7.2.1 Kernel per la codifica .... *viii*
7.2.2 Kernel per la decodifica .... *ix*
7.2.3 Kernel per altre elaborazioni .... *xiii*
7.3 CONFRONTO TRA LE PRESTAZIONI DEL DECODER VARIANDO IL NUMERO DI PAROLE DECODIFICATE IN PARALLELO . XV
7.4 CONFRONTO TRA LE PRESTAZIONI DEL DECODER VARIANDO LA QUANTITÀ DI L1 CACHE XVI

CAPITOLO 1

Introduzione

Negli ultimi tempi il mondo delle telecomunicazioni ha visto l'evoluzione degli standard per la trasmissione di dati multimediali in digitale, con l'obiettivo di sfruttare al meglio lo spettro radio e di garantire qualità di servizio sempre maggiori.

Questa evoluzione prosegue, tuttora, di pari passo con quella dei sistemi di calcolo, che permettono l'implementazione di algoritmi e protocolli sempre più complicati, garantendo comunque un throughput elevato[1].

Esistono varie soluzioni per l'implementazione di algoritmi particolarmente onerosi: l'implementazione su logiche programmabili (come gli FPGA), e in seguito su circuiti integrati specializzati (ASIC), è agevolata dai numerosi strumenti presenti oggi sul mercato. Tuttavia l'utilizzo di queste tecniche non permette al sistema di essere flessibile; inoltre il tempo necessario per la progettazione di questo è spesso proibitiva, rendendo poco conveniente questa strada.

Lo sviluppo di sistemi multi-core ha reso possibile un'implementazione, con prestazioni vicine a quelle ottenute su ASIC, flessibile e relativamente semplice, in quanto realizzata utilizzando linguaggi di programmazione con un più alto livello di astrazione.

Questo tema è già stato affrontato ampiamente da molti autori, a partire dalle tecniche per implementare al meglio un qualsiasi algoritmo in ambito Software Defined Radio su sistemi multi-core [1], fino agli studi di G. Falcao sui decodificatori LDPC [2] [3] [4] che, oltre a studiare le proprietà degli algoritmi, in questo caso la decodifica LDPC, ha cercato delle soluzioni efficienti per la loro implementazione su vari

[1] Il throughput è definito come la quantità di dati elaborati dal sistema per unità di tempo

tipi di architettura, analizzandone ampiamente tutti gli aspetti salienti come i colli di bottiglia.

Allo stato dell'arte i decoder LDPC sviluppati hanno prestazioni eccellenti, nell'ordine delle decine di Mbps[2]; questi sono stati progettati unicamente per avere le migliori prestazioni possibili, risultando in un'implementazione statica.

Per quanto le prestazioni siano un requisito fondamentale in questo tipo di applicazioni, un sistema integrato dovrà avere la possibilità di modificare il proprio funzionamento durante l'esecuzione stessa.

In questo lavoro si esplora la possibilità di realizzare un sistema più flessibile sacrificando, in parte, le prestazioni; si è cercato, inoltre, di impostare il software in maniera modulare in modo che, in futuro, si possano aggiungere facilmente altre funzionalità fino alla realizzazione un apparato di ricezione DVB-S2 completo.

Il decodificatore sviluppato, nonostante implementi lo stesso algoritmo usato nei lavori degli altri autori, ha una struttura molto diversa dal punto di vista della scrittura del codice: esso gode di una più alta flessibilità a discapito delle prestazioni che sono inferiori anche se comparabili con lo stato dell'arte.

In futuro sarà necessario valutare la realizzazione di un decoder che sia un compromesso tra queste diverse filosofie.

Nei primi capitoli sono introdotti i concetti teorici fondamentali circa gli algoritmi implementati e l'ambiente di sviluppo utilizzato. In particolare, partendo dall'algoritmo di decodifica presentato in vari articoli, tra cui [5], in che modo questo è stato modificato per essere più veloce nell'esecuzione sull'hardware in esame.

In seguito, è analizzata la filosofia e il funzionamento di base del sistema implementato, concentrandosi sulle strategie utilizzate per renderlo efficiente.

Nel capitolo quinto, sono misurate e analizzate le prestazioni del sistema dimostrandone l'efficienza, grazie all'ausilio di alcuni strumenti messi a disposizione all'interno del framework utilizzato.

Infine, nel capitolo conclusivo, dopo aver confermato la validità del progetto, sono proposte migliorie e alcune considerazioni riguardo a possibili studi futuri, insieme con delle linee guida da seguire per realizzare, infine, un terminale completo.

[2] In questo lavoro si è utilizzato hardware con prestazioni molto inferiori a quelle usati negli altri lavori, l'unico confronto reale è stato eseguito mediante l'esecuzione del decoder di uno degli autori sull'hardware a disposizione

CAPITOLO 2

Codici LDPC

2.1 Introduzione

In questo capitolo sarà presentata una panoramica sulla teoria dei codici lineari e LDPC, saranno quindi presentate alcune sottoclassi di codici ed alcuni algoritmi di codifica/decodifica con riferimento alla sottoclasse di appartenenza dei codici dello standard DVB-S2.

I codici di canale LDPC (Low Density Parity Check) furono inventati da Robert G. Gallager nel 1960, e poi dimenticati per trent'anni a causa dell'elevata capacità di calcolo richiesta per la decodifica.

Solo nel 1993 furono "riscoperti" in parallelo ai Turbocodici, che offrono prestazioni simili, per applicazioni satellitari e in spazio profondo.

Nel 2003 un codice LDPC è stato introdotto per implementare la codifica interna nello standard DVB-S2 e, negli anni successivi, sempre più applicazioni hanno adottato questa classe di codici.

I codici LDPC sotto certe condizioni offrono prestazioni cosiddette "Capacity Approaching" ovvero che tendono ad avvicinarsi al limite teorico di Shannon.

2.1.1 Definizioni e parametri caratteristici

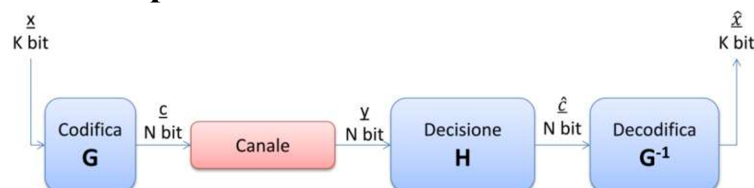


Figura 2.1: Schema a blocchi di riferimento

I codici LDPC appartengono alla classe dei codici lineari a blocco, pertanto godono di certe proprietà e sono definiti da alcune matrici.

Il flusso di bit che si vuole codificare è suddiviso in blocchi da K bit, ognuno indipendente dagli altri; assumendo che un singolo bit sia rappresentato come uno scalare appartenente a $GF(2)$[3] il blocco di bit è un vettore colonna $\underline{x}$ appartenente a $GF(2)^K$.

Ciascun blocco di bit è codificato mediante una moltiplicazione matriciale del tipo[4]

$$\underline{c} = \boldsymbol{G}\underline{x} \tag{2.1}$$

con $\boldsymbol{G}$ matrice N x K (detta matrice generatrice), $\underline{c}$ vettore colonna lungo N (detto parola di codice o codeword) con $N \geq K$.

Il rapporto $R = {}^{K}\!/_{N} = 1 - {}^{M}\!/_{N}$ è detto "rate" del codice ed è un'indicazione della ridondanza aggiunta ai bit trasmessi, a sua volta questa dà un'indicazione di quanto i dati siano protetti dal rumore: infatti più la rate è bassa più bit vengono aggiunti alla trasmissione più è bassa la probabilità di errore di decodifica a parità di rapporto segnale rumore.

Dalla matrice generatrice è possibile ricavare una seconda matrice $M \times N$ con $M=N-K$, detta matrice di controllo parità **H**, tale per cui

$$\boldsymbol{H}\underline{c} = \underline{0} \Rightarrow \boldsymbol{HG} = \boldsymbol{0} \Rightarrow \boldsymbol{H} \text{ è il complemento ortogonale di } \boldsymbol{G} \tag{2.2}$$

La formula (2.2) definisce un sistema di equazioni che le parole di codice devono soddisfare sempre. Se per N bit ricevuti tutte le equazioni sono soddisfatte, ovvero se la moltiplicazione matriciale $\boldsymbol{H}\underline{c}$ dà come risultato il vettore $\underline{0}$, allora gli N bit costituiscono una parola di codice; viceversa il risultato è detto sindrome dell'errore.

La decodifica, in caso di canale rumoroso, si basa sulla matrice $\boldsymbol{H}$: infatti una volta ricostruita la parola di codice effettivamente trasmessa è facile risalire ai bit di informazione iniziali, applicando la trasformazione inversa di $\boldsymbol{G}$.

[3] GF(2) è un campo finito di Galois definito dall'insieme di valori {0,1}, dalla somma $\oplus$ analoga all'operazione booleana XOR e dalla moltiplicazione analoga all'operazione booleana AND

[4] Le matrici vengono indicate in grassetto, i vettori sono sottolineati e salvo ulteriori indicazioni sono sempre vettori colonna

Nel caso di codici LDPC la matrice $\boldsymbol{H}$ è sparsa, ovvero il numero di '1' nella matrice è molto minore del numero di '0': "Low Density" è riferito appunto al numero di '1'.

Questa proprietà della matrice $\boldsymbol{H}$ offre come principale vantaggio quello di diminuire la complessità dell'algoritmo di decodifica che, come vedremo in seguito, è proporzionale al numero di '1'; in questo modo si possono utilizzare codici di grandi dimensioni, ovvero con N molto grande, che si avvicinano maggiormente al limite teorico di Shannon.

È da notare che, pur essendo $\boldsymbol{H}$ sparsa, $\boldsymbol{G}$ non lo è necessariamente. In caso di matrici grandi ciò costituisce un problema a causa del tempo richiesto per la moltiplicazione tra i bit di informazione e la matrice $\boldsymbol{G}$. In tali casi è preferibile usare algoritmi di codifica diversi.

Si può definire 'E' il numero di simboli '1' e la densità di '1' nel modo seguente

$$\rho = \frac{E}{N \cdot M} \tag{2.3}$$

Se i primi K bit di una parola di codice corrispondono ai bit di informazione originali, il codice si dice sistematico. In questo caso si semplifica notevolmente l'estrazione di bit di informazione una volta completata la decisione in quanto essi sono già presenti senza bisogno di ulteriori calcoli.

Per i codici sistematici le matrici $\boldsymbol{G}$ e $\boldsymbol{H}$ assumono una forma del tipo

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{I_K} \\ \boldsymbol{P} \end{pmatrix}, \qquad \boldsymbol{H} = (\boldsymbol{P} \quad \boldsymbol{I_M}), \qquad \boldsymbol{P} \text{ è } matrice\ M \times K \tag{2.4}$$

Oppure

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{P} \\ \boldsymbol{I_K} \end{pmatrix}, \qquad \boldsymbol{H} = (\boldsymbol{I_M} \quad \boldsymbol{P}), \qquad \boldsymbol{P} \text{ è } matrice\ M \times K \tag{2.5}$$

Un importante parametro per tutti i tipi di codici lineari è rappresentato dalla distanza minima d_{min} che indica la minima distanza di Hamming[5] tra due parole di codice qualsiasi.
Si può dimostrare che ogni codice a blocco lineare ha la capacità di rilevare $d_{min} - 1$ errori e di correggere $\left\lceil \frac{d_{min}}{2} \right\rceil - 1$ errori.

[5] La distanza di Hamming tra due vettori di bit è definita come il numero di bit per i quali essi differiscono

Si possono definire anche due sequenze di numeri, spesso considerate come coefficienti di due polinomi che indicano il peso di colonne e righe della matrice ***H***, dove con peso si intende il numero di '1' in quella colonna o riga:

$$w_c(m) = \text{numero di '1' nella riga m } \forall m = 0,1,\dots,M-1 \tag{2.6}$$

$$w_b(n) = \text{numero di '1' nella colonna n } \forall n = 0,1,\dots,N-1 \tag{2.7}$$

Quando i pesi sono costanti per tutte le righe e tutte le colonne il codice viene detto regolare e in genere si indicano semplicemente w_c e w_b come parametro singolo. In caso contrario il codice è irregolare.

Da notare che deve essere sempre verificato

$$\sum_{n=0}^{N-1} w_b(n) = \sum_{m=0}^{M-1} w_c(m) = E \tag{2.8}$$

che nel caso di codici regolari si riduce a

$$N w_b = M w_c \tag{2.9}$$

Nel caso di codici regolari valgono anche le seguenti relazioni

$$\begin{aligned} w_b &= \rho M \\ w_c &= \rho N \end{aligned} \tag{2.10}$$

$$R = 1 - {}^{M}\!/_{N} = 1 - {}^{w_c}\!/_{w_b} \tag{2.11}$$

2.1.2 Rappresentazione della matrice di parità

Nell'ambito dei codici LDPC la matrice di parità viene in genere rappresentata con un grafo bipartito, detto grafo di Tanner.

Il grafo è formato da *N* nodi detti nodi variabile (o nodi bit) che corrispondono ai bit della parola di codice che si vuole decodificare, e da *M* nodi, detti nodi di controllo, che corrispondono alle equazioni di parità definite dalla matrice ***H***.

Tra il generico nodo variabile *n* e il nodo di controllo *m* vi è un collegamento se $H_{mn} = 1$

Dato che la matrice ***H*** è sparsa anche il grafo risulterà sparso, ovvero avrà pochi collegamenti.

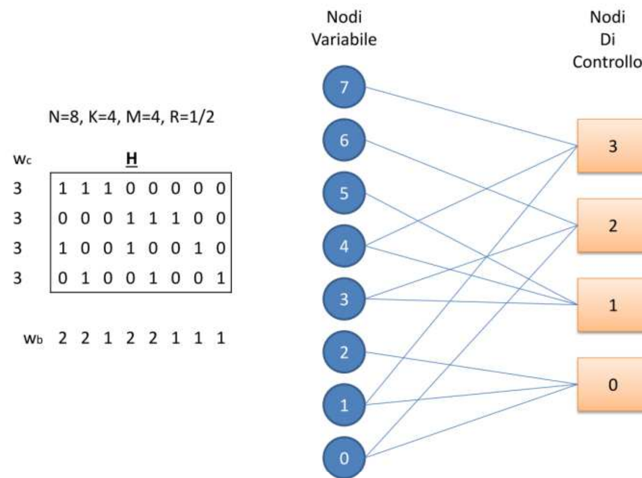


Figura 2.2: Codice di esempio: parametri caratteristici e relativo grafo di Tanner

2.2 Algoritmi di Codifica/Decodifica

L'algoritmo di codifica più semplice è rappresentato dal prodotto del vettore di bit di informazione con la matrice ***G*** del codice; la decodifica è ottenuta mediante l'operazione inversa, ovvero $\underline{x} = \boldsymbol{G}^{-1}\underline{c}$.

Come già descritto precedentemente il semplice prodotto ha una complessità accettabile solo per codici di piccole dimensioni; nei casi di nostro interesse la matrice ***G***, che non è sparsa, è troppo grande: mentre la decodifica è comunque possibile, la codifica non lo è, e sono necessari algoritmi più semplici oppure codici studiati ad-hoc per rendere la codifica più semplice.

Inoltre per memorizzare una matrice binaria così grande è richiesta una grande quantità di memoria.

Per esempio la matrice ***G*** del codice LDPC usato nello standard DVB-S2 nel caso di trama normale e rate *1/2* è 64800×32400, nell'ipotesi in cui ogni elemento venga memorizzato con un bit di memoria, occuperebbe circa 250 MB.

Questo problema è solo lievemente mitigato nel caso di codici sistematici che offrono il vantaggio di annullare la complessità dell'estrazione dei bit di informazione dalla parola di codice decodificata.

Ciononostante è comunque possibile ricorrere alla moltiplicazione matriciale in applicazioni per le quali *N* e *K* risultino limitati in valore.

In caso si conosca solo $\boldsymbol{H}$ è possibile risalire a $\boldsymbol{G}$ eseguendo l'algoritmo di eliminazione di Gauss-Jordan su $\boldsymbol{H}$ fino ad arrivare a una forma del tipo $(\boldsymbol{I_M} \quad \boldsymbol{P})$: a questo punto $\boldsymbol{G} = \begin{pmatrix} \boldsymbol{P} \\ \boldsymbol{I_K} \end{pmatrix}$ che è in forma sistematica.

Fortunatamente esistono alcune sottoclassi di codici LDPC che hanno proprietà tali per cui l'algoritmo di codifica si può semplificare notevolmente: in alcuni casi si riesce ad avere addirittura una complessità lineare e una codifica in tempi piuttosto brevi.
Come vedremo i codici LDPC dello standard DVB-S2 appartengono a una di queste sottoclassi.

2.2.1 Codici Repeat-Accumulate

Per questi codici (**RA**) lo schema del codificatore è rappresentato in Figura 2.3.

Ogni bit di informazione viene ripetuto q volte, ottenendo un frame lungo qK. A questo frame è quindi applicata una permutazione casuale di dimensione fissa ai bit e in seguito questi vengono elaborati da un accumulatore differenziale[6] con funzione di trasferimento $\frac{1}{1+D}$.

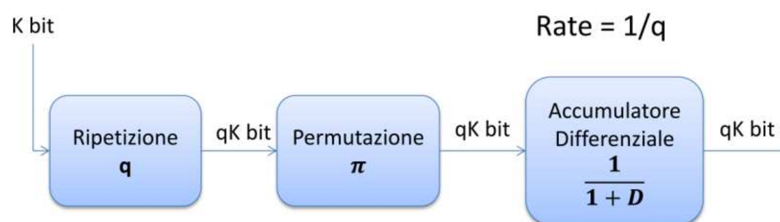


Figura 2.3: Diagramma a blocchi di un encoder RA

I principali difetti dei codici RA sono gli elevati valori di probabilità di errore per alti rapporti segnale-rumore e la scarsa possibilità di scelta della rate: dato che q è un numero intero le rate possibili sono solo $1/2$, $1/3$, $1/4$ eccetera.

Per ovviare a queste limitazioni sono stati inventati i codici RA irregolari (**IRA**): lo schema è simile a quello dei codici RA ma, a differenza di questi, i bit vengono ripetuti un numero variabile di volte e possono anche essere sommati tra di loro prima di essere mandati all'accumulatore differenziale.

[6] Un accumulatore differenziale inverte l'uscita quando in ingresso è presente un '1': 0**1**000**11**0 -> 0**1**111**0**1**1**

La definizione originale dei codici IRA in [1] consiste in un grafo di Tanner con una struttura particolare raffigurata in Figura 2.4. Vedremo in seguito che le due definizioni qui date coincidono.

Vi sono K nodi variabile a sinistra detti nodi informazione, R nodi di controllo al centro, e altri R nodi variabile all'estrema destra detti nodi di parità, collegati ai nodi di controllo a zigzag.

Il codice è definito con una sequenza di numeri del tipo $(f_1, f_2, \ldots, f_J | a)$: il significato degli f_i è che ci sono Kf_i nodi informazione che sono collegati ad i nodi di controllo, mentre a significa che ogni nodo di controllo è collegato ad a nodi informazione.

Si può pensare a questo grafo come a uno schema a blocchi che partendo dall'input a sinistra genera, andando verso destra, i bit di parità.

In fase di codifica i bit di informazione vengono assegnati ai nodi informazione; i nodi di controllo impongono la condizione che la somma dei nodi collegati ad essi deve essere 0: in questo modo i nodi parità a destra assumono un certo valore.

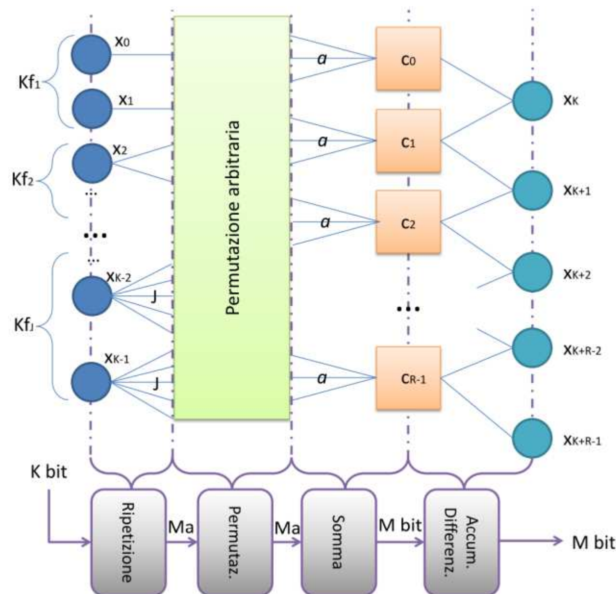


Figura 2.4: Grafo di Tanner di un codice IRA e schema a blocchi equivalente

Questo schema può anche essere utilizzato per generare i soli bit di parità (R=M): in questo caso il codice sarà sistematico, per semplicità d'ora in poi sarà considerato solo questo caso.

È facile vedere che il grafo in Figura 2.4 è una versione con i nodi disposti in maniera differente di un generico grafo di Tanner.

La matrice $\boldsymbol{H}$ associata al grafo di un codice IRA avrà dimensioni $R \times N = M \times N$ ossia

$$\boldsymbol{H} = \left(\begin{array}{c|ccccc} & 1 & & & & \\ & 1 & 1 & & & \\ \boldsymbol{G_1} & & 1 & \ddots & & \\ & & & \ddots & 1 & \\ & & & & 1 & 1 \end{array} \right) \quad (2.12)$$

matrice sparsa con righe di peso $a+2$, tranne la prima che ha peso $a+1$ (in tutto la matrice contiene $M(a+2)-1$ '1').

In precedenza sono state date due definizioni di codice IRA. Sarà ripresa ora la definizione basata sul metodo di codifica come successione di blocchi di elaborazione dei bit e vedremo come la matrice di parità risultante coincida con quella appena ottenuta dalla definizione originale.

Un encoder IRA è formato da quattro blocchi di elaborazione. Ciascun blocco può essere pensato come un codice a sé stante e quindi con la propria matrice generatrice associata.

- Un codice a ripetizione variabile con una matrice $aM \times K$ del tipo

$$\boldsymbol{G_r} = \begin{pmatrix} 1 & & & \\ \vdots & & & \\ 1 & & & \\ & 1 & & \\ & \vdots & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ & & & \vdots \\ & & & 1 \end{pmatrix} \quad (2.13)$$

  dove ogni gruppo verticale di '1' ha una lunghezza variabile, in particolare in riferimento alla notazione descritta precedentemente, ci sono Kf_i colonne di peso i.
- Una permutazione $aM \times aM$ con generatrice $\boldsymbol{\pi}$ che consiste in una versione con le colonne permutate della matrice identità $\boldsymbol{I_{aM}}$
- Un'ulteriore matrice $M \times aM$che somma tra loro gruppi di a bit adiacenti del tipo

$$\boldsymbol{G_a} = \begin{pmatrix} 1\cdots 1 & & & \\ & 1\cdots 1 & & \\ & & \ddots & \\ & & & 1\cdots 1 \end{pmatrix} \tag{2.14}$$

ogni riga è di peso a

- Un accumulatore differenziale, che si può dimostrare ha matrice generatrice $M \times M$ triangolare del tipo

$$\boldsymbol{D} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 1 & 1 & & \vdots \\ \vdots & & \ddots & 0 \\ 1 & \dots & 1 & 1 \end{pmatrix} \tag{2.15}$$

  Si può dimostrare inoltre che la sua inversa è una matrice con doppia diagonale:

$$\boldsymbol{D^{-1}} = \begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ & 1 & \ddots & & \\ & & \ddots & 1 & \\ & & & 1 & 1 \end{pmatrix} \tag{2.16}$$

Con riferimento ai soli codici sistematici, dove la catena di blocchi serve solo per produrre i bit di parità che vengono poi appesi ai bit di informazione, si ha che

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{I_K} \\ \boldsymbol{P} \end{pmatrix} \; con\; \boldsymbol{P} = \boldsymbol{D G_a \pi G_r} = \boldsymbol{D G_1} \tag{2.17}$$

Da notare che $\mathbf{G_1} = \mathbf{G_a \pi G_r}$ è una matrice sparsa, precisamente formata, a meno di casi degeneri, da aM '1'.

Supponendo che la matrice parità $\boldsymbol{H}$ sia formata da due sottomatrici $\boldsymbol{H} = (\boldsymbol{H_1}|\boldsymbol{H_2})$ rispettivamente $M \times K$ e $M \times M$, deve essere soddisfatto $\mathbf{HG}=\mathbf{0}$. Si ha che

$$\boldsymbol{HG} = (\boldsymbol{H_1}|\boldsymbol{H_2})\begin{pmatrix} \boldsymbol{I_K} \\ \boldsymbol{P} \end{pmatrix} = \boldsymbol{H_1 I_K} + \boldsymbol{H_2 P} = \boldsymbol{H_1} + \boldsymbol{H_2 P} = \boldsymbol{0}$$

$$\boldsymbol{H_1} = -\boldsymbol{H_2 P} = \boldsymbol{H_2 P} \;\left(in\; GF(2)\right)$$

$$\boldsymbol{H_2^{-1} H_1} = \boldsymbol{H_2^{-1} H_2 P} = \boldsymbol{P}$$

Dato che $\boldsymbol{P}$ è anche uguale a $\boldsymbol{DG_1}$,

$$\boldsymbol{H_2^{-1} H_1} = \boldsymbol{P} = \boldsymbol{DG_1}$$

$$\boldsymbol{H_1} = \boldsymbol{G_1} \quad \text{e} \quad \boldsymbol{H_2} = \boldsymbol{D^{-1}}$$

e quindi la matrice di parità del codice risulta essere

$$\boldsymbol{H} = (\boldsymbol{G_1}|\boldsymbol{D^{-1}}) = \left(\begin{array}{c|ccccc} & 1 & & & & \\ & 1 & 1 & & & \\ \boldsymbol{G_1} & & 1 & \ddots & & \\ & & & \ddots & 1 & \\ & & & & 1 & 1 \end{array} \right) \tag{2.18}$$

che è sparsa. In particolare è facile verificare che è la stessa matrice ottenuta partendo dalla definizione originale di codice IRA.

Infine esiste una ulteriore estensione dei codici IRA , i codici **eIRA**, dove la "e" sta per "extended".

Questi codici differiscono dagli IRA per il fatto che la matrice $\boldsymbol{H}$ non presenta una doppia diagonale sulla destra, ma una generica matrice con colonne di peso 2 tranne l'ultima che è di peso 1. Questo significa che il blocco finale non sarà più un accumulatore differenziale bensì una qualche funzione indefinita a priori.

Si noti che i codici RA sono un caso particolare dei codici IRA che a loro volta sono un caso particolare dei codici eIRA.

Infine, per questa sottoclasse di codici, l'algoritmo può essere diviso in due fasi. Nella prima fase si esegue la ripetizione dei bit, la loro permutazione e la loro somma in gruppi, semplicemente facendo la moltiplicazione matriciale tra i bit di informazione e la matrice $\boldsymbol{G_1}$. Questo prodotto matriciale non è particolarmente oneroso perché essendo $\boldsymbol{H}$ sparsa e visto che $\boldsymbol{G_1}$ compare direttamente in questa, allora sarà anch'essa sparsa rendendo questa moltiplicazione poco complessa. Nella seconda fase i bit risultati devono passare per l'accumulatore differenziale, ovvero devono essere moltiplicati per **D**. Questa operazione si può fare con complessità lineare basandosi sul soddisfacimento delle equazioni di parità.

Si definiscono: p_i l'i-esimo bit di parità da calcolare, il $\underline{u}$ il vettore dei K bit di informazione da codificare e il vettore b lungo M tale che $b = \boldsymbol{G_1}u$, calcolato nella prima fase.
In questo modo le condizioni di parità si riducono a un sistema di equazioni composto da

$$p_i = b_i \tag{2.19}$$

dove i è l'indice dell'unica riga di **H** di peso $a+1$, e altre $M\text{-}1$ equazioni del tipo

$$p_j = p_k + b_j \tag{2.20}$$

risolvibili iterativamente, con p_k bit di parità calcolato al passo precedente, e j indice dell'equazione in esame nel passo corrente.

I bit di parità vengono quindi calcolati seguendo un certo ordine dipendente dalla matrice, l'encoder deve quindi conoscere questo ordine che può essere comunicato al programma tramite una sequenza di numeri oppure può essere ricavato dall'encoder stesso aumentando la complessità dell'algoritmo.

A causa dell'iteratività del procedimento, il computo dei bit di parità può essere svolto solo in maniera sequenziale.

2.3 Algoritmi di Decisione

Con algoritmi di decisione si intendono quegli algoritmi che, a partire dal segnale ricevuto, risalgono ai bit effettivamente trasmessi sul canale.

Per i codici LDPC la classe di algoritmi più utilizzata è quella Message Passing (**MP**), o a **passaggio di messaggi**.

Questi sono algoritmi iterativi dove a ogni iterazione un nodo del grafo di Tanner "trasmette" ai nodi vicini dei messaggi contenenti informazioni basate sul segnale ricevuto e sulle informazioni ricavate nei passi precedenti in modo da decidere, collaborando, quale parola di codice è stata inviata sul canale.

La sotto-classe più importante di questo tipo di algoritmi sono gli algoritmi Belief Propagation (**BP**) o a **propagazione di opinione**, in questi algoritmi i nodi si trasmettono la rispettiva "opinione" sul fatto che un bit ricevuto sia uno 0 o un 1, e decidono collaborativamente il valore del bit.

L'opinione è espressa sotto forma di probabilità che un bit sia uno 0 o un 1. In alternativa l'opinione viene espressa con altre grandezze collegate alla probabilità, come il rapporto di verosimiglianza o il suo logaritmo.

I messaggi vengono propagati solo lungo i collegamenti del grafo di Tanner della matrice ***H***.

Poiché per i codici LDPC questo grafo è sparso, anche il numero di messaggi è relativamente piccolo, rendendo vantaggiosa l'implementazione di questi algoritmi.

2.3.1 SPA

La prima versione dell'algoritmo di decodifica considerato si chiama **SPA** (Sum Product Algorithm). Le formule per il calcolo delle variabili interne durante l'elaborazione si ricavano direttamente dalle definizioni di probabilità.

In questa versione dell'algoritmo i messaggi scambiati contengono informazioni sull'opinione sotto forma di probabilità.

In particolare un singolo messaggio è formato dalle probabilità che un certo bit sia 0 e 1 condizionate a tutte le informazioni provenienti dagli altri nodi e dal canale.

Si definiscono le probabilità che l'n-esimo bit sia 1 o 0 condizionate al solo segnale ricevuto:

$$\Pr[x_n = 1 \mid y_n] = P_n \text{ e } \Pr[x_n = 0 | y_n] = 1 - P_n \tag{2.21}$$

Ogni nodo variabile tiene traccia dell'opinione che all'iterazione *l* il proprio bit sia 0 o 1 attraverso $Q_n^{(l)}(0)$ e $Q_n^{(l)}(1)$, queste sono le probabilità che il bit sia rispettivamente 0 o 1 condizionate al segnale ricevuto y_n e al soddisfacimento delle equazioni di parità.

Il nodo variabile inoltre manda ai suoi nodi di controllo connessi, un messaggio composto da $q_{nm}^{(l)}(0)$ e $q_{nm}^{(l)}(1)$, che hanno lo stesso significato dei $Q_n^{(l)}(0)$ e $Q_n^{(l)}(1)$ con l'unica differenza che quella probabilità non è condizionata dal soddisfacimento della equazione di parità del nodo destinatario.

La fase dell'algoritmo che calcola questi messaggi prende il nome di **processing verticale**.

L'algoritmo si inizializza ponendo $q_{nm}^{(0)}(0) = 1 - P_n$ e $q_{nm}^{(0)}(1) = P_n$, in quanto ovviamente all'inizio del calcolo le uniche informazioni che si hanno sono quelle in arrivo dal solo canale.

Si noti che nel caso di canale AWGN con modulazione BPSK

$$P_n = q_{nm}^{(0)}(1) = \frac{1}{1 + e^{-\frac{2y_n}{\sigma^2}}} \tag{2.22}$$

dove y_n è il segnale ricevuto.

Il generico nodo di controllo *m*, invece, invia al nodo variabile *n* un messaggio (composto da $r_{mn}^{(l)}(0)$ e $r_{mn}^{(l)}(1)$) che rappresenta l'opinione, all'iterazione *l,* che la

somma dei bit connessi a quel nodo di controllo, tranne il nodo variabile a cui si sta inviando il messaggio sia 0.

La fase dell'algoritmo che calcola questi messaggi prende il nome di **processing orizzontale**.
Un altro modo di esprimere questa definizione è che il messaggio $r_{mn}^{(l)}(0)$ rappresenta la probabilità che il bit x_n sia 0 dato che deve essere soddisfatta l'equazione di parità di indice m, ovvero[7]

$$\Pr[\mathrm{x_n} = 0 \mid \text{equazione di parità } m] = \Pr\left[\bigoplus_{n\prime \in N(m)/\mathrm{n}} x_{n\prime} = 0\right] \qquad (2.23)$$

Per trovare la formula usata in questa seconda fase dell'algoritmo si consideri che

$$\begin{aligned} \Pr[x_1 \oplus x_2 = 0] &= \Pr[x_1 = 0, x_2 = 0] + \Pr[x_1 = 1, x_2 = 1] \\ &= (1 - q_1)(1 - q_2) + q_1 q_2 \\ &= \frac{1}{2}[1 + (1 - 2q_1)(1 - 2q_2)] \end{aligned} \qquad (2.24)$$

dove q_1 e q_2 sono le probabilità che i nodi variabile 1 e 2 siano a 1. Aumentando il numero di addendi si ottiene

$$\begin{aligned} &\Pr[x_1 \oplus x_2 \oplus x_3 = 0] \\ &\quad = \frac{1}{2}[1 + (1 - 2\Pr[x_1 \oplus x_2 = 0])(1 - 2q_3)] \\ &\quad = \frac{1}{2}[1 + (1 - 2q_1)(1 - 2q_2)(1 - 2q_3)] \end{aligned} \qquad (2.25)$$

Quindi per induzione si può dire che

$$\Pr\left[\bigoplus_{n\prime \in N(m)/n} x_{n\prime} = 0\right] = \frac{1}{2} + \frac{1}{2} \prod_{n\prime \in N(m)/n} (1 - 2q_{n\prime}) \qquad (2.26)$$

Di conseguenza il messaggio che dal nodo di controllo va al nodo variabile sarà dato da

[7] La dicitura $\mathrm{N(m)/n}$ significa insieme dei nodi variabile connessi al nodo di controllo m tranne il nodo variabile di indice n

$$\mathrm{r}_{mn}^{(\mathrm{l})}(0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in N(m)/n} \left(1 - 2q_{n'm}^{(l-1)}(1)\right)$$
$$\mathrm{r}_{mn}^{(\mathrm{l})}(1) = 1 - \mathrm{r}_{\mathrm{mn}}^{(\mathrm{l})}(0) \tag{2.27}$$

Questi messaggi rappresentano le probabilità condizionate al solo soddisfacimento delle equazioni di parità.

Il nodo variabile può ora combinare tutte queste informazioni insieme alle informazioni provenienti dal canale facendo il prodotto di queste probabilità, ottenendo così la probabilità finale che un bit sia 0 o 1.

$$Q_n^{(l)}(0) = K_{nm}(1 - P_n) \prod_{m' \in M(n)} r_{m'n}^{(l)}(0)$$
$$Q_n^{(l)}(1) = K_{nm} P_n \prod_{m' \in M(n)} r_{m'n}^{(l)}(1) \tag{2.28}$$

Come detto precedentemente, i messaggi da inviare ai nodi di controllo si calcolano in modo simile senza includere le informazioni provenienti dal nodo destinatario, quindi i messaggi per la prossima iterazione si calcolano come

$$q_{nm}^{(l)}(0) = k_{nm}(1 - P_n) \prod_{m' \in M(n)/m} r_{m'n}^{(l)}(0)$$
$$q_{nm}^{(l)}(1) = k_{nm} P_n \prod_{m' \in M(n)/m} r_{m'n}^{(l)}(1) \tag{2.29}$$

In entrambi i casi i coefficienti K_{nm} e k_{nm} vanno scelti in modo che la somma delle due componenti sia 1.

L'algoritmo termina quando si raggiunge un numero massimo di iterazioni, oppure quando la sequenza di bit più verosimile in un certo momento soddisfa le equazioni di parità.

La sequenza di bit più verosimile si calcola a partire dai $Q_n^{(l)}(0)$ come

$$\widehat{x_n} = \begin{cases} 0 \;\; se \; Q_n^l(0) > \frac{1}{2} \\ 1 \; se \;\; Q_n^l(0) \leq \frac{1}{2} \end{cases} \tag{2.30}$$

Si può dimostrare che se il grafo del codice soddisfa certe proprietà, in particolare se non ha degli anelli chiusi di 4 collegamenti, l'algoritmo converge.

In definitiva questo algoritmo, che è quasi-ottimo, è caratterizzato da numerose moltiplicazioni che dal punto di vista computazionale sono operazioni onerose. Nella sezione seguente saranno illustrate alcune semplificazioni che permettono di semplificare il procedimento.

2.3.2 MSA

Per migliorare le prestazioni dell'algoritmo SPA si può notare per prima cosa come ogni probabilità è memorizzata in due versioni: la probabilità che un bit sia 0 e che sia 1.

È molto semplice ridurre questo "spreco" di memoria usando al posto delle probabilità i rapporti di verosimiglianza (likelihood ratios, **LR**) oppure il logaritmo del rapporto di verosimiglianza (log-likelihood ratio **LLR**). In questo modo si possono concentrare in un numero solo le stesse informazioni di prima.

Per una generica variabile aleatoria binaria *x* si definiscono

$$LR_x = \frac{\Pr[x=0]}{\Pr[x=1]} \text{ e } LLR_x = \ln\frac{\Pr[x=0]}{\Pr[x=1]} = Lx \text{ per brevità} \tag{2.31}$$

Esprimendo tutte le variabili dell'algoritmo come LLR le formule si modificano come descritto di seguito.

L'inizializzazione dell'algoritmo si ha con

$$Lq_{nm}^{(0)} = \ln\frac{1-P_n}{P_n} = -\frac{2y_n}{\sigma^2} \tag{2.32}$$

Partendo dalla formula (2.27) del processing orizzontale possiamo fare alcune manipolazioni

$$\mathrm{r}_{mn}^{(\mathrm{l})}(0) = \frac{1}{2} + \frac{1}{2}\prod_{n'\in N(m)/n}\left(1-2q_{n'm}^{(l-1)}(1)\right)$$

$$1-2\mathrm{r}_{mn}^{(\mathrm{l})}(1) = \prod_{n'\in N(m)/n}\left(1-2q_{n'm}^{(l-1)}(1)\right)$$

Considerando che

$$tanh\left(\frac{1}{2}Lp\right) = tanh\left[\frac{1}{2}ln\left(\frac{1-p}{p}\right)\right] = 1 - 2p$$

l'espressione precedente diventa

$$tanh\left(\frac{1}{2}Lr_{mn}^{(l)}\right) = \prod_{n\prime\in N(m)/n} tanh\left(\frac{1}{2}Lq_{n\prime m}^{(l-1)}\right)$$

Questa espressione è più complicata di quella dell'algoritmo SPA, ma in questo caso si può scomporre il rapporto di verosimiglianza del messaggio q nel modo seguente

$$Lq_{nm}^{(l)} = \alpha_{nm}\beta_{nm}$$
$$\alpha_{nm} = sign\left(Lq_{nm}^{(l)}\right)$$
$$\beta_{nm} = \left|Lq_{nm}^{(l)}\right|$$

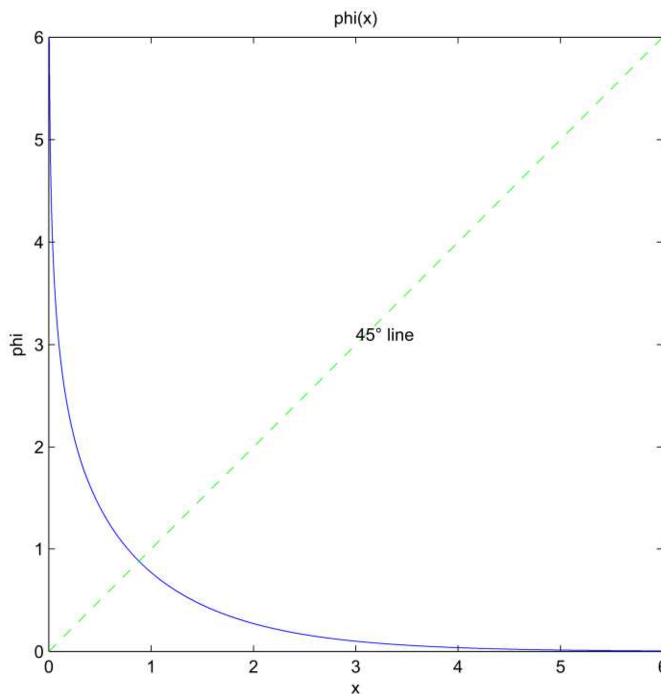


Figura 2.5: funzione φ(x)

Dato che la tangente iperbolica è una funzione dispari e monotona crescente, si può portare fuori il segno dell'argomento

$$tanh\left(\frac{1}{2}Lr_{mn}^{(l)}\right) = \left(\prod_{n'\in N(m)/n} \alpha_{n'm}\right)\left(\prod_{n'\in N(m)/n} tanh\left(\frac{1}{2}\beta_{n'm}\right)\right)$$

Si definisce una funzione

$$\varphi(x) = \ln\left(\frac{e^x + 1}{e^x - 1}\right) \tag{2.33}$$

inversa di se stessa, ossia $\varphi(x) = \varphi^{-1}(x)\ \forall x > 0$, e otteniamo infine la nuova formula del processing orizzontale

$$Lr_{mn}^{(l)} = \left(\prod_{n'\in N(m)/n} \alpha_{n'm}\right)\cdot \varphi\left(\sum_{n'\in N(m)/n} \varphi(\beta_{n'm})\right) \tag{2.34}$$

In questo modo il prodotto dei segni si può calcolare come somma del bit di segno dei vari fattori, la produttoria è stata sostituita da una sommatoria (la S in MSA sta per sum) e la funzione $\varphi(x)$ è abbastanza regolare da essere implementata come lookup table.

Successivamente è possibile applicare il logaritmo a entrambi i membri e derivare la nuova formula del processing verticale di seguito:

$$\begin{aligned} Lq_{nm}^{(l)} &= ln\frac{q_{nm}^{(l)}(0)}{q_{nm}^{(l)}(1)} = ln\left(\frac{\cancel{k_{nm}}(1-P_n)}{\cancel{k_{nm}}P_n}\prod_{m'\in M(n)/m}\frac{r_{m'n}^{(l)}(0)}{r_{m'n}^{(l)}(1)}\right) \\ &= LP_n + \sum_{m'\in M(n)/m} Lr_{m'n}^{(l)} \end{aligned} \tag{2.35}$$

Anche in questo caso la produttoria è diventata una sommatoria a vantaggio del tempo di esecuzione dell'algoritmo. In modo similare

$$LQ_n^{(l)} = LP_n + \sum_{m'\in M(n)} Lr_{m'n}^{(l)} \tag{2.36}$$

Come per l'SPA, a partire dagli $LQ_n^{(l)}$si possono decidere i bit della parola ricevuta per terminare l'algoritmo nel modo seguente:

$$x_n = \begin{cases} 0 \ \ se\ LQ_n^{(l)} > 0 \\ 1\ se\ \ LQ_n^{(l)} \le 0 \end{cases} \tag{2.37}$$

Con questo algoritmo la complessità è notevolmente ridotta, tuttavia resta il problema della funzione $\varphi(x)$ che, se implementata con una look-up table, manca di precisione e può richiedere una notevole quantità di memoria.

Inoltre fino ad ora sono stati introdotti solo alcuni artifici matematici ma l'algoritmo è fondamentalmente lo stesso dell'SPA. La vera approssimazione si ottiene studiando le proprietà della funzione $\varphi(x)$.

Si può infatti dimostrare che vale la seguente approssimazione

$$\varphi\left(\sum_{n'\in N(m)/n} \varphi(\beta_{n'm})\right) \simeq \varphi\left(\varphi\left(\min_{n'\in N(m)/n} \beta_{n'm}\right)\right) = \min_{n'\in N(m)/n} \beta_{n'm} \tag{2.38}$$

Quindi sostituendo questo termine nella formula precedente si ottiene la formula definitiva del processing orizzontale

$$Lr_{mn}^{(l)} = \left(\prod_{n'\in N(m)/n} \alpha_{n'm}\right) \cdot \min_{n'\in N(m)/n} \beta_{n'm} \tag{2.39}$$

in questo modo si ottiene quindi il vero e proprio algoritmo MSA, acronimo di min-sum algorithm ad evidenziare come sono state sostituite le numerose moltiplicazioni dell'SPA con più semplici somme e confronti.

2.3.3 Considerazioni per l'implementazione di MSA su sistemi many-core

Pensando di implementare l'algoritmo MSA su un sistema di calcolo parallelo è opportuno valutare il grado di parallelismo di questo algoritmo.

Vi sono due diverse dimensioni di parallelismo che si possono sfruttare, la prima rappresentata dal numero di parole di codice decodificate contemporaneamente.

Aumentando il numero di parole decodificate in parallelo possiamo aumentare in modo significativo l'utilizzo dei processori a discapito, però, del tempo di latenza. Inoltre le strutture dati utilizzate (per implementare il grafo di Tanner) sono sempre le stesse e di conseguenza anche la successione di istruzioni eseguite sarà la stessa per ogni flusso, come vedremo nel capitolo successivo ciò costituisce un vantaggio per una implementazione efficiente.

La seconda dimensione di parallelismo si individua all'interno della decodifica della stessa parola di codice nel processing orizzontale, nel processing verticale e nel calcolo degli $LQ_n^{(l)}$.

Prendendo in esame le formule (2.35), (2.36) e (2.39), si nota come il calcolo di un certo messaggio sia indipendente dagli altri. Per l'esattezza ci sono tanti flussi di calcolo indipendenti, e quindi che possono essere eseguiti in parallelo, quante sono le connessioni del grafo di Tanner ovvero gli '1' della matrice $\boldsymbol{H}$.

In [2] gli autori studiano il parallelismo di MSA e in particolare quale potrebbe essere la strategia più efficiente per una possibile implementazione su GPU.

MSA – processing orizzontale	
	Numero di operazioni
Comparazioni	$w_cM = E$
Abs	$w_cM = E$
Min	$(w_c\text{-}1)M = E\text{-}M$
Xor	$w_cM = E$
Accessi in Memoria	$2w_cM = 2E$
MSA –Processing Verticale	
	Numero di operazioni
Addizioni / Sottrazioni	$2w_bN = 2E$
Accessi in memoria	$(2w_b+1)N = 2E\text{-}N$

Tabella 2.1: Numero di operazioni aritmetiche e di accessi in memoria per iterazione per MSA ottimizzato e codice regolare [2]

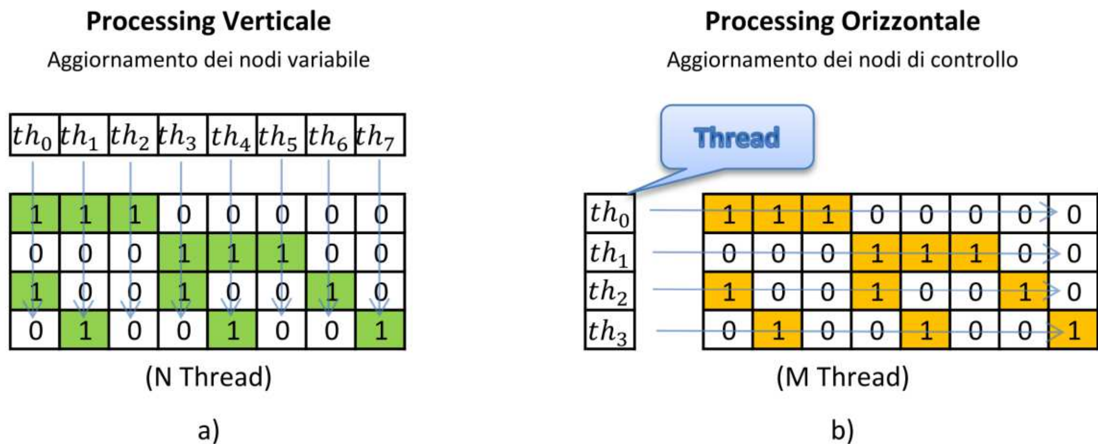


Figura 2.6: Associazione tra i thread e i messaggi che questo deve calcolare

La Tabella 2.1 riassume il tipo e il numero di operazioni elementari che un calcolatore deve svolgere per eseguire le due fasi dell'algoritmo MSA, ottenuti dagli autori.

Si può notare come il numero di operazioni dipenda in modo lineare dalle dimensioni e dalla densità di '1' della matrice ***H,*** espressa, considerando un codice regolare, coi coefficienti w_c e w_b.

Essi suggeriscono, per minimizzare il numero di accessi in memoria, di considerare un flusso di controllo per ogni nodo variabile nel processing orizzontale, e uno per ogni nodo di controllo nel processing verticale piuttosto che uno per ogni connessione nel grafo. In questo modo l'implementazione risulta più efficiente.

L'associazione tra un thread e i messaggi che questo deve calcolare è espressa in Figura 2.6 la quale mostra anche la provenienza della dicitura processing orizzontale e verticale.

Con questa suddivisione si possono introdurre alcuni stratagemmi per minimizzare ulteriormente il numero di operazioni.

Per esempio nel processing verticale si possono calcolare insieme gli $LQ_n^{(l)}$e gli $Lq_{nm}^{(l)}$. Le due formule sono molto simili e per non calcolare due volte la stessa sommatoria si può notare che

$$Lq_{nm}^{(l)} = LP_n + \sum_{m' \in M(n)/m} Lr_{m'n}^{(l)} = LQ_n^{(l)} - Lr_{mn}^{(l)} \qquad (2.40)$$

In questo modo, una volta calcolato $LQ_n^{(l)}$con la sommatoria, si possono calcolare i vari $Lq_{nm}^{(l)}$ che partono dal nodo variabile in esame con delle semplici sottrazioni.

Un approccio simile è usato nel processing verticale, dove con una prima iterazione si calcolano il β_{nm} minimo, il secondo β_{nm} più piccolo con i relativi indici e le produttorie

$$\prod_{n' \in N(m)} \alpha_{n'm}$$

Successivamente si trasmette il minimo trovato in precedenza se destinato a un nodo qualsiasi, il secondo valore minimo al nodo dal quale proviene il valore minimo, moltiplicando il valore da spedire con la produttoria dei segni che si calcola semplicemente come

$$\prod_{n' \in N(m)/n} \alpha_{n'm} = \alpha_{nm} \underbrace{\prod_{n' \in N(m)} \alpha_{n'm}}_{\text{già calcolato}} \qquad (2.41)$$

2.4 Codici LDPC nello standard DVB-S2

Lo standard DVB-S2 (Digital Video Broadcasting Satellite Second Generation) ha come scopo la definizione e la regolamentazione delle trasmissioni televisive digitali via satellite.

Fa parte di un'intera famiglia di standard del consorzio DVB project, come il DVB-T per le trasmissioni terrestri (comunemente chiamato "digitale terrestre"), e consiste in una evoluzione dello standard DVB-S.

I documenti contenenti le specifiche tecniche di tutti questi standard possono essere recuperati dal sito web dell'ETSI (European Telecommunications Standards Institute) [3].

Un sistema DVB-S2 è composto da varie parti: vari flussi di dati sono combinati dopo alcuni trattamenti per eliminare la ridondanza e per eseguire una sincronizzazione; in seguito viene applicato un primo codice di correzione BCH che considera cioè il canale binario; in seguito è applicata una codifica LDPC che invece considera un canale AWGN e lavora con valori reali.

I bit vengono quindi interlacciati in modo da essere protetti contro errori a burst e sono modulati e filtrati prima di essere inviati effettivamente sul canale.

Come si vede nello schema a blocchi in Figura 2.7, all'uscita dell'encoder il blocco di dati prende il nome di FECFRAME e può essere costituito da 64800 bit (normal FECFRAME) oppure 16200 bit (short FECFRAME).

La rate del codice può inoltre essere configurata in diversi modi a seconda del livello di protezione dal rumore che si vuole ottenere. Ovviamente una maggiore protezione (e quindi una rate più bassa) comporta una riduzione del throughput del sistema.

In [3] è descritto dettagliatamente l'algoritmo da utilizzare per fare la codifica: il codice LDPC usato nello standard è di tipo IRA sistematico. L'algoritmo non è esattamente quello descritto precedentemente, ma è facilmente riconducibile ad esso e non ha particolari vantaggi dal punto di vista computazionale. Negli allegati del documento dell'ETSI vi sono alcune tabelle che contengono alcuni indici di riferimento da utilizzare durante la codifica per risalire esattamente a quali bit di parità vadano sommati di volta in volta i bit di informazione. Alternativamente si può ri-

cavare con altri metodi la matrice di parità di ognuno di questi codici ed eseguire la moltiplicazione matriciale con G_I come illustrato precedentemente[8].

Le dimensioni delle matrici sono piuttosto significative, soprattutto nel caso di normal FECFRAME. È evidente come è necessario sfruttare il fatto che la matrice del codice è sparsa per poter eseguire tutti questi calcoli in tempi ragionevoli. I codici sono stati scelti appositamente per garantire una bassa densità di '1' pur mantenendo delle prestazioni vicine al massimo teorico.

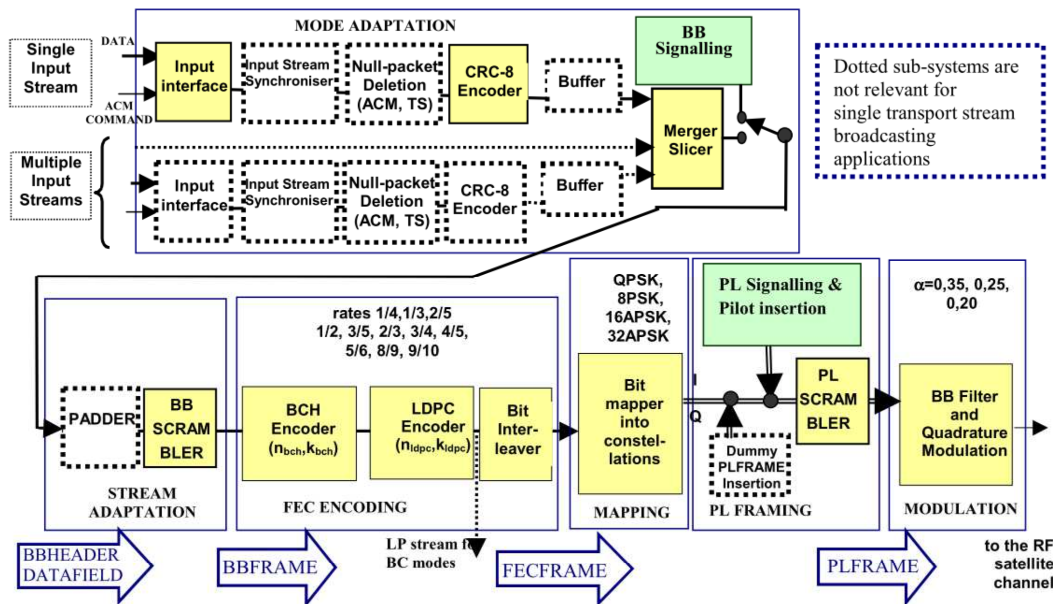


Figura 2.7: Schema a blocchi di un sistema di trasmissione DVB-S2 [3]

Nel caso di normal FECFRAME il numero di connessioni nel grafo di Tanner è nell'ordine delle decine di migliaia contro un numero totale di elementi nella matrice H dell'ordine dei miliardi.

Il grande numero di elementi della matrice significa inoltre che l'uso degli algoritmi presentati in precedenza garantirà un alta efficienza e scalabilità nell'utilizzo dei core di un sistema parallelo.

[8] La funzione Matlab dvbs2ldpc(rate) restituisce la matrice H del codice LDPC per una certa rate.

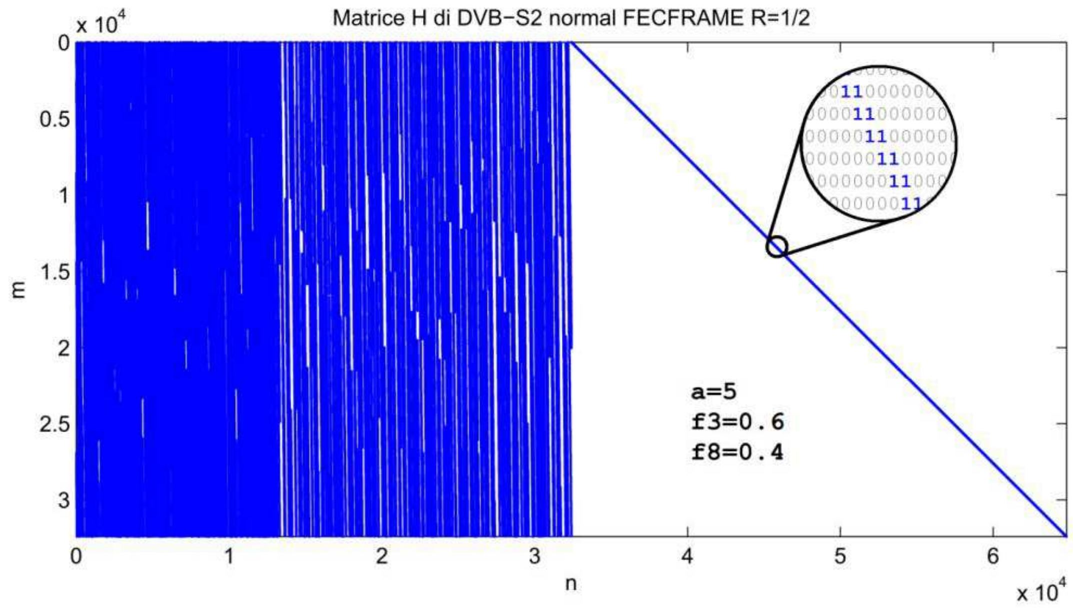


Figura 2.8: Esempio di matrice H di un codice LDPC del DVB-S2[9]

[9] Immagine ottenuta con il comando Matlab spy(dvbs2ldpc(1/2))

CAPITOLO 3

CUDA

3.1 Introduzione alla programmazione GPGPU

Già da diverso tempo si è capito che per migliorare le prestazioni di un calcolatore elettronico non è una soluzione intelligente il solo aumento della frequenza di clock, ma è necessario aumentare il numero di processori su un singolo chip in modo da poter eseguire più istruzioni contemporaneamente.

Ci sono applicazioni come l'elaborazione dei segnali o le simulazioni scientifiche che possono trarre un enorme beneficio dall'aumento dei processori nel sistema di calcolo. Queste infatti sono spesso costituite da una moltitudine di operazioni indipendenti tra di loro che, se eseguite in parallelo piuttosto che in modo sequenziale, permetterebbero l'esecuzione dell'intero programma in tempi drasticamente minori.

Già da qualche anno le maggiori case produttrici di processori hanno iniziato a mettere sul mercato CPU con un numero crescente di unità di elaborazione (o core). Oggigiorno ne esistono svariate con un numero di core vicino alla decina. Tuttavia, pur offrendo un vantaggio dal punto di vista del tempo di calcolo, le CPU multi-core restano specializzate nell'eseguire efficientemente istruzioni sequenziali e non sono adatte a una massiccia elaborazione parallela.

Per questo tipo di applicazioni, dove il livello di parallelismo può essere di un fattore nell'ordine delle migliaia piuttosto che delle decine, si possono usare le **GPU** (**Graphics Processing Unit**). Questa classe di dispositivi è stata pensata inizialmente per applicazioni puramente legate al mondo della grafica e dei videogiochi, ma ci si è resi conto successivamente che possono eseguire egregiamente qualsiasi tipo di calcolo massiccio parallelo.

Il termine **GPGPU** (**General Purpose GPU**) **Computing** è comunemente utilizzato da chi si occupa dell'integrazione di algoritmi di calcolo per indicare il loro adattamento ad essere eseguiti su dispositivi concepiti per la grafica.

Per scrivere un programma che utilizzi le potenzialità di una GPU è necessario utilizzare un linguaggio appropriato, che consenta di esprimere il parallelismo di un algoritmo, cosa che con i linguaggi tradizionali è impossibile.

Per la precisione, nei linguaggi di programmazione classici, è possibile creare nuovi flussi di esecuzione concorrente, o thread, utilizzando le API di sistema, ovvero chiedendo al sistema operativo sottostante di crearne e gestirne uno nuovo per noi. Questo poi sarà eseguito veramente in parallelo se la CPU sottostante è dotata di più core.

Esistono però molte situazioni in cui è interessante eseguire un insieme ristretto di istruzioni un numero elevato di volte, spesso utilizzando dati collocati in celle di memoria adiacenti. Chiedere al sistema operativo di creare così tanti thread comporta un enorme spreco di tempo, in quanto gestire un thread comporta un certo overhead; inoltre, visto che la CPU ha un numero limitato di core, il codice verrebbe eseguito almeno in parte in modo sequenziale.

I due principali ambienti di sviluppo per il GPGPU Computing sono OpenCL, che è uno standard aperto e disponibile gratuitamente agli sviluppatori, e CUDA, che invece è un prodotto commerciale e può essere utilizzato solo con schede grafiche di un determinato produttore.

Nel corso di questo lavoro è stato scelto CUDA. In questo capitolo verrà mostrata la filosofia di questo linguaggio e sarà schematizzato in funzionamento dell'hardware di supporto, concentrandosi maggiormente su quegli aspetti che sono rilevanti per lo svolgimento della tesi e che influenzano maggiormente le performance ottenute.

3.2 Elementi di CUDA

CUDA non è un vero e proprio linguaggio, ma è semplicemente un'estensione del C++ e di altri linguaggi: sostanzialmente introduce alcune parole chiave e costrutti che danno un'indicazione al compilatore su come andrà configurata la parte di programma che sarà eseguita sulla scheda grafica.

Il linguaggio però non è completamente svincolato dall'hardware sottostante: infatti, per scrivere un programma efficiente, è necessario aver ben chiaro cosa succede realmente all'interno della scheda grafica e CUDA permette di gestire manualmente

molti aspetti di basso livello come il tipo di memoria da usare o come organizzare l'esecuzione dei thread.

Tuttavia CUDA mantiene un livello di astrazione tale per cui lo stesso codice può essere compilato ed eseguito su sistemi con GPU Nvidia diverse, anche se può essere necessario cambiare alcuni parametri per migliorarne l'efficienza.

Il parametro più importante per valutare le capacità di una scheda grafica Nvidia (e quindi la compatibilità del codice scritto) è la **Compute Capability**. Questo valore indica in modo standard di quali feature è dotata una certa scheda grafica e, in generale, le sue capacità (ad esempio le più moderne GPU della famiglia Tesla usate nei grandi data center che hanno oggigiorno una Compute Capability di 3.5. [4]

3.2.1 L'Hardware

Secondo la terminologia di CUDA un sistema con scheda video è formato da un **Host** (ovvero un sistema PC dotato di una CPU di tipo standard su cui viene eseguito il programma principale) e da un **Device** (ovvero, la scheda video che esegue un certo compito richiesto dall'Host).

È importante sottolineare che Host e Device sono due entità ben separate: esse hanno infatti spazi di indirizzamento in memoria diversi e possono scambiarsi informazioni solo attraverso alcune API chiamate dall'Host. Ad esempio un errore comune è quello di dichiarare una variabile nella memoria del Device e poi cercare di usarla nel programma Host. Questo non è consentito perché l'indirizzo di memoria di quella variabile è nello spazio di indirizzamento del Device, e quindi per l'Host è completamente privo di significato. Per copiare il valore di una variabile dal Device all'Host si deve usare una funzione apposita chiamata cudaMemcpy.

La principale differenza che si nota osservando il layout di una CPU e quello di una GPU è come viene utilizzata l'area (Figura 3.1). Una CPU infatti ha buona parte dell'area occupata dall'elettronica che implementa alcune tecniche per velocizzare le istruzioni di controllo come la Branch Prediction; una GPU invece utilizza quasi la totalità dell'area per le unità matematiche. Una CPU quindi è molto efficiente nell'eseguire un programma con frequenti salti, mentre invece una GPU lo è nell'eseguire calcoli puramente aritmetici, e un singolo thread sarà complessivamente più lento rispetto all'esecuzione su CPU.

Le unità di elaborazione principali in una GPU sono gli **Streaming Multiprocessor (SM)**: ognuno di essi è formato da 32 **CUDA core**. Il numero di CUDA core è uno dei parametri che si trovano nelle specifiche tecniche delle schede video ed è

un'indicazione del livello di parallelismo e quindi del guadagno in prestazioni che si può ottenere.

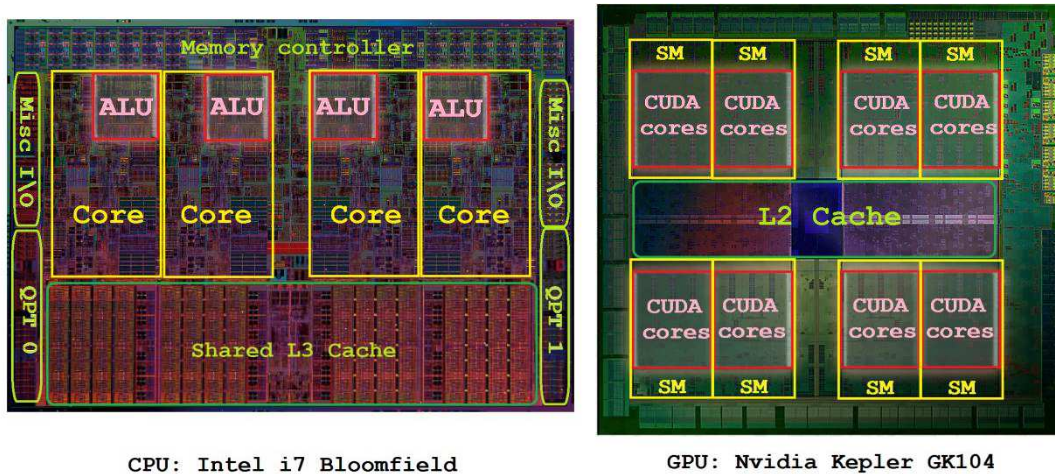


Figura 3.1: Confronto sull'utilizzo dell'area del chip tra una CPU e una GPU ALU e CUDA cores sono le porzioni di chip riservate ai calcoli aritmetici)

Come si può vedere in Figura 3.2, i core sono dotati di una unità aritmetica intera (INT Unit) e una floating point (FP Unit), quest'ultima in grado di eseguire un'operazione congiunta di moltiplicazione e somma in un ciclo di clock. [5]

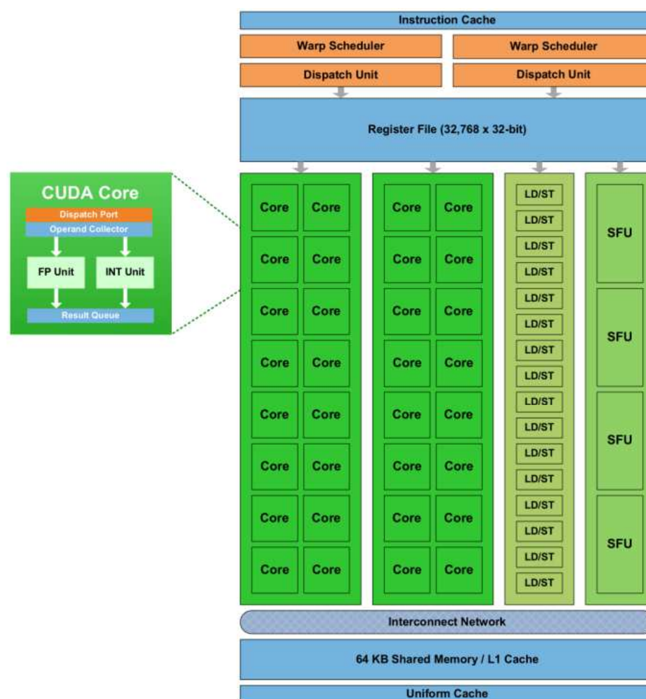


Figura 3.2: Architettura di un CUDA Streaming Multiprocessor

Questa capacità di eseguire circa con la stessa rapidità operazioni con numeri interi o con numeri a virgola mobile permette al programmatore di usare senza troppe preoccupazioni i float come tipo di dati. In una CPU spesso l'unità aritmetica a virgola mobile è più lenta e quindi impone al programmatore una riflessione sulle prestazioni prima di decidere che tipo di dato usare.

I 32 core sono poi organizzati in due **Execution Block (EB)** da 16 core ognuno e in aggiunta lo SM è dotato di altri due EB composti uno da 16 unità di Load/Store per l'accesso in memoria e l'altro da 4 Special Function Unit (SFU) in grado di eseguire operazioni speciali come il coseno o il reciproco di un numero.

Infine i 4 EB possono accedere al Register File composto da 32768 registri da 32 bit ciascuno, utilizzabili sia come interi che come numeri in virgola mobile, oppure a 64KB di memoria che può essere suddivisa tra memoria condivisa e cache di primo livello per la memoria principale.

3.2.2 Modello di esecuzione SIMD e Branch Divergence

Dal punto di vista dell'esecuzione delle istruzioni i thread sono raggruppati in entità chiamate **warp** che ne contengono tanti quanti sono i core.

Idealmente tutti i thread di un warp eseguono a ogni ciclo la stessa istruzione ma con operandi diversi: in pratica, l'unità che regola l'esecuzione delle istruzioni nello SM (la Dispatch Unit) propaga a tutti core di un EB l'istruzione da eseguire in parallelo (dato che un EB è composto da solo 16 core l'esecuzione verrà suddivisa in due fasi).
In particolare ogni SM ha due Dispatch Unit che collaborano e cercano di utilizzare due EB alla volta, prelevando le istruzioni dal Warp Scheduler che tiene traccia delle istruzioni dei warp in attesa di essere eseguiti.

Questo modello di esecuzione prende il nome di **Single Instruction Multiple Data (SIMD).**

In Figura 3.3 si può vedere il modo in cui le Dispatch Unit assegnano le istruzioni ai vari EB e come questi rimangono impegnati nel tempo. Si nota come ogni volta che un'istruzione viene assegnata a un EB questa sia ripetuta sempre 32 volte ma divisa in 8 cicli di clock nelle SFU e in 2 cicli di clock negli altri EB.

Il principale problema dovuto al modello di esecuzione SIMD si ha con le istruzioni di tipo branch, ovvero i salti condizionati.

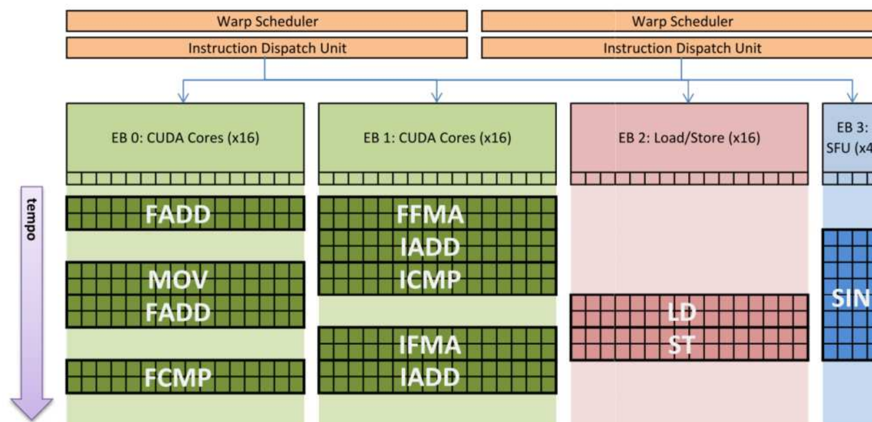


Figura 3.3: Esempio di come le Dispatch Unit assegnano ai vari EB le istruzioni.

A titolo d'esempio, supponiamo che a un certo punto, dei 32 thread in un warp, trenta debbano eseguire un'istruzione "A" e solo due thread una seconda istruzione "B" prima di ricongiungersi; supponiamo inoltre per semplicità che un solo EB sia libero e gli altri impegnati.

Quello che succede è che la Dispatch Unit prima propaga l'istruzione "A" disabilitando i due thread che non devono eseguirla, e dopo propaga l'istruzione "B" disabilitando gli altri trenta thread.

Questo fenomeno prende il nome di **Branch Divergence**.

Complessivamente il tempo di esecuzione è doppio rispetto al caso in cui tutti i thread avessero eseguito la stessa istruzione.

Per questo motivo, e per il fatto che in generale una GPU non è ottimizzata per eseguire branch, è consigliabile che il codice da eseguire in parallelo sia costituito da sole operazioni matematiche minimizzando l'uso di costrutti if o switch-case e dei cicli for/do/while.

3.2.3 Organizzazione dei thread

Dal punto di vista logico i thread vengono organizzati in maniera diversa rispetto al solo raggruppamento in warp.

In generale, un'applicazione completa è fatta da più **kernel:** questi sono l'equivalente di una funzione in C con l'unica differenza che il codice viene eseguito da tanti thread in modo concorrente sulla scheda grafica.

CUDA infatti permette di definire quanti thread creare per eseguire un kernel specificando attraverso un costrutto particolare.

I thread vengono raggruppati in **Thread Block (TB),** fino a un massimo di 1536 thread per Block.

Tutti i thread di un TB vengono eseguiti sullo stesso SM, quindi sono divisi a loro volta in warp da 32 thread. I thread dello stesso warp sono eseguiti in parallelo, mentre invece thread di warp diversi possono essere eseguiti anche sequenzialmente.

Infine i thread vengono distinti tra loro con un indice che può essere a 1, 2 o 3 dimensioni. La cardinalità di questo indice è una questione puramente formale e non inficia in nessun modo le prestazioni del programma.

Infine i TB sono organizzati a loro volta in una **Grid (G)** e identificati tramite un altro indice che di nuovo può avere 1, 2 o 3 dimensioni.

La Figura 3.4 riassume visivamente quanto detto.

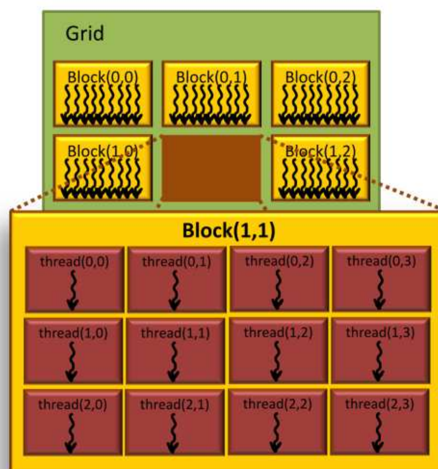


Figura 3.4: Organizzazione dei thread

```
__global__ void vecAdd(float d_a,float d_b,float d_c){      //kernel
        int idx = blockDim.x*blockIdx.x+threadIdx.x;        //blockDim = (32,1,1)
        d_c[idx] = d_a[idx] + d_b[idx];
}

int main(int argc,char **argv){
        ...
        vecAdd<<<4,32>>>(d_a,d_b,d_c);       //kernel lanciato, 32*4 = 128 thread
        ...
}
```

Codice 3.1: Come lanciare un kernel impostando il numero dei thread

Il Codice 3.1 mostra il programma Host che lancia un kernel, distinguibile dal qualificatore __global__ che lo identifica come codice eseguito dalla GPU. Questo viene lanciato con 4 TB da 32 thread ciascuno. All'interno del codice vengono usate delle variabili speciali, blockDim che contiene le dimensioni di ogni TB, blockIdx e threadIdx che contengono l'indice corrente rispettivamente del TB e del thread.

Per maggiori dettagli l'Nvidia mette a disposizione il manuale di CUDA [6].

3.2.4 Memorie

A differenza dell'Host dove esiste solo un tipo di memoria, nelle schede video ve ne sono più tipi a seconda dell'utilizzo che se ne vuole fare.

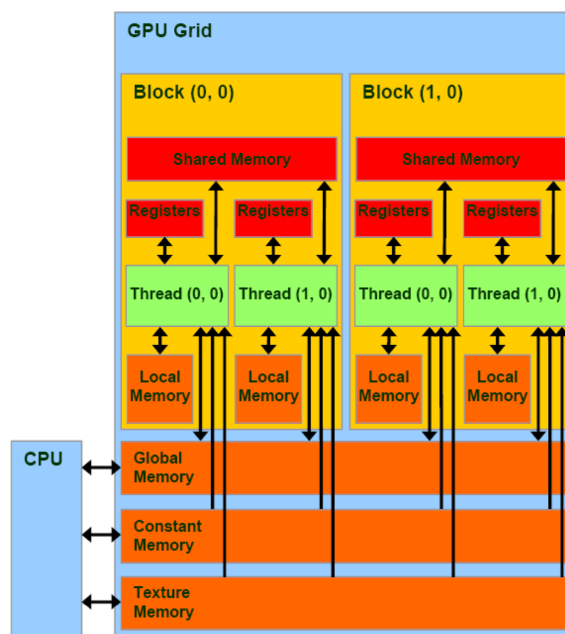


Figura 3.5: Le memorie della GPU e la loro visibilità

La memoria principale prende il nome di **Global Memory**. Questa non è ottimizzata per nessuna applicazione particolare ma la banda dei trasferimenti tra questa memoria e gli SM può essere molto alta. L'unica particolarità di questa memoria è che l'hardware cerca di fondere insieme tante richieste di lettura o scrittura da parte di thread dello stesso warp in un unico trasferimento così da risparmiare tempo. Questo processo è descritto in dettaglio nel paragrafo 3.3.2 sulla coalescenza. I trasferimenti possono essere da 32, 64 o 128 byte.

Per dispositivi dotati di Compute Capability superiore a 2.0, l'accesso alla Global Memory avviene attraverso una cache di livello 2 e, se l'hardware è configurato in modo appropriato, anche attraverso la cache di livello 1 presente in ogni SM.

Come già accennato in precedenza, la **Shared Memory** condivide insieme alla cache di livello 1 i 64KB di memoria presenti in ogni SM.

Al di là di applicazioni dove i thread devono cooperare tra loro e quindi usano questa memoria come supporto di scambio dei dati, la Shared Memory è usata in genere come se fosse una cache controllabile manualmente: il programmatore, infatti, può decidere di caricarvi dalla Global Memory i dati necessari al TB per eseguire il suo compito.

Visto che la Shared Memory fa parte dello SM, questa è suddivisa in tante parti quanti sono i TB in esecuzione su quel particolare SM. Per questo motivo una certa porzione di Shared Memory è visibile solo ai thread dello stesso TB.

Spesso ci sono variabili che sono accedute in sola lettura e quindi non cambiano mai di valore. In questi casi conviene usare la **Constant Memory**, che pur essendo piuttosto piccola(solo 64KB), è dotata di una cache dedicata ed è ottimizzata per la lettura.

Nel mondo della grafica una Texture è un'immagine che riveste un oggetto tridimensionale virtuale dando l'illusione che questo sia composto da un certo materiale. La GPU è nata per applicazioni di grafica e visto che in questo ambito si fa un uso intensivo delle Texture, è dotata di una memoria apposita per caricare queste immagini, la **Texture Memory**.

In generale una memoria cache ha lo scopo di mantenere una copia "locale", a rapido accesso, di un certo dato. Per ragioni di efficienza, vengono copiati anche i dati adiacenti, perché per il principio di località spaziale è probabile che saranno i prossimi ad essere caricati. Tuttavia la memoria è lineare negli indirizzi.

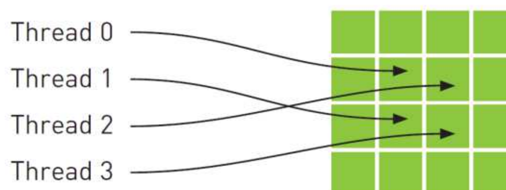


Figura 3.6. Principio di Località Spaziale in 2 dimensioni

La cache della Texture Memory invece è ottimizzata per sfruttare la località spaziale anche in 2 dimensioni, quindi è l'ideale per tutte quelle applicazioni dove si elaborano immagini o, in generale, dove thread vicini disposti in due dimensioni leggono i rispettivi pixel di una Texture bidimensionale.

La Texture Memory è di sola lettura e permette anche di lavorare con “immagini” in 3 dimensioni. Inoltre sul chipset è presente dell’hardware apposito che permette di indirizzare i pixel con un numero reale piuttosto che con un indice intero e avere in uscita un valore normalizzato da 0 a 1 e interpolato nel caso in cui venga indirizzata una posizione intermedia tra due pixel. Infine nel caso in cui venga indirizzato un pixel all’esterno della Texture questa viene ripetuta indefinitamente in tutte le direzioni.

Se si vuole poter anche scrivere in strutture dati simili alle Texture o comunque in 2 o 3 dimensioni si può usare la **Surface Memory**.

In realtà la Surface Memory risiede sempre nella Texture Memory, ma usando API diverse è possibile anche scriverci a discapito delle prestazioni.

3.3 Tecniche di ottimizzazione

Dopo questa panoramica è possibile capire appieno alcune delle tecniche che si usano per migliorare l’efficienza di un programma.

Nei paragrafi successivi le analizzeremo con maggiore dettaglio.

3.3.1 Occupancy

Il principale collo di bottiglia per le performance delle GPU è costituito dal notevole tempo di accesso alla memoria globale.

La banda del bus, infatti, viene utilizzata per intero solo nel momento in cui il dato è pronto per essere trasferito; durante il lungo periodo tra la richiesta e il recupero dell’informazione da parte della memoria, questo è inutilizzato, portando a una riduzione complessiva del tasso di utilizzo.

Quello che si cerca di fare è di attivare più trasferimenti possibili in modo da usare tutta la capacità del bus. Ovvero, una volta che un thread ha fatto una richiesta, un secondo thread ne fa subito un’altra, e così via, fintanto che a un certo punto la prima richiesta sarà pronta e immediatamente dopo anche la seconda eccetera, impegnando al 100% la banda del bus.

Il problema di questa strategia è che ogni SM deve tenere traccia di tutte le transazioni: questo limita il numero massimo di warp che possono essere attivi in uno SM.

Questo limite per Compute Capability 2.x è pari a 48, ovvero ci possono essere $48 \times 32 = 1536$ thread attivi per ogni SM[10].

Tuttavia ci possono essere delle situazioni tali per cui ci sono effettivamente meno thread attivi rispetto al limite massimo. L'occupancy è proprio il rapporto tra il numero di thread attivi per SM e il numero massimo possibile (ovvero 1536).

Una delle cause di questa inefficienza può essere l'utilizzo eccessivo da parte dei thread delle risorse come il Register File o la Shared Memory: per esempio se ogni thread utilizza 64 registri, dato che ogni SM ne ha 32768, ci possono essere solo $\frac{32768}{64} = 512$ thread attivi contro i 1536 massimi, riducendo l'occupancy al 33%.

Un'altra causa frequente di bassa occupancy è una configurazione errata di thread e thread Block. Per questioni analoghe a quelle sopracitate, anche il numero massimo di TB attivi in uno SM è limitato. Se il numero di thread per TB è impostato a un valore basso, anche il numero di thread attivi sarà inferiore al massimo teorico.

La Nvidia mette a disposizione un foglio di calcolo che una volta inserita la Compute Capability della GPU che si vuole usare permette di trovare facilmente le impostazioni ottimali per garantire il massimo dell'occupancy.

Un'alta occupancy permette, in definitiva, di mascherare il tempo di latenza della memoria globale, influenzando maggiormente le prestazioni di un'applicazione che esegue soprattutto accessi in memoria; effetti secondari permettono di mascherare altre latenze sull'esecuzione di calcoli.

La guida sulle buone norme da seguire programmando in CUDA [7] afferma che esiste un valore di occupancy oltre il quale non vi è più nessun miglioramento significativo sulle prestazioni, ovvero che non è necessario avere il 100% di occupancy per sfruttare il bus al massimo.

Addirittura, per certe applicazioni, un'occupancy troppo alta potrebbe portare a un peggioramento, per esempio causando l'aumento della miss rate della cache.

Questo valore si aggira attorno al 60%.

[10] Queste specifiche oltre ad essere raccolte in diversi documenti come [6], si possono anche ricavare utilizzando l'API CUDA "cudaGetDeviceProperties"

3.3.2 Coalescenza

La Global Memory non è ottimizzata per scopi particolari, tuttavia offre un meccanismo, la **coalescenza**, che permette di ridurre drasticamente la quantità di trasferimenti dalla memoria.

Per ridurre la frequenza degli accessi in memoria, lo SM, a fronte di più richieste da parte di vari thread nello stesso warp, cerca di aggregarle in un unico accesso.

Dalla Compute Capability 2.0 in poi, è presente una cache tra gli SM e la Global Memory, quindi i singoli accessi in memoria vengono già naturalmente aggregati dal funzionamento stesso di questa, che, caricando un'intera linea di cache da 128 byte, preleverà così anche i dati adiacenti.

Con le GPU di Compute Capability 1.0, invece, i trasferimenti avvengono direttamente tra gli SM e la memoria globale e il fatto che venga fatta una richiesta piuttosto che molte di più fa una grande differenza.

Prendendo in esame il caso di GPU dotata di cache di primo livello, supponiamo che i 32 thread di un warp vogliano accedere ad altrettanti interi di 4 byte consecutivi in un vettore, ad esempio dall'indirizzo 0x1000 all'indirizzo 0x107F, cioè 128 byte in totale. Questo blocco di memoria è allineato e perciò viene caricato in cache con un unico trasferimento da 128 byte. Viceversa se gli indirizzi non fossero stati consecutivi tra loro, ogni accesso avrebbe potuto richiedere un trasferimento a sé stante e quindi un tempo 32 volte più grande.

Per ottimizzare gli accessi in memoria con questa tecnica è importante anche tenere conto dell'allineamento dei dati in memoria. Infatti, se a differenza del caso precedente, gli indirizzi richiesti dai thread fossero stati da 0x1004 a 0x1083 sarebbero stati fatti due trasferimenti da 128 byte dalla Global Memory alla cache, uno dall'indirizzo 0x1000 al 0x107F e uno da 0x1080 a 0x10FF con un tempo di esecuzione complessivo doppio.

Nel caso di Compute Capability 1.0 l'effetto sarebbe stato anche peggiore perché lo SM avrebbe spezzato la richiesta in 5 trasferimenti da 32 byte ciascuno, il primo da 0x1000 a 0x101F fino all'ultimo da 0x1080 a 0x109F per un tempo di esecuzione quintuplo.

La Figura 3.7 rappresenta i tre esempi appena descritti mostrando in verde i trasferimenti eseguiti dalla memoria globale agli SM.

La cache ha quindi un effetto mitigante ma il massimo delle prestazioni si ottiene in entrambi i casi accedendo ad indirizzi consecutivi ed allineati.

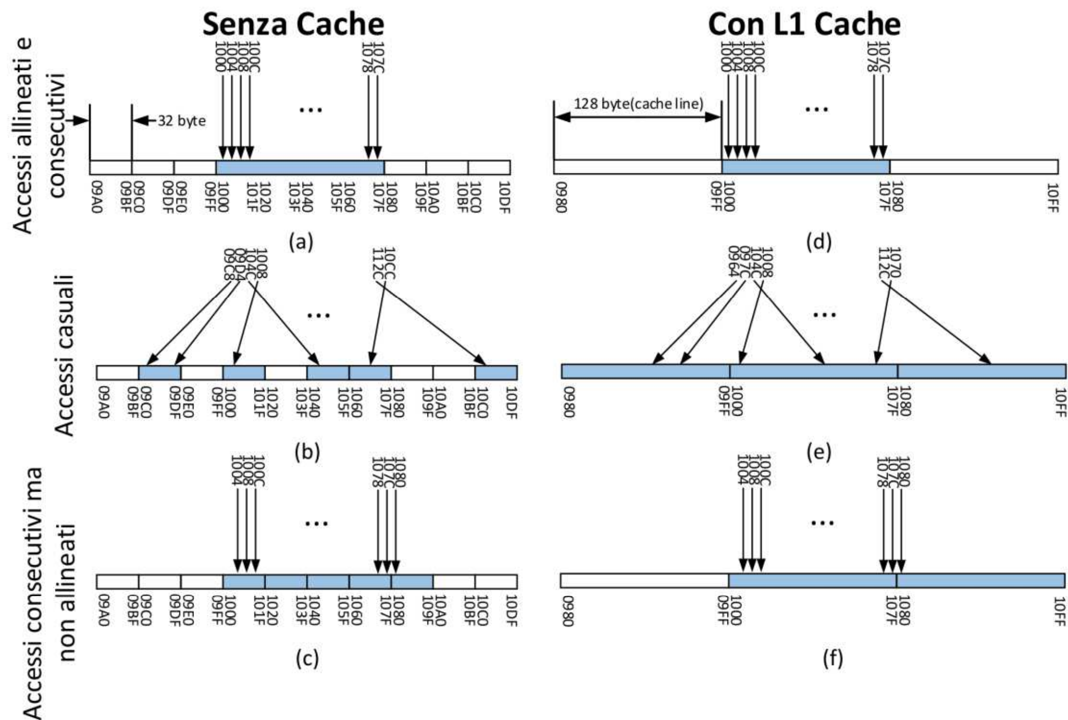


Figura 3.7: Effetto della cache e del disallineamento sul numero di trasferimenti (i riquadri verdi).

3.3.3 Evitare le Branch

Come è stato descritto nei capitoli precedenti, il metodo di esecuzione SIMD offre numerosi vantaggi ma impone grande attenzione nell'utilizzo delle istruzioni branch.

Di per sè una GPU non è ottimizzata per eseguire queste istruzioni velocemente, inoltre in caso di Branch Divergence si può avere un ulteriore calo delle prestazioni.

Spesso è preferibile che i kernel siano una semplice sequenza di istruzioni senza salti, addirittura il compilatore cerca dove possibile di svolgere i cicli che non dipendono dall'indice del thread o dal valore di un registro in modo che diventino veramente una sequenza ripetuta di istruzioni macchina.

Non esiste una vera e propria tecnica per ottenere una maggiore efficienza nell'esecuzione perché dipende molto dall'algoritmo.

Capita spesso di osservare come un codice con un ciclo solo e tante istruzioni al suo interno sia più veloce di un codice con più cicli ma con meno istruzioni da eseguire.

Inoltre in alcuni casi è possibile evitare di usare del tutto le branch. Ad esempio, nel Codice 3.2, che non fa altro che assegnare 0 o 1 alla variabile "result" a seconda che "a" sia maggiore o minore di 0, si può osservare come sia possibile prelevare il bit di segno di "a" per poi scriverlo direttamente, una volta traslato a destra, nel risultato usando così solo pure istruzioni di calcolo.

```
signed int a;                        //SXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
                                     //|
                                     // `-> Sign Bit in complemento a 2
unsigned char result;

//codice lento
if(a>0) result=0x00;
else result=0x01;

//equivalente ma più veloce perchè senza branch
int temp = a;                        //SXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX   &
temp = temp & 0x80000000;            //1000 0000 0000 0000 0000 0000 0000 0000   =
                                     //-----------------------------------------
                                     //S000 0000 0000 0000 0000 0000 0000 0000
temp = temp >> 31;                   //0000 0000 0000 0000 0000 0000 0000 000S
result = (unsigned char)temp;        //0000 000S (0x01 se a<0, 0x00 se a>0)
```

Codice 3.2: Esempio di stratagemma per evitare l'uso di branch

3.3.4 Memorie speciali e trasferimenti da Host a Device

Precedentemente sono stati descritti tutti i tipi di memorie presenti su una scheda video: a seconda del tipo di applicazione conviene usare un tipo di memoria piuttosto che un altro.

Tuttavia bisogna ricordarsi che il reale vantaggio si ha quando un applicazione esegue un grande numero di trasferimenti. Nel caso in cui ogni indice in memoria venga letto o scritto solo una volta il vantaggio diventa trascurabile rispetto all'utilizzo della semplice Global Memory con coalescenza; anzi talvolta, soprattutto per Compute Capability 2.0 o maggiore, questo sistema è da preferire perché se usata bene la L1 cache della Global Memory ha la banda più alta rispetto agli altri tipi di memorie.

I trasferimenti interni tra le memorie nella GPU e i Core non sono però gli unici trasferimenti possibili: spesso l'applicazione richiede un trasferimento di informazioni tra l'Host e il Device.

Questo spesso è il vero collo di bottiglia in un'applicazione ed esistono due tecniche per ridurre o mascherare il problema.

Normalmente nell'Host per implementare il concetto di memoria virtuale, la memoria è "paginata" ovvero suddivisa in blocchi chiamati pagine, che possono essere spostati dalla RAM all'Hard Disk a seconda delle necessità del sistema operativo.

Quando si vuole trasferire un dato dalla memoria principale alla Global Memory del Device usando il DMA (Direct Memory Access) una pagina viene allocata, gli indirizzi vengono traslati dallo spazio di indirizzamento virtuale a quello fisico e tutto ciò comporta un overhead non indifferente.

È consigliabile quindi usare un'API apposita per allocare i dati in pagine bloccate in memoria (page-locked memory). In questo modo i dati sono fissi ad un certo indirizzo e non c'è bisogno di nessuna conversione od overhead ottenendo così dei miglioramenti fino al 50%.

Il principale svantaggio è che se troppi dati vengono allocati in questo modo il sistema potrebbe andare in deadlock per esaurimento di memoria, anche se oggigiorno è un'eventualità remota.

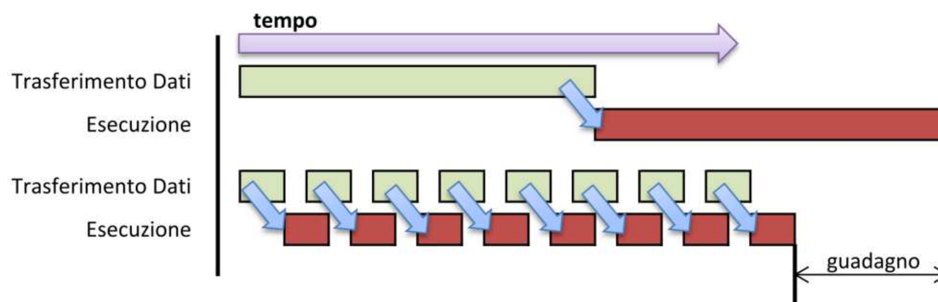


Figura 3.8: Vantaggio dell'esecuzione asincrona e sovrapposta ai trasferimenti

La seconda tecnica si basa sul fatto che un'applicazione con intensi trasferimenti dall'Host al Device in genere alterna fasi in cui i dati vengono copiati a fasi in cui vengono elaborati. Dato che il Device può elaborare dati e contemporaneamente trasferirne di nuovi, conviene alternare più frequentemente queste fasi in modo da minimizzare il tempo di inattività dei core o del bus. La Figura 3.8 riassume questo concetto in modo evidente.

CAPITOLO 4

Progetto e Implementazione

4.1 Introduzione

La parte sperimentale di questa tesi consiste nella progettazione e implementazione di un applicativo che permetta di eseguire la codifica e la decodifica LDPC di dati, sfruttando una scheda video.

Il principale requisito richiesto per questo progetto è la grande flessibilità e la possibilità di interfacciarsi con altre applicazioni o parti del sistema operativo, in particolare ci dovrà essere la possibilità di codificare dati qualsiasi presenti in memoria principale e di posizionare qui le informazioni decodificate provenienti da un'interfaccia radio esterna.

Le funzionalità sviluppate potranno essere, in futuro, integrate con altri blocchi di elaborazione che, installati su una piattaforma hardware embedded, andranno a formare un terminale di ricezione completo.

Con flessibilità si intende la possibilità, per esempio, di cambiare a tempo di esecuzione il codice LDPC in uso, oltre che al fatto di scrivere un codice riutilizzabile facilmente in futuro secondo la filosofia della Software Defined Radio[11].

[11] Nuova tecnologia emergente che prevede l'elaborazione dei dati e l'implementazione di protocolli completamente via software in modo altamente flessibile ed utilizzando hardware riprogrammabile

Dal punto di vista delle prestazioni è richiesto un grande throughput: il software deve essere eseguito nel modo più efficiente possibile senza compromettere la flessibilità.

Nei paragrafi seguenti è descritta la struttura generale del progetto, come vengono implementati i codici LDPC, lo schema a blocchi della catena di elaborazione implementata e come utilizzare le API messe a disposizione. Infine saranno brevemente analizzate le strategie usate per l'implementazione degli algoritmi. Questo capitolo ha lo scopo di far comprendere funzionamento e le strategie utilizzate per rendere il software il più efficiente possibile.

4.2 Struttura generale

Dato che le applicazioni di un encoder/decoder LDPC su GPU possono essere innumerevoli, è sembrato ragionevole racchiudere tutte le funzionalità utili in un unica libreria in modo da poter realizzare agevolmente, in un secondo momento, l'applicazione vera e propria.

Le funzionalità principali sono state racchiuse, quindi, nella libreria LDPC.lib.

Il sistema è composto da vari blocchi che possono essere abilitati e disabilitati in base alle esigenze; lo schema è rappresentato in Figura 4.1.

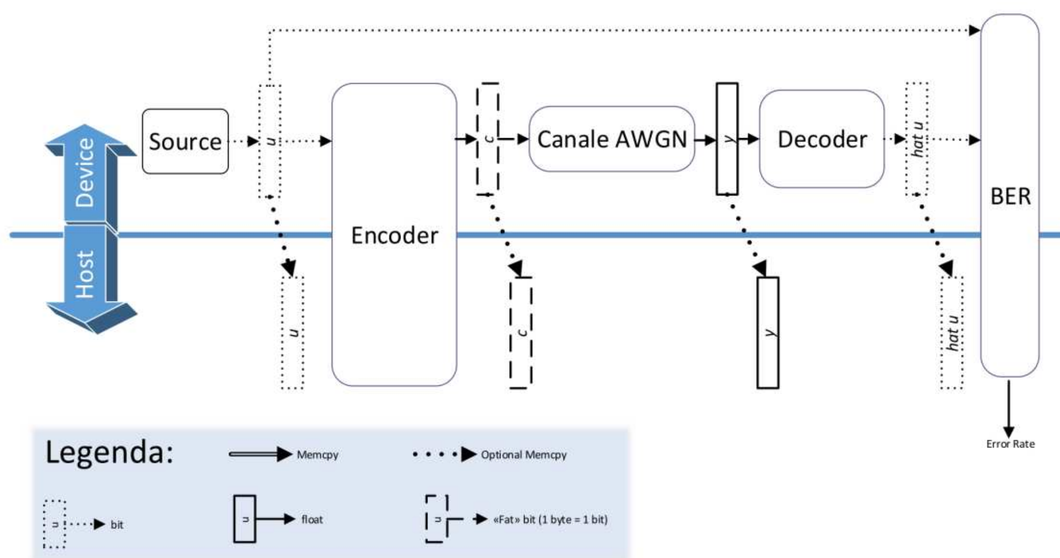


Figura 4.1: Schema a blocchi del sistema

I vari blocchi, e le funzioni ad essi associate, implementano gli algoritmi di elaborazione che manipolano le informazioni lungo la catena di trasmissione e ricezione: la

generazione dei bit casuali, la codifica LDPC, la simulazione del canale AWGN, la decodifica e la misura della Bit Error Rate (**BER**).

Tra un blocco e il successivo sono posti dei buffer, i dati in essi contenuti sono immagazzinati in diversi formati a seconda delle esigenze, dai bit senza nessuna formattazione fino ai numeri reali memorizzati in virgola mobile (float).

Per ridurre al minimo i trasferimenti di dati tra l'Host e il Device, tutti questi buffer, così come quelli interni ai blocchi, sono allocati nella memoria globale del Device, in modo che, dalla generazione dei bit fino alla loro decodifica, non esista nessun bisogno di coinvolgere la memoria dell'Host.

Tuttavia esistono delle copie dei buffer anche nella memoria principale, in questo modo altre parti del programma, o il sistema operativo stesso, possono interagire con i blocchi eseguendo dei trasferimenti ad hoc.

È stata creata una struttura dati, systemCtrl, per raccogliere gli indirizzi ai quale si trovano tutti questi buffer insieme con altre informazioni. È possibile trovare la definizione di questa struttura nel Codice 4.5.

Il sistema elabora contemporaneamente più blocchi di bit, o parole.

Il primo blocco crea, nel buffer 'u', le C_P parole da K bit ognuna usando una libreria che permette la generazione di numeri pseudo-casuali (cuRand.lib)

Queste parole vengono codificate dal blocco encoder LDPC; questo, per ora, implementa solo la codifica IRA e genera, nel buffer 'c', le parole di codice.

Il simulatore di canale somma, attraverso la libreria sopracitata, il rumore AWGN a una versione modulata in BPSK[12] dei bit codificati.

Il decoder, invece, esiste in tre versioni in base alla precisione delle variabili interne usate: 8 bit, 16 bit e float. Questo esegue l'algoritmo MSA per risalire alle parole di codice inviate. In seguito, assumendo il codice come sistematico, ricava i bit di informazione originari ponendoli nel buffer '*hat u*'.

Il blocco finale confronta i bit di informazione trasmessi con quelli decodificati e calcola il tasso di errore (BER).

[12] La modulazione BPSK non è tra quelle previste dallo standard DVB-S2, tuttavia ha le stesse caratteristiche, in termini di Bit Error Rate, della QPSK

Infine, tutte le informazioni relative alle matrici e alle caratteristiche del codice LDPC, sono contenute in una seconda struttura dati: CodeSpecs, la cui definizione è visionabile nel Codice 4.1.

Allo stesso modo di systemCtrl, questa struttura consiste in un elenco di puntatori ai vettori allocati nella memoria del Device contenenti le informazioni riguardo la matrice di parità; CodeSpecs, contiene inoltre vari parametri come le dimensioni delle matrici ***G*** ed ***H***, ed altri valori di comodo come la rate del codice espressa come frazione e l'occupazione in byte dei blocchi di bit.

4.2.1 Rappresentazione dei codici LDPC

Sicuramente una delle parti più importanti di questo lavoro è il modo in cui vengono rappresentati i codici LDPC utilizzati e, in particolare, le loro matrici di parità.

```
struct CodeSpecs{
        int N,K,M,Rn,Rd,Edges,wcmax,wbmax;   //rate = Rn/Rd = R
        float R,wcavg,wbavg;         //code rate, average weight of columns and rows
        int NinByte,KinByte,MinByte;
        int type,subtype;
        int *d_Hbn,*d_Hcn;           //links between nodes in the Tanner Graph
        int *d_wc,*d_wb,*d_wbcum,*d_wccum;//col and row weights and their cumulative
        int *d_Hrows;         //matrix H in sparse format per row (great for H*c)
        int extrasLen;        //length of extra data
        char *d_extras;       //extra data (unimplemented)
};
```

Codice 4.1: Definizione della struttura CodeSpecs

La struttura dati che raccoglie tutte le informazioni pertinenti al codice LDPC, CodeSpecs, è composta da una serie di valori e di puntatori a vettori.

N, K ed M sono, secondo la dicitura usata anche nel capitolo 2, le dimensioni delle matrici ***G*** ed ***H***, oppure, rispettivamente, la lunghezza di una parola di codice, la lunghezza di un blocco di bit di informazioni e il numero di bit di parità.

Rn e Rd rappresentano numeratore e denominatore di una frazione di valore pari alla rate del codice.

Edges è il numero di '1' della matrice ***H***, oppure il numero di lati del grafo di Tanner; indicato in precedenza come E.

Wcmax e wbmax esprimono il massimo dei pesi rispettivamente di righe e colonne della matrice ***H***.

R è pari alla rate del codice.

Wcavg e wbavg sono la media dei pesi di righe e colonne di ***H***.

N,K e MinByte esprimono il numero di byte che una parola da N,K e M bit occupa in memoria, sono calcolati semplicemente arrotondando per eccesso il numero di bit diviso per 8.

Type e Subtype sono due parametri che, in base al valore assunto, indicano al sistema di che tipo di codice LDPC si tratta, e quindi, l'algoritmo di codifica o decodifica più opportuno da utilizzare. I valori che queste variabili possono assumere sono definiti in LDPC_constants_macros.h; dato che in questo lavoro ci si è concentrati solo sullo standard DVB-S2, l'unico tipo di codice che può essere elaborato interamente è l'IRA sistematico.

Infine la struttura dati contiene una serie di puntatori a vettori memorizzati nella Global Memory della scheda video.

I vettori d_wc e d_wb contengono i pesi delle singole righe e colonne della matrice ***H***, mentre d_wccum e d_wbcum le corrispondenti cumulative ovvero

$$
\begin{aligned}
d_wccum[k] &= \sum_{n=0}^{k-1} d_wc[n] \;\; \forall k \in [0, M] \\
d_wbcum[k] &= \sum_{n=0}^{k-1} d_wb[n] \;\; \forall k \in [0, N]
\end{aligned}
\tag{4.1}
$$

Questi vettori sono utili per il calcolo degli indici durante le operazioni di codifica e decodifica.

Il vettore d_Hrows contiene gli indici di colonna dove compaiono degli '1' nella matrice di parità, raggruppati per righe. Utilizzando la notazione del paragrafo 2.2.1, si ricorda che i primi 'a' indici di ogni riga sono anche gli indici della matrice $\boldsymbol{G_1}$ di un codice IRA, e quindi si possono usare per eseguire la moltiplicazione matriciale richiesta dalla codifica.

Infine i due vettori d_Hcn e d_Hbn sono utilizzati dal decodificatore per risalire agli insiemi dei nodi connessi $M(n)$ e $N(m)$ presenti nelle formule (2.35), (2.36) e (2.39), e a recuperare, quindi, i messaggi giusti.

Facendo riferimento alla Figura 4.2-c, i messaggi Lq e Lr sono memorizzati in due vettori, e vi è un messaggio per ogni connessione del grafo.

Supponendo di dover trovare il seguente minimo

$$\min_{n' \in N(m)/n} \left| Lq_{mn'}^{(l)} \right| \quad \text{con } n = 4, m = 2$$

che richiede un ciclo attraverso gli elementi dell'insieme N(2)/4, ovvero, come nell'esempio mostrato in figura,

$$N(2)/4 \equiv \{0, 6, 7\} \Rightarrow \min\left\{ \left| Lq_{20'}^{(l)} \right|, \left| Lq_{26'}^{(l)} \right|, \left| Lq_{27'}^{(l)} \right| \right\}$$

vediamo come questi tre messaggi siano sparpagliati lungo il vettore Lq.

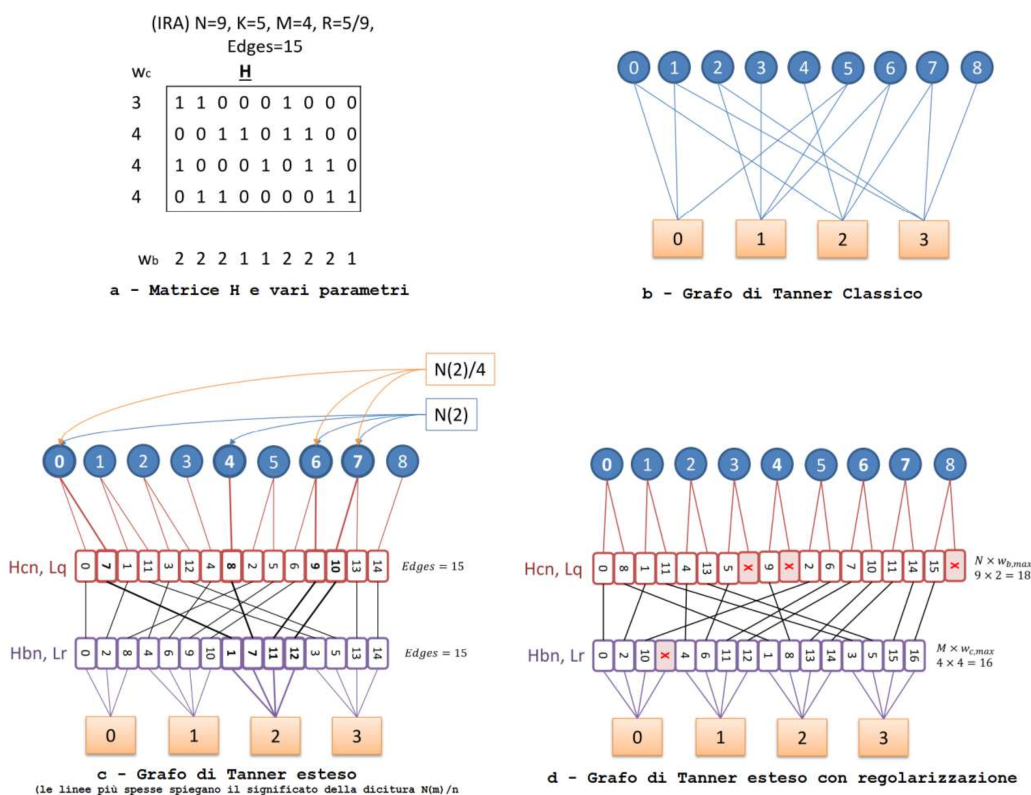


Figura 4.2: Spiegazione grafica del significato dei vettori Hcn e Hbn

Hbn contiene, adiacenti tra loro, gli indici all'interno di Lq di questi messaggi, quindi è possibile eseguire un ciclo attraverso di essi come proposto dal Codice 4.2.

Per velocizzare questo algoritmo si è pensato di modificare la struttura dei vettori posizionando gli elementi in modo più regolare in modo da calcolare più facilmente l'indice iniziale e finale entro i quali eseguire il ciclo (from e to).

Nella Figura 4.2-d, si può esaminare la versione definitiva dei vettori Hbn e Hcn, detta "regolarizzata".

Gli elementi dei vettori sono raggruppati in maniera uniforme basandosi sul peso massimo di righe e colonne: in questo modo è possibile ricavare gli indici impiegando meno tempo ottenendo, quindi, un guadagno in prestazioni. Tuttavia alcuni elementi dei vettori resteranno inutilizzati (nella figura, quelli evidenziati con una croce rossa).

Il Codice 4.3 mostra come eseguire un'iterazione tra gli elementi dell'insieme N(m) tenendo conto di quest'ultima modifica. Si può notare che all'inizio viene eseguito un solo accesso in memoria invece che due.

```
m=2 => ciclo tra i N(m) = N(2)
from = wccum[m]        //7    //accesso in memoria
to = wccum[m+1]        //11   //accesso in memoria

for(i=from;i<to;i++){
        index = Hbn[i]         //index = 1,7,11,12
        Lq_element = Lq[index]
...
}
```

Codice 4.2: esempio di iterazione tra i messaggi dell'insieme N(m) in pseudo-codice

```
m=2 => ciclo tra i N(m) = N(2)
from = m*wcmax         //8
to = from+wc[m]        //12   //accesso in memoria

for(i=from;i<to;i++){
        index = Hbn[i]         //index = 1,8,13,14
        Lq_element = Lq[index]
...
}
```

Codice 4.3: esempio di iterazione tra i messaggi dell'insieme N(m) in pseudo-codice con vettori regolarizzati

Lo svantaggio principale della regolarizzazione è lo spreco di memoria: tanto più un codice è irregolare, ovvero tanto più i pesi di righe e colonne differiscono dal loro massimo, tanti più elementi dei vettori resteranno. Tuttavia a meno di casi estremi questo consumo è accettabile.

Viceversa per codici regolari questa modifica non avrà nessun effetto dal punto di vista della memoria.

Tutti i vettori e i parametri presentati in questo paragrafo sono pre-calcolati da uno script Matlab (generateDecodingStructs.m).

Il Codice 4.4 evidenzia l'algoritmo con il quale si calcolano Hbn e Hcn.

Un secondo script, "CodFileGenerator.m", mette insieme tutte le informazioni necessarie in un unico file binario con estensione .cod che viene poi caricato dal programma principale.

Il file binario è creato semplicemente scrivendovi, uno dopo l'altro, tutti i parametri e i vettori appena descritti. La funzione che carica in memoria il file, si limita a leggerne il contenuto partendo dai parametri fondamentali, utilizzati dalla funzione stessa per delimitare all'interno del file i singoli vettori, per poi copiare questi ultimi nella memoria globale della scheda video.

Il file così ottenuto ha dimensioni modeste: quello creato per il codice LDPC del DVB-S2 normal FECFRAME e rate 1/2 ha una dimensione complessiva di circa 4,5 MB. Dato che la dimensione del file è circa pari a quella occupata dai vettori nella memoria globale, significa che questi ne consumano meno dell'1% nonostante le dimensioni della matrice.

```
//Creazione di Hbn
Hbn = -1*ones(1,M*wcmax)      //vettore di M*wcmax elementi tutti a -1
cnt = zeros(1,N)              //vettore di N elementi tutti a 0
for(m=0;m<M;m++){
  idx=0
  for(n=0;n<N;n++){
    if(H(m,n)==1){            //elementi della matrice H
      Hbn[wcmax*m+idx] = wbmax*n+cnt(n)
      cnt(n)++
    }
    idx++
  }
}

//Creazione di Hcn
for(i=0;i<M*wcmax;i++){
   if(Hbn[i]!=-1){
      Hcn[Hbn[i]] = i;
   }
}
```

Codice 4.4: Algoritmo per calcolare il contenuto di Hcn e Hbn

4.2.2 Schema a blocchi e filosofia

Nel paragrafo introduttivo e nella Figura 4.1 è già stata introdotta la struttura del sistema realizzato.

L'intera catena di blocchi è stata realizzata principalmente per eseguire delle simulazioni, oltre che la semplice decodifica; in base alle necessità si possono abilitare o disabilitare i blocchi.

Il campo "configFlags" deve essere riempito con una parola creata utilizzando alcuni flag, dichiarati nel file "LDPC_constants_macros.h". Questi vanno combinati in-

sieme attraverso l'operatore "or" in modo da creare una parola che indichi alle altre funzioni come inizializzare il sistema.

L'intero sistema è in grado di elaborare più parole di codice contemporaneamente. Questo, come sarà approfondito in seguito, permette ai thread di accedere alla memoria in modo più regolare ottenendo la coalescenza.

Il trasferimento di informazioni dal Device e l'Host costituisce il principale collo di bottiglia del sistema, perciò si è scelto di mantenere il più a lungo possibile le informazioni sul Device, dove vengono elaborate. In questo modo i buffer principali si trovano nel Device, mentre nell'Host esistono solo delle copie che possono essere aggiornate opzionalmente nel caso in cui sistema operativo e schema a blocchi dovessero interagire.

Per esempio, se si vuole simulare un sistema di telecomunicazioni, l'utente o l'Host non hanno bisogno di accedere direttamente ai dati trasmessi, ma solo al valore finale di Bit Error Rate, pertanto non è necessario nessun trasferimento e quindi nessuna allocazione sull'host.

Viceversa un sistema embedded che esegue solo la decodifica di dati provenienti da un'ipotetica interfaccia radio, dovrà prima trasferirli al Device, elaborarli, e in seguito trasferire il risultato in una qualche zona di memoria in modo che un'altra applicazione possa utilizzarli.

L'utilizzatore della libreria è libero, quindi, di allocare i buffer sulla memoria Host come crede ed usare le tecniche più appropriate per il trasferimento di questi dati ove necessario, mentre i buffer sul Device devono essere comunque allocati per il funzionamento del sistema.

I buffer contengono dati in formati diversi, a seconda del punto in cui sono posti all'interno della catena.

I buffer "u" e "hat u" contengono dati senza nessun tipo di formattazione; questi, infatti, potrebbero provenire direttamente da un'applicazione e quindi non possono subire nessun tipo di modifica.

È stato pensato un nuovo formato, detto "fat bit", che prevede l'utilizzo di un intero byte per rappresentare un singolo bit. Questo è giustificato dal fatto che a livello di codifica di canale l'unità fondamentale di memoria è il singolo bit, in questo modo è possibile accedervi senza complicate operazioni di mascheramento ed estrazione. Questa rappresentazione costituisce uno spreco di spazio in memoria che, tuttavia, è trascurabile rispetto alla quantità di memoria globale disponibile.

I buffer "c" ed "hat c" contengono dati in questo formato.

Infine tutti gli altri buffer contengono valori reali che vengono rappresentati con il tipo di dati a virgola mobile (float).

Per i buffer che possono essere copiati da e verso l'Host, per generalità i bit sono memorizzati in ordine di "arrivo", ovvero dal primo all'ultimo ricevuto.

Per gli altri buffer, utilizzati internamente dai singoli blocchi, non esiste questo vincolo. Per esempio, nel paragrafo 5.4.1 sarà illustrato come il decoder esegua, all'inizio e alla fine dell'algoritmo, un riordinamento dei bit all'interno dei vettori che permette di sfruttare meglio la coalescenza.

I buffer nel Device hanno necessariamente una lunghezza fissa per via del funzionamento del sistema; non c'è nessun requisito particolare, invece, per la lunghezza dei buffer nell'Host, in quanto questi vengono gestiti nel programma principale.

Le dimensioni dei buffer dipendono dal formato dei dati, dalle dimensioni del codice LDPC e dal numero di codeword elaborate in parallelo. Ad esempio il buffer "y" che contiene i dati in formato float (4 byte), supponendo che il codice LDPC abbia N=1024 e vengano elaborate 32 parole in parallelo, avrà una dimensione pari a $32 \times 1024 \times 4 = 64KB$.

Nel caso di dati in formato binario reale, i K bit di una parola vengono memorizzati senza nessuna formattazione utilizzando il minor spazio possibile; da notare che se K non è un multiplo di 8, l'ultimo byte sarà occupato solo parzialmente e gli ultimi bit non saranno considerati. In questo modo il parametro KinByte risulta essere pari a $\left\lceil \frac{K}{8} \right\rceil$ analogamente a MinByte e NinByte.

La struttura dati "systemCtrl" è predisposta in modo da contenere i puntatori a tutti i buffer di sistema.

4.2.3 Le API

Per interfacciarsi alla libreria sviluppata è necessario conoscere le API (Application Programming Interface).

Queste interfacce sono raccolte in tre header file: LDPC.h, LDPC_constants_macros.h e LDPC_parameters.h; nella scrittura di un programma in C è sufficiente includere il primo file.

LDPC.h contiene la dichiarazione di tutte le funzioni che possono essere richiamate dal programma principale: una serie di funzioni utili in fase di debug che stampano

a schermo il contenuto dei buffer di sistema, le funzioni di inizializzazione del sistema e quelle che il programma principale richiama per eseguire l'elaborazione dei vari blocchi.

LDPC_constants_macros.h contiene la definizione delle strutture CodeSpecs e systemCtrl, varie costanti e flag utilizzati da altre porzioni di codice, ed infine alcune macro che eseguono funzioni utili: in particolare ve ne sono alcune per l'utilizzo dei timer.

```
struct systemCtrl{
	//SYSTEM VARIABLES, CONFIGURATIONS AND OBJECTS//
	int bitSent;		//bit sent (BER statistics)
	int wrongBits;		//wrong decoded bits (BER statistics)
	FILE *debugFile;	//debug output file
	int configFlags;	//configuration word
	curandGenerator_t sg; //random number generator
	//SYSTEM MAIN BUFFER// //h_ = host, d_ = device, hat = decoded version
	//source
	char *h_uBuf;		//info bits
	char *d_uBuf;		//info bits
	//encoder
	char *d_GuBuf; //IRA specific
	char *d_cBuf;  //encoded bits
	char *h_cBuf;  //encoded bits
	//AWGN
	float *h_yBuf; //soft bits
	float *d_nBuf; //noise buffer
	float *d_yBuf; //soft bits
	//DECODER
	//float decoder
	float *d_PiBuf;		//a priori
	float *d_rjiBuf;
	float *d_qijBuf;
	//8bit decoder
	char *d_8bPi;
	char *d_8bq;
	char *d_8br;
	//16bit decoder
	int16_t *d_16bPi;
	int16_t *d_16bq;
	int16_t *d_16br;
	//decoder Common
	char *d_hatcBuf;
	char *d_hatuBuf;
	char *h_hatuBuf;};
```

Codice 4.5: definizione della struttura systemCtrl

LDPC_parameters.h, infine, contiene alcuni parametri utili per fare il tuning del sistema o per cambiare alcune configurazioni.

Agendo sulle costanti definite come direttive al compilatore, si possono cambiare il numero di parole elaborate in contemporanea (CDW_IN_PARALLEL), cambiare il numero di thread in un thread block, abilitare l'utilizzo della surface o texture memory o cambiare il numero di iterazioni che esegue il decoder.
Si possono inoltre abilitare alcune funzioni di debug che stampano, a schermo o su file, il contenuto delle variabili principali.

Il criterio con il quale sono stati scelti i valori di default di questi parametri è discusso nel capitolo 5.

Per inizializzare il sistema occorre allocare in memoria, sia dell'Host che del Device, tutti i vettori e i buffer di cui si ha bisogno.

Per prima cosa è opportuno caricare dal file .cod le informazioni riguardanti il codice LDPC, in quanto le successive inizializzazioni si baseranno su di esso.
La funzione LDPC_loadCodeFromFile esegue questo compito, prima riempiendo i vari campi della struttura dati CodeSpecs con i parametri già accennati, in seguito allocando nella Global Memory i vettori necessari e copiandone il contenuto dal file ad essa.

Completato questo compito si può procedere all'inizializzazione vera e propria dei buffer: la loro allocazione avviene basandosi sulla parola di configurazione "configFlags" presente nella struttura systemCtrl che va impostata opportunamente come OR dei flag corrispondenti ai blocchi che si vogliono attivare.

In seguito la funzione LDPC_system_dev_init si occupa di allocare tutti i buffer sulla memoria del device.

Per allocare i buffer nell'Host, si può utilizzare in modo simile la funzione LDPC_system_host_init, oppure gestirli manualmente in base alle necessità come già accennato precedentemente.

Il Codice 4.6 mostra un esempio in cui viene inizializzato un sistema con tutti i blocchi attivati per poter eseguire una simulazione completa.

Le funzioni che il programma principale richiamerà per elaborare i dati, corrispondono ai blocchi di cui è composto il sistema: LDPC_RandomSource_Simulate, LDPC_encode, LDPC_simulateAWGN, LDPC_decode, LDPC_BERcollectData e LDPC_BERresetData; queste saranno analizzate con maggior dettaglio nei paragrafi seguenti.

Il Codice 4.7 mostra come utilizzare queste funzioni per eseguire una simulazione della catena di trasmissione e ricezione.

```
CodeSpecs code;
systemCtrl ctrl;

//CARICAMENTO DEL CODICE LDPC
LDPC_loadCodeFromFile(&code,"..\\..\\CodeFiles\\DVBS2_1_2.cod");

//CONFIGURAZIONE E ABILITAZIONE DEI BLOCCHI
ctrl.configFlags = RNDSRC|ENC|AWGN|DEC|BER;  //tutti, decoder float

//CONFIGURAZIONE DELL'OUTPUT FILE PER IL DEBUG
ctrl.debugFile = stdout;      //debug stampato su schermo

//SYSTEM INITIALIZATION, x>>20 = x/1024/1024 = da B a MB
printf("Allocati %d MB sul Device\n",LDPC_system_dev_init(&code,&ctrl)>>20);
```

Codice 4.6: Esempio di inizializzazione di sistema per la simulazione

```
float simulate(float EbN0dB,int iterations){
        LDPC_BERresetData(&ctrl,&code);
        for(int it=0;it<interations;it++){
                LDPC_RandomSource_Simulate(&ctrl,&code);
                LDPC_encode(&ctrl,&code);
                LDPC_simulateAWGN(&ctrl,&code,EbN0dB);
                LDPC_decode(EbN0dB,&ctrl,&code);
                LDPC_BERcollectData(&ctrl,&code);
        }
        return 1.0*ctrl.wrongBits/ctrl.bitSent;      //bit error rate
}

int main(int argc,char **argv){
        //INIZIALIZZAZIONE
        ...
        float BER = simulate(1.5,100);
        ...
}
```

Codice 4.7:Esempio di utilizzo delle API per trovare la Bit Error Rate a un certo rapporto segnale rumore

Per valutare le prestazioni del sistema così realizzato è opportuno eseguire misure precise sul tempo di esecuzione delle varie parti. Sono state realizzate alcune macro che permettono l'uso dei timer messi a disposizione da CUDA; questi timer non misurano il tempo usando le risorse dell'Host, bensì usano dei contatori situati sulla scheda video stessa: in questo modo il tempo misurato non è affetto da overhead dovuti al sistema operativo.

Tuttavia per eliminare ogni contaminazione della misura da eventi aleatori, è consigliabile ripetere l'esecuzione più volte misurando il tempo totale. Per creare un nuovo timer, si può scrivere TIMER_START(nome_del_timer), per avviarlo e fermarlo TIMER_START(nome_del_timer) e TIMER_STOP(nome_del_timer), ed infine TIMER_GET_MS(nome_del_timer) e TIMER_GET_S(nome_del_timer) per recuperare il tempo cronometrato rispettivamente in millisecondi e in secondi.

Il Codice 4.8 costituisce un esempio completo del codice del programma principale per eseguire la simulazione di tutta la catena, con relative misure del tempo di esecuzione delle varie parti e il relativo throughput massimo. Il throughput è sempre calcolato rispetto ai bit di informazione piuttosto che quelli codificati, perciò per un generico blocco di elaborazione che esegue il suo compito in T secondi il throughput si calcola usando la seguente formula

$$Throughput = \frac{bit\ elaborati}{T} = \frac{K \times CDW_IN_PARALLEL}{T} \tag{4.2}$$

```
#include<stdio.h>
#include<stdlib.h>
#include<LDPC.h>

CodeSpecs code;
systemCtrl ctrl;

int main(int argc,char **argv){
        char *fileName = argv[1];
        int iterations = atoi(argv[2]);
        float EbN0dB = atof(argv[3]);

        //INIZIALIZZAZIONE
        TIMER_INIT(total)
        TIMER_INIT(src)
        TIMER_INIT(enc)
        TIMER_INIT(awgn)
        TIMER_INIT(decoder)
        TIMER_INIT(ber)
        LDPC_loadCodeFromFile(&code,fileName);
        ctrl.configFlags = RNDSRC|ENC|AWGN|DEC|BER;
        LDPC_system_dev_init(&code,&ctrl);
        LDPC_BERresetData(&ctrl,&code);

        //START
        TIMER_START(total)
```

```
        TIMER_START(src)
        for(int it=0;it<iterations;it++) LDPC_RandomSource_Simulate(&ctrl,&code);
        TIMER_STOP(src)

        TIMER_START(enc)
        for(int it=0;it<iterations;it++) LDPC_encode(&ctrl,&code);
        TIMER_STOP(enc)

        TIMER_START(awgn)
        for(int it=0;it<iterations;it++) LDPC_simulateAWGN(&ctrl,&code,EbN0dB);
        TIMER_STOP(awgn)

        TIMER_START(decoder)
        for(int it=0;it<iterations;it++) LDPC_decode(EbN0dB,&ctrl,&code);
        TIMER_STOP(decoder)

        TIMER_START(ber)
        for(int it=0;it<iterations;it++) LDPC_BERcollectData(&ctrl,&code);
        TIMER_STOP(ber)

        TIMER_STOP(total)

        printf("FINAL STATISTICS:\n");
        printf("Total Bit Sent: %d\n",ctrl.bitSent);
        printf("Wrong Bits: %d\n",ctrl.wrongBits);
        printf("@ Eb/N0 = %f [dB], BER =
%f\n\n",EbN0dB,1.0*ctrl.wrongBits/ctrl.bitSent);
        PRINT_SEPARATOR
        printf("BLOCK\t|\tLATENCY\t\t|\tTHROUGHPUT\n");
        printf("src\t|\t%0.1f ms\t\t|\t%0.3f
Mbps\n",TIMER_GET_MS(src)/iterations,ctrl.bitSent/(1000.0*TIMER_GET_MS(src)));
        printf("enc\t|\t%0.1f ms\t\t|\t%0.3f
Mbps\n",TIMER_GET_MS(enc)/iterations,ctrl.bitSent/(1000.0*TIMER_GET_MS(enc)));
        printf("awgn\t|\t%0.1f ms\t\t|\t%0.3f
Mbps\n",TIMER_GET_MS(awgn)/iterations,ctrl.bitSent/(1000.0*TIMER_GET_MS(awgn)));
        printf("dec\t|\t%0.1f ms\t|\t%0.3f
Mbps\n",TIMER_GET_MS(decoder)/iterations,ctrl.bitSent/(1000.0*TIMER_GET_MS(decoder))
);
        printf("ber\t|\t%0.1f ms\t\t|\t%0.3f
Mbps\n",TIMER_GET_MS(ber)/iterations,ctrl.bitSent/(1000.0*TIMER_GET_MS(ber)));
        printf("-------------------------------------------------------------\n");
        printf("total\t|\t%0.1f ms\t|\t%0.3f
Mbps\n",TIMER_GET_MS(total)/iterations,ctrl.bitSent/(1000.0*TIMER_GET_MS(total)));
        //END
        LDPC_system_free(&code,&ctrl);
        LDPC_code_free(&code);
        cudaDeviceReset();
        return 0;
}
```

Codice 4.8: Programma utilizzato per misurare il throughput dei vari blocchi

```
#include<stdio.h>
#include<stdlib.h>
#include<LDPC.h>

CodeSpecs code;
systemCtrl ctrl;

int main(int argc,char **argv){
        char *fileName = argv[1];
        float EbN0dB = atof(argv[2]);

        //INIZIALIZZAZIONE
        LDPC_loadCodeFromFile(&code,fileName);
        ctrl.configFlags = DEC;
        float *input = (float *)malloc(CDW_IN_PARALLEL*code.N*sizeof(float));
        char *output = (char *)malloc(CDW_IN_PARALLEL*code.KinByte);
        LDPC_system_dev_init(&code,&ctrl);

        for(;;){
                TIMER_INIT(total)
                TIMER_START(total)

                //from input to device
                cudaMem-
cpy(ctrl.d_yBuf,input,CDW_IN_PARALLEL*code.N*sizeof(float),cudaMemcpyHostToDevice);

                //computation
                LDPC_decode(EbN0dB,&ctrl,&code);

                //from device to output
                cudaMem-
cpy(output,ctrl.d_hatuBuf,CDW_IN_PARALLEL*code.KinByte,cudaMemcpyDeviceToHost);

                TIMER_STOP(total); float Time = TIMER_GET_MS(total);

                printf("Latency: %.3f ms, Throughput: %.3f
Mbps\n",Time,code.K*CDW_IN_PARALLEL/Time/1000.0);
        }

        //END
        LDPC_code_free(&code);
        LDPC_system_free(&code,&ctrl);
        cudaDeviceReset();

        return 0;
}
```

Codice 4.9: Esempio di decoder real-time embedded

Il Codice 4.9, invece, mostra un esempio di un programma che esegue la decodifica dei dati provenienti da un ipotetica interfaccia radio. Nell'esempio si suppone che i vettori "input" e "output" siano gestiti dal sistema operativo (o dal DMA) in modo da riempire ciclicamente il primo con i dati reali provenienti dall'interfaccia, e indirizzare i dati nel secondo alle applicazioni in corso sul sistema che ne hanno fatto richiesta.

4.3 I blocchi di elaborazione in dettaglio

In questo paragrafo è analizzato il funzionamento di ogni blocco di elaborazione. La descrizione completa dei kernel implementati può essere trovata nell'allegato 7.2.

4.3.1 Generatore di bit Casuali

La funzione che genera i dati casuali per la simulazione della catena è LDPC_RandomSource_Simulate.
Questa funzione utilizza la libreria cuRand per generare i CDW_IN_PARALLEL blocchi di K bit da trasmettere, ovvero genera esattamente $KinByte \times CDW_IN_PARALLEL = \left\lceil \frac{K}{8} \right\rceil \times CDW_IN_PARALLEL$ byte casuali direttamente nel buffer "u" in Global Memory.

Dato che la funzione curandGenerate della libreria citata può generare solo numeri interi casuali (4 byte), è necessario che il numero di byte totali da generare sia multiplo di 4.

4.3.2 Encoder

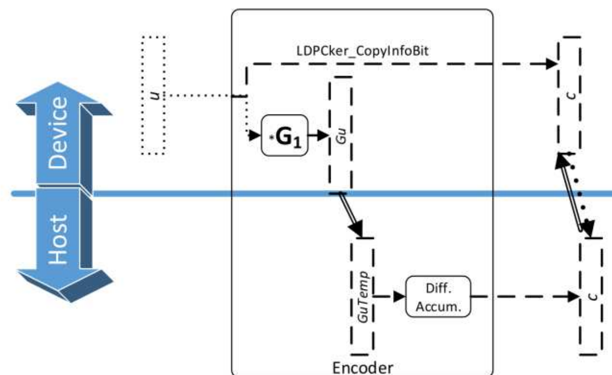


Figura 4.3: Schema a blocchi dell'encoder

La funzione LDPC_encode esegue la codifica LDPC dei dati contenuti nel buffer "u" in Global Memory.

In base al tipo di codice LDPC caricato, verranno eseguiti algoritmi diversi: nel caso di codici IRA sistematici, come illustrato nel paragrafo 2.2.1, verrà eseguita per prima una moltiplicazione matriciale tra il vettore "u" e la matrice $\boldsymbol{G_1}$ ponendo il risultato nel vettore "GuBuf".

Questa operazione è compiuta dal kernel "LDPCker_enc_LimitedMatrixMul" che esegue la moltiplicazione matriciale tra un vettore di numeri binari e una matrice sparsa memorizzata per righe, limitandosi però a considerare solo alcuni degli '1' di ogni riga della matrice. In precedenza è già stato discusso come considerare i primi *a* '1' della matrice $\boldsymbol{H}$ (dove *a = wcmax-2*) corrisponda a considerare la sola matrice $\boldsymbol{G_1}$, come richiesto dall'algoritmo di codifica per i codici IRA.
In seguito si esegue l'operazione di accumulazione differenziale, che consiste nell'operazione descritta nelle formule (2.19) e (2.20).

Dato che si tratta di un procedimento sequenziale, è opportuno eseguirla sull'Host piuttosto che sul Device, in quanto non ottimizzato per questo tipo di operazioni, quindi il vettore GuBuf viene copiato sull'Host e il risultato scritto sulla versione Host del buffer "c" che viene poi copiato nuovamente sul Device.
Infine l'intero contenuto del buffer "u" viene copiato, convertito nel formato "fat bit" e trasferito anch'esso nei primi *K* byte del buffer "c", completando così il calcolo della parola di codice.

4.3.3 Simulatore di canale

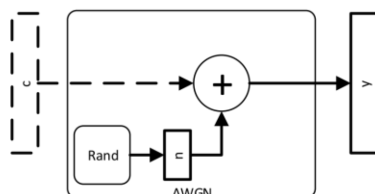


Figura 4.4: Schema a blocchi del simulatore di canale

La funzione LDPC_simulateAWGN si limita ad applicare il rumore gaussiano bianco ai bit di codice da trasmettere. La quantità di rumore è definita specificando il rapporto segnale rumore in decibel (E_b/N_0).

Dato che viene applicata una modulazione BPSK, ovvero ogni bit di codice è convertito in un unico simbolo pari a ± 1, l'energia per simbolo è pari all'energia per bit che a sua volta è pari a 1, quindi si può calcolare l'energia per simbolo del rumore utilizzando la seguente formula

$$N_0 = \frac{1}{10^{\frac{E_b/N_0[dB]}{10}}} \tag{4.3}$$

La sua radice quadrata corrisponde alla deviazione standard del rumore, valore che va fornito alle API della libreria CUDA per la generazione dei numeri casuali, per ottenere una realizzazione di un processo aleatorio gaussiano.
Rumore e versioni modulate delle parole di codice vengono sommate insieme e trasferite nel buffer "y".

4.3.4 Decoder

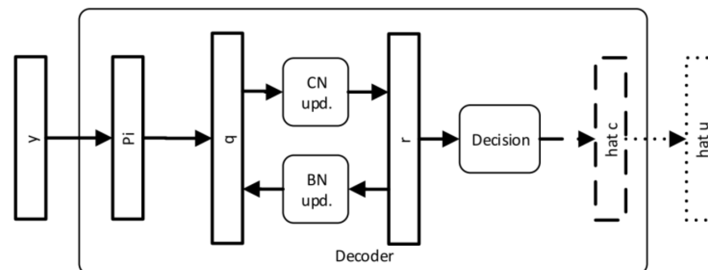


Figura 4.5: Schema a blocchi del decoder (float)

La funzione LDPC_decode, esegue la decodifica.

Attualmente sono state implementate tre versioni dell'algoritmo MSA, che differiscono per il tipo di dati utilizzato nei calcoli: una versione lavora con variabili float (32 bit), una versione con variabili intere a 16 bit, e l'ultima con variabili intere a 8 bit.
Il motivo di queste tre versioni è legato alla sperimentazione per ottimizzare le prestazioni, discorso che verrà approfondito nel capitolo successivo; tuttavia la versione attuale del progetto è funzionante solo con il decoder a valori float.
Gli altri due decoder soffrono del problema della saturazione delle variabili interne, ovvero, nel corso delle iterazioni, il valore assoluto da ogni variabile tende ad aumentare fino a raggiungere il massimo previsto dalla rappresentazione binaria utilizzata; questo fenomeno fa sì che la parola di codice non venga decodificata correttamente.

I tre decoder, comunque, differiscono unicamente per alcuni dettagli implementativi mentre il funzionamento di base è lo stesso.

Nella fase di inizializzazione un kernel provvede, partendo dal segnale ricevuto nel buffer "y", a calcolare le probabilità a priori, come illustrato nella formula (2.32), e a posizionarle nel vettore "Pi".

Questi valori vengono replicati e riportati anche nel vettore “q” che contiene i messaggi dai nodi variabile ai nodi di controllo e che nella prima iterazione sono uguali alle sole probabilità a priori.

Nelle versioni a 8 e 16 bit i valori provenienti dal canale vengono adattati alla rappresentazione come numero intero, e al nuovo range $[-128; 127]$ per gli 8 bit e $[-32768; 32767]$ per i 16 bit, con la moltiplicazione per un fattore di scala.

La moltiplicazione per un fattore di scala non modifica il comportamento dell’algoritmo, in quanto per le operazioni di addizione e confronto, le uniche utilizzate nei kernel, la rappresentazione intera equivale a una rappresentazione a virgola fissa che mantenga la stessa magnitudine degli equivalenti valori float.

In seguito un ciclo for esegue un numero pari alla costante MAX_ITERATIONS, definita in LDPC_parameters.h, di iterazioni composte dai due kernel che eseguono il processing orizzontale e verticale. La nomenclatura di questi kernel si riferisce al fatto che vengono aggiornati i nodi variabile o i nodi di controllo e quindi contengono la dicitura “CNupdate” o “BNupdate”.

Questi due kernel costituiscono la parte di software più importante di tutto il sistema, in quanto la scheda video passa la maggior parte del tempo ad eseguire questa porzione di codice, e quindi, di fatto, limitando il throughput di sistema.

Completato il ciclo un ultimo kernel, calcola i valori di $LQ_n^{(l)}$ come mostrato nella formula (2.36) e, utilizzando la formula (2.37), calcola il valore dei bit più verosimili ponendoli nel buffer “hat_c”.

Nel caso in cui il codice sia sistematico, i primi K bit di “hat_c” vengono trasferiti e convertiti nel formato bit “reale” nel buffer “hat_u” completando così la decodifica. Nel caso di codici non sistematici non è stata implementata nessuna decodifica, anche se è stato predisposto nella struttura dati codeSpecs il puntatore a un vettore “extras” che potrebbe contenere, in futuro, la matrice inversa di $\boldsymbol{G}$ necessaria per la decodifica.

4.3.5 Calcolatore di Bit Error Rate

La funzione LDPC_BERcollectData permette di calcolare il numero di errori di decodifica e quindi la Bit Error Rate.

Per fare questo, è necessario confrontare il contenuto dei buffer “u” e “hat_u” e contare il numero di bit per i quali differiscono.

Per trovare i bit diversi tra le due parole, si può eseguire l'operazione XOR bit a bit e quindi contare il numero di '1' del risultato, infatti lo XOR tra due bit è pari a 0 se sono uguali e 1 se diversi; il kernel LDPCker_ber_bitwiseXor si occupa di questa fase preliminare.

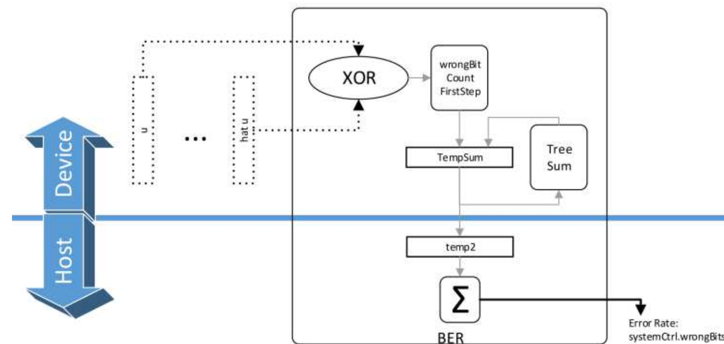


Figura 4.6: Schema a blocchi del calcolatore di BER

Nell'ambiente CUDA un'operazione come contare il numero di '1', o in generale analizzare un vettore per calcolare un singolo valore , è detta riduzione. Per risolvere questo tipo di problemi efficacemente tramite l'uso di un'architettura parallela, è opportuno eseguire un ciclo dove ad ogni iterazione si riduce il numero di elementi con una gerarchia ad albero; la Figura 4.7 illustra graficamente il concetto.

Un primo kernel, LDPCker_ber_wrongBitCountFirstStep, per ogni byte di memoria del risultato dello XOR, conta il numero di '1' e lo inserisce in un vettore ausiliario.

In seguito viene eseguito un ciclo, dove LDPCker_ber_treeSum esegue questa riduzione re-inserendo il risultato della somma di due elementi nel vettore ausiliario. Raggiunto un certo numero di iterazioni, il calcolo passa sulla CPU in quanto se il numero di thread è troppo ridotto la GPU non viene sfruttata al 100% e l'esecuzione diventa più lenta rispetto alla CPU.

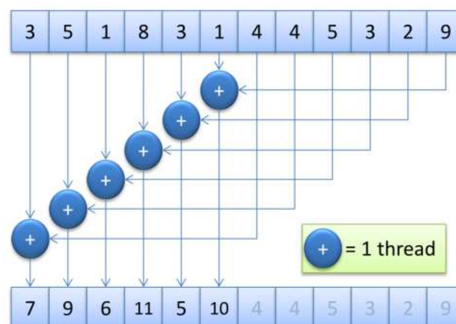


Figura 4.7: Principio di funzionamento delle operazioni di riduzione

Alla fine del procedimento al primo indirizzo del vettore ausiliario si troverà la somma di tutti gli elementi e quindi il numero di errori totale. Questo valore, insieme a il numero di bit trasmessi in totale, viene inserito nei corrispettivi campi della struttura “systemCtrl”; la Bit Error Rate è semplicemente il rapporto tra i bit sbagliati e il numero di bit totale.

La funzione LDPC_BERresetData, azzera questi due campi in modo da poter iniziare da capo un nuovo conteggio.

CAPITOLO 5

Risultati e Misure

5.1 Introduzione e tool utilizzati

Nel capitolo precedente è stata illustrata la struttura del progetto realizzato, tuttavia per raggiungere questo risultato è stata necessaria un'attenta esplorazione di più strategie per capire quali di queste sarebbero dovute essere utilizzate per soddisfare i requisiti nel modo più efficiente possibile.

Nel corso di questo capitolo sono illustrati gli strumenti e le tecniche utilizzate per misurare l'efficienza nell'esecuzione, in particolare della decodifica; in seguito sono esplorate diverse soluzioni per superare le limitazioni dell'hardware e per sfruttare al meglio le sue potenzialità, ed infine sono riportate le prestazioni ottenute discutendole e confrontandole con lavori di altri autori.

5.2 Gli strumenti utilizzati

La macchina sulla quale è stato sviluppato il progetto è dotata di una CPU Intel i3-3240 e una scheda video Nvidia GeForce GT620.

Quest'ultima ha Compute Capability 2.1, è dotata di 96 CUDA core e 1GB di memoria con una banda massima teorica di 14.4 GB/s.[4]

La scheda video è di tipo commerciale, e ha prestazioni decisamente inferiori rispetto alle schede di fascia più alta progettate per l'elaborazione parallela usate nei lavori di altri autori.

La macchina è dotata di sistema operativo Windows 8.1, mentre il progetto è stato sviluppato all'interno del framework Visual Studio 2010 con l'aggiunta dei vari strumenti di sviluppo messi a disposizione dall'Nvidia.

Uno dei più importanti è sicuramente Nsight: aggiunto a Visual Studio permette di eseguire il debug dei kernel direttamente sulla scheda video, permettendo al programmatore di analizzare facilmente l'esecuzione di thread diversi. Nsight, inoltre, è dotato di uno strumento che esegue il profiling dell'applicazione, ovvero che esegue una serie di esperimenti sul software progettato e misura, attraverso dei contatori, le prestazioni ottenute sotto vari punti di vista.

Esso è in grado di fornire informazioni dettagliate sull'utilizzo delle memorie e della cache, sul numero di branch eseguite e il loro esito, o statistiche sull'esecuzione delle istruzioni.

In modo similare un altro software, il Visual Profiler, è in grado di fornire statistiche sulle prestazioni ottenute, visualizzare graficamente l'esecuzione dei kernel nel tempo, ed evidenziare se questi soffrono di certi problemi, se possono essere migliorati ulteriormente e sotto quale aspetto.

Il Visual Profiler è stato utile soprattutto nelle fasi preliminari del progetto dei kernel, in quanto permette velocemente di rilevare problemi e di avere un'indicazione su dove è più opportuno dedicare tempo per migliorare l'esecuzione complessiva.

In Figura 5.1 è mostrata una rappresentazione temporale dell'esecuzione di una simulazione ottenuta con il Visual Profiler.

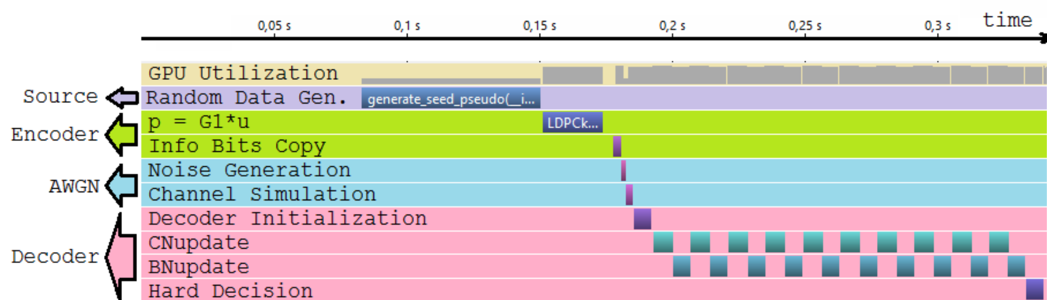


Figura 5.1: Visualizzazione grafica dell'esecuzione di una simulazione di trasmissione e ricezione nel tempo

Infine per caratterizzare le prestazioni finali, in particolare del decodificatore, sono stati usati i timer CUDA per misurare il tempo di esecuzione, o di latenza, di questo. Il throughput, valore riassuntivo della bontà dell'intero progetto, è calcolato a partire dalla latenza secondo la formula (4.2).

5.3 Figure di efficienza

Nel corso dello studio delle prestazioni del software realizzato, e nel corso dell'esplorazione di diverse soluzioni, si è rivelato subito necessario poter valutare l'efficienza del programma a prescindere dalla matrice di parità del codice LDPC utilizzato o, in generale, da variabili estrinseche.

Il throughput, che sia quello comprendente il tempo di esecuzione di tutti i blocchi o di un singolo blocco, in prima approssimazione, può dare un'idea di quanto efficacemente il sistema è in grado di elaborare i bit.

Il throughput del decoder, per esempio, è il rapporto tra i bit di informazione elaborati e la latenza dello stesso; è evidente che queste due grandezze sono legate dal fatto che una maggiore quantità di informazioni richiederà più tempo per essere elaborata, rendendo il loro rapporto circa costante.

Tuttavia il tempo di calcolo, come evidenzia la Tabella 2.1, dipende soprattutto dal numero di '1' nella matrice di parità, perciò risulterà che due codici LDPC con matrici di parità con un numero simile di '1' ma rate diverse saranno caratterizzate da tempi di decodifica simili ma throughput diversi: perciò questo valore non è indicativo dell'efficienza reale del calcolo.

Il tempo di calcolo per la decodifica è influenzato non solo dal numero di '1' nella matrice di parità, ma anche dal numero di iterazioni eseguite dall'algoritmo MSA e dal numero di parole di codice decodificate contemporaneamente.

Nei paragrafi successivi sarà discusso in dettaglio in che modo questi parametri influenzano il comportamento del software, tuttavia osservando tutti i tempi di esecuzione raccolti nel corso di questa indagine si è osservato che, definendo L_D latenza del decodificatore, C_P numero di parole elaborate in parallelo, E numero di '1' della matrice $\boldsymbol{H}$ ed I il numero di iterazioni, la quantità

$$l_D = \frac{L_D}{E \cdot C_P \cdot \left(I + \frac{3}{2}\right)} \qquad \forall C_P \geq 32 \tag{5.1}$$

è costante, e vale circa $1.94 \left[\frac{ns}{Edge \cdot Iteration}\right]$ con una deviazione standard, tra i vari casi, del 3.6%.

Si è deciso di chiamare questo valore **latenza specifica di decodifica** in quanto permette di valutare le prestazioni di questa a prescindere dal codice LDPC usato e dagli altri parametri.

Conoscendo questo valore è quindi possibile predire la latenza e il throughput del decodificatore usando le seguenti formule.

$$L_D \simeq l_D E C_P \left(I + \frac{3}{2}\right) \tag{5.2}$$

$$T_D = \frac{K C_P}{L_D} \simeq \frac{K}{l_D E \left(I + \frac{3}{2}\right)} = \frac{RN}{l_D E \left(I + \frac{3}{2}\right)} \tag{5.3}$$

Queste formule rappresentano una grossa approssimazione; tuttavia, studiare come si modifica la latenza specifica di decodifica, permette di valutare velocemente i benefici di una tecnica usata o se il sistema sta lavorando alla massima capacità.

5.4 Strategie usate e loro validazione

5.4.1 Coalescenza

In assoluto la coalescenza rappresenta la tecnica che permette di aumentare maggiormente l'efficienza di esecuzione in un'architettura parallela.

Il grosso problema, in questo caso, è che durante la decodifica si deve accedere ai vettori dei messaggi $Lq_{nm}^{(l)}$ ed $Lr_{nm}^{(l)}$ secondo gli indirizzi contenuti in Hbn e Hcn che sono irregolari.

È impossibile, dunque, che i thread in un warp accedano ad indirizzi consecutivi ed allineati come richiesto dalla tecnica.

La soluzione adottata consiste nell'elaborazione di più parole di codice in parallelo.

Il pattern di accesso ai messaggi è lo stesso per ogni parola di codice; disponendo consecutivamente in memoria messaggi con indici uguali ma associati a parole di codice diverse, e facendo sì che thread consecutivi dello stesso warp elaborino questi messaggi, si ottiene la coalescenza.

In Figura 5.2 è mostrato come cambia l'accesso alla memoria aumentando il numero di parole elaborate in parallelo.

In questo modo inevitabilmente aumenta la quantità di dati da elaborare e quindi anche il tempo di latenza, tuttavia la coalescenza permette al software di lavorare con un'efficienza maggiore e quindi di avere un throughput più elevato.

In Figura 5.3 è mostrata la dipendenza delle prestazioni dal numero di parole elaborate in parallelo, mentre nell'allegato 7.3 sono raccolti i valori ottenuti dal profiling del kernel LDPCker_dec_float_BNupdate (rappresentativo dell'intero decoder) nel caso di 4 e 32 parole di codice in parallelo.

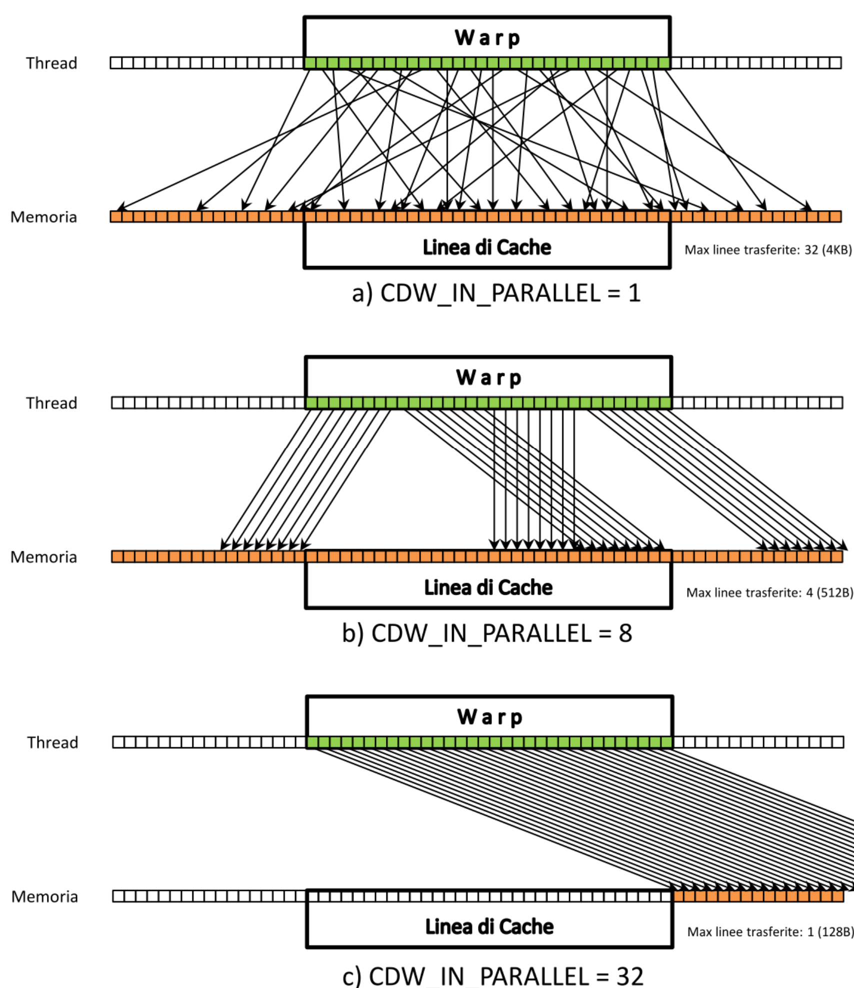


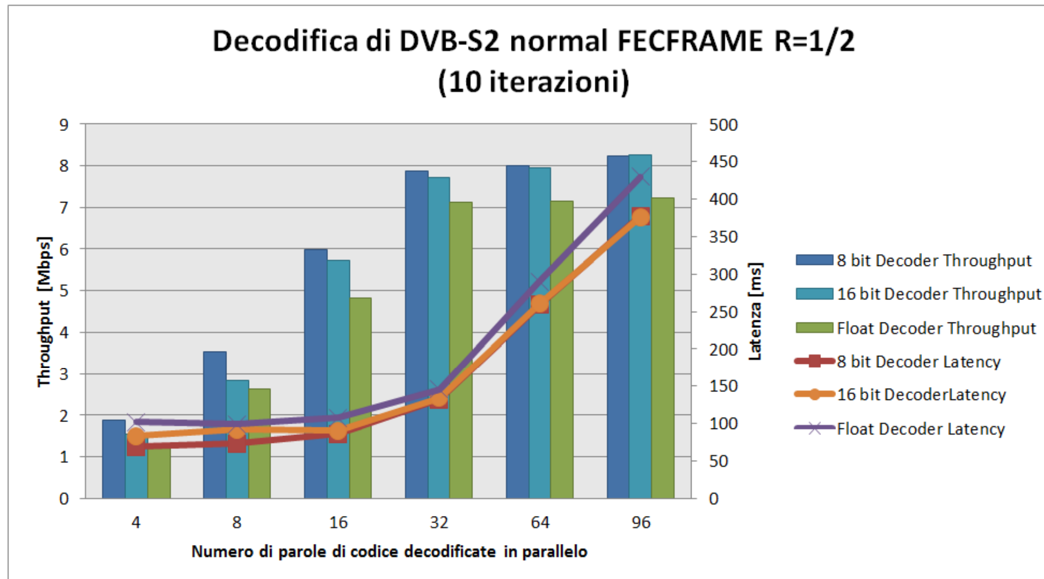
Figura 5.2: Pattern di accesso in memoria variando il numero di parole decodificate in parallelo, nel caso c si ha coalescenza

Nella Figura 5.3-b è mostrata, in particolare, la latenza specifica di decodifica che evidenzia come l'efficienza aumenti con l'aumentare del numero di parole decodificate in parallelo.

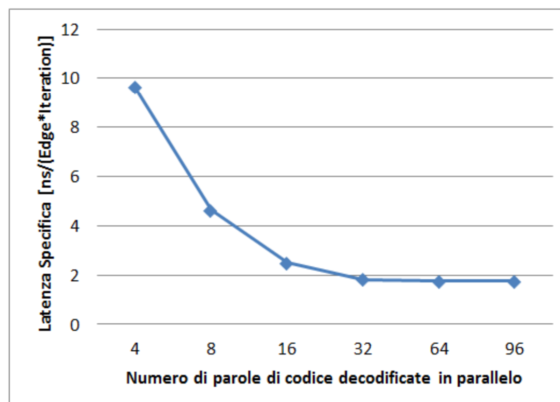
Quando il numero di parole è inferiore a 32, gli indirizzi ai quali thread consecutivi fanno accesso sono consecutivi a gruppi: abbastanza per far sì che la cache richiami meno linee, ma non abbastanza da avere un trasferimento per warp (Figura 5.2-b).

Quando il numero di parole raggiunge 32 allora gli accessi saranno tutti consecutivi all'interno del warp; nel caso del decoder float, la cache recupererà questi 32 valori, che coincidono con una singola linea da 128 byte, con un unico trasferimento (Figura 5.2-c).

Per un numero maggiore di parole decodificate in parallelo, non vi è un reale miglioramento, infatti il throughput aumenta solo in modo trascurabile per effetti del second'ordine, mentre la latenza aumenta in modo lineare.



a)



b)

Figura 5.3: Dipendenza delle prestazioni dal numero di parole elaborate in parallelo

Osservando l'allegato 7.3, in particolare i campi evidenziati, si può notare come nel caso di 4 parole in parallelo il numero di transizioni per richiesta sia notevolmente aumentato: questo a indicare che sono necessari più trasferimenti per recuperare tutte le informazioni necessarie ad un warp, in quanto sono sparpagliate a diversi indirizzi di memoria.

Nel caso di 32 parole, invece, ad ogni richiesta corrisponde una transizione, ovvero la linea di cache contenente i 32 valori float.

Osservando la Hit Rate si ha la conferma di come questo fatto incida sulle prestazioni: più questa è bassa, più vengono meno i benefici della cache.

Il parametro Replay Overhead, allo stesso modo, mostra come esistano alcuni indirizzi di memoria che devono essere caricati più volte in cache a causa del fatto che la linea di cache contiene informazioni necessarie a thread non consecutivi e in momenti diversi.

I kernel che eseguono l'inizializzazione e l'Hard Decision alla fine della decodifica si occupano anche di riordinare in questo modo i messaggi $Lq_{nm}^{(l)}$ ed $Lr_{nm}^{(l)}$ nei rispettivi vettori.

5.4.2 Memoria Texture e Surface

Nel corso di questo lavoro è stata esplorata la possibilità, per i decoder a 8 e 16 bit, di utilizzare la memoria Texture, in modo da ottenere un miglioramento delle prestazioni.

Dato che le Texture sono utilizzabili in sola lettura, si è pensato di memorizzare in questo modo solo le strutture dati relative alla matrice $\boldsymbol{H}$.

Trattandosi di vettori, le Texture così create, si svilupperanno in una sola dimensione.

I vettori dei messaggi, invece, sono stati memorizzati come Surface, in modo da poter essere anche scritti durante il funzionamento dei kernel; in particolare per sfruttare la località spaziale al meglio, sono stati memorizzati in due dimensioni.

Entrambe queste modifiche sono risultate vane, provocando un peggioramento delle prestazioni.

Questo peggioramento si verifica principalmente per due ragioni: la soluzione con memoria globale, cache di primo livello e coalescenza è già di per sé una soluzione molto efficiente che sfrutta bene le potenzialità dell'hardware, inoltre la memoria Texture ha tempi di accesso e velocità di bus persino peggiori di quella globale, ma avendo una cache dedicata e dell'hardware dedicato per eseguire accessi in più dimensioni, può mascherare questi difetti nel caso di applicazioni che eseguono più trasferimenti agli stessi indirizzi.

Tutti i kernel del decoder implementato accedono in lettura e scrittura solo una volta, perciò la memoria Texture appare essere complessivamente più lenta di quella Globale.

5.5 Tuning dei Parametri

5.5.1 Numero di parole elaborate in parallelo

Nel paragrafo 5.4.1 è stato illustrato come è necessario che il numero di parole di codice elaborate in parallelo sia almeno 32; tuttavia questa scelta dipende molto dall'applicazione che si vuole realizzare e dai requisiti che questa deve avere.

Quando il software deve essere molto reattivo, è necessaria una bassa latenza che può essere ottenuta elaborando poche parole in parallelo, al limite solo una, sacrificando l'efficienza dell'elaborazione.

Quando, invece, non esiste questo vincolo e l'obiettivo è elaborare una grande quantità di dati è consigliabile decodificare più parole in parallelo, in questo modo l'utilizzo della scheda video diventa sempre più efficiente: come mostra la Figura 5.3-a, infatti, il throughput tende ad aumentare, anche se solo lievemente, con l'aumentare di questo valore. Le simulazioni rientrano in questa tipologia di applicazioni.

In generale, considerando come applicazione un decoder real time, è opportuno raggiungere un buon compromesso tra latenza e throughput; in questo caso la scelta migliore è di elaborare esattamente 32 parole in contemporanea: l'efficienza e il throughput sono già prossimi al massimo, ma la latenza è ancora piuttosto bassa.

5.5.2 Configurazione dei thread

I kernel hanno un comportamento riassumibile con la lettura di dati, la loro elaborazione e la scrittura dei risultati in memoria. Per questo tipo di kernel è opportuno impostare un'alta Occupancy in modo da mascherare la latenza della memoria.

Per ottenere la massima Occupancy è stato usato l'Occupancy Calculator, messo a disposizione da Nvidia, in modo da trovare il numero ottimale di thread in un TB.

La Figura 5.4 mostra i grafici contenuti nell'Occupancy Calculator che evidenziano come 192 thread per TB siano un'impostazione che, potenzialmente, può risultare nel 100% di Occupancy.

Dato che, come già discusso, un'alta Occupancy può rivelarsi controproducente, è stato misurato il throughput del decoder in diverse configurazioni per confermare la validità del risultato ottenuto con l'Occupancy Calculator.

Il risultato, visualizzato nella Figura 5.5 insieme con l'Occupancy, mostra come 192 thread per TB sia effettivamente la configurazione migliore.

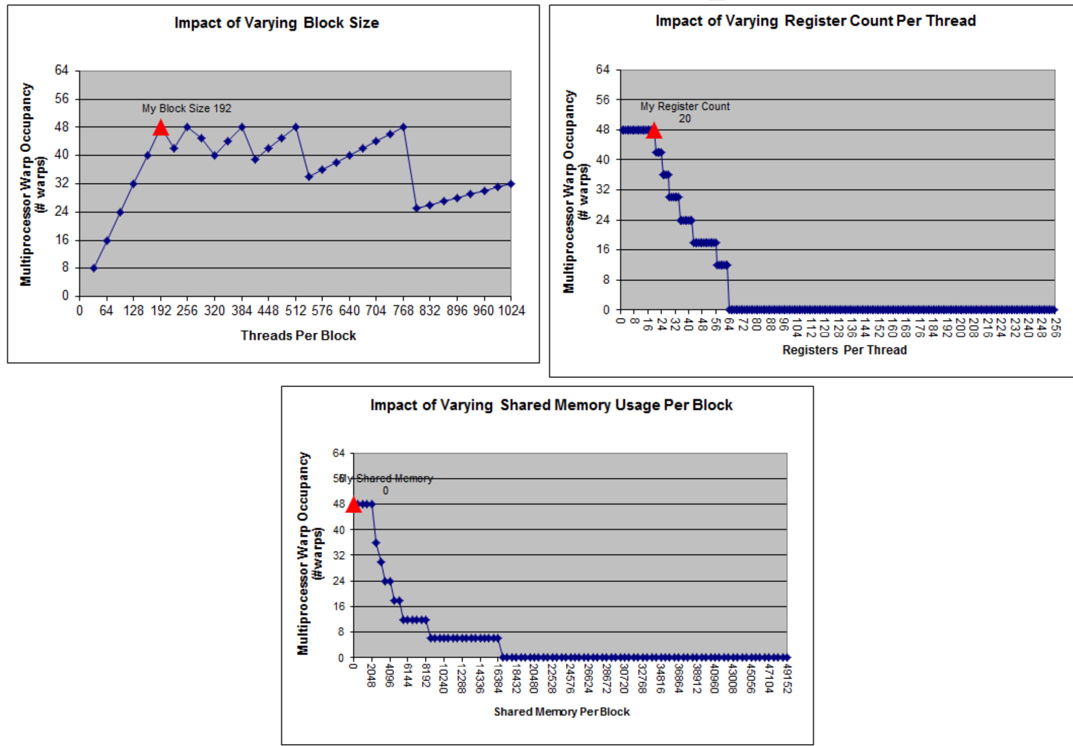


Figura 5.4: Grafici tratti dall'Occupancy Calculator che evidenziano le impostazioni ottimali per i kernel del progetto

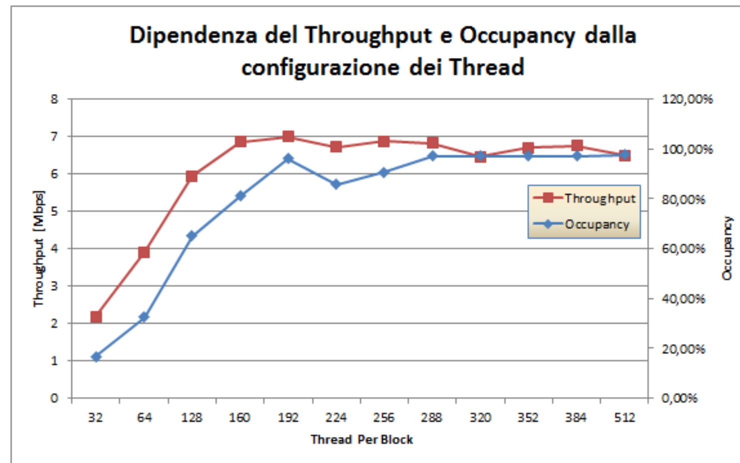


Figura 5.5: Dipendenza delle prestazioni dall'Occupancy e dalla configurazione dei thread

Inoltre appare evidente che il throughput ha un andamento simile a quello dell'Occupancy; questo fatto dimostra come questa applicazione sia limitata soprat-

tutto dalla memoria, e come, quindi, un'alta Occupancy migliori le prestazioni nascondendo i lunghi tempi di accesso.

5.5.3 Numero di Iterazioni del Decoder

Il numero di iterazioni eseguite dall'algoritmo di decodifica MSA, è l'unico parametro regolabile che non solo influisce sulle prestazioni del software, ma anche sulla precisione del decodificatore LDPC[13].

L'algoritmo MSA tende a convergere dopo un certo numero di iterazioni, a meno che il grafo di Tanner del codice non abbia degli anelli composti da quattro lati.

La situazione ideale consiste nell'avere un numero di iterazioni pari a quelle necessarie perché l'algoritmo converga, in modo da avere un risultato corretto nel minor tempo possibile.

Nel paragrafo 2.3.2 si definisce l'algoritmo MSA asserendo che esso debba terminare le iterazioni non appena l'equazione di parità $\boldsymbol{H}\hat{c} = 0$ sia verificata oppure una volta raggiunto un numero massimo di iterazioni.

Nella prima versione del decoder il kernel che si occupava di controllare il soddisfacimento di questa equazione impiegava un tempo non trascurabile per eseguire il compito, perciò si è deciso di eliminare questo controllo ed eseguire un numero fisso di iterazioni.

A meno di codici LDPC per i quali l'algoritmo non converge, questo sistema permette di avere il medesimo risultato ma in tempi mediamente inferiori, inoltre rende i tempi di latenza, e quindi i throughput, indipendenti dalle condizioni di lavoro, qualità apprezzabile per un sistema real time che deve essere prevedibile.

Resta il problema di tarare opportunamente il numero massimo di iterazioni. Per farlo bisogna conoscere il numero di iterazioni per cui l'algoritmo converge, valore difficile da calcolare che dipende dal codice LDPC e dalle condizioni di lavoro (rapporto segnale rumore).

Uno dei programmi sviluppati[14] consiste nell'esecuzione di un'intera simulazione a diversi valori di rapporto segnale rumore in modo da poter tracciare le curve BER.

È interessante osservare come queste curve si modificano con la variazione del numero di iterazioni (Figura 5.6).[15]

---

[13] Anche la precisione delle variabili (float, 16 bit, 8 bit) modificano la precisione del decoder ma non è considerata un parametro regolabile in quanto per ogni versione del decoder essa è fissata

[14] BER_curve.exe

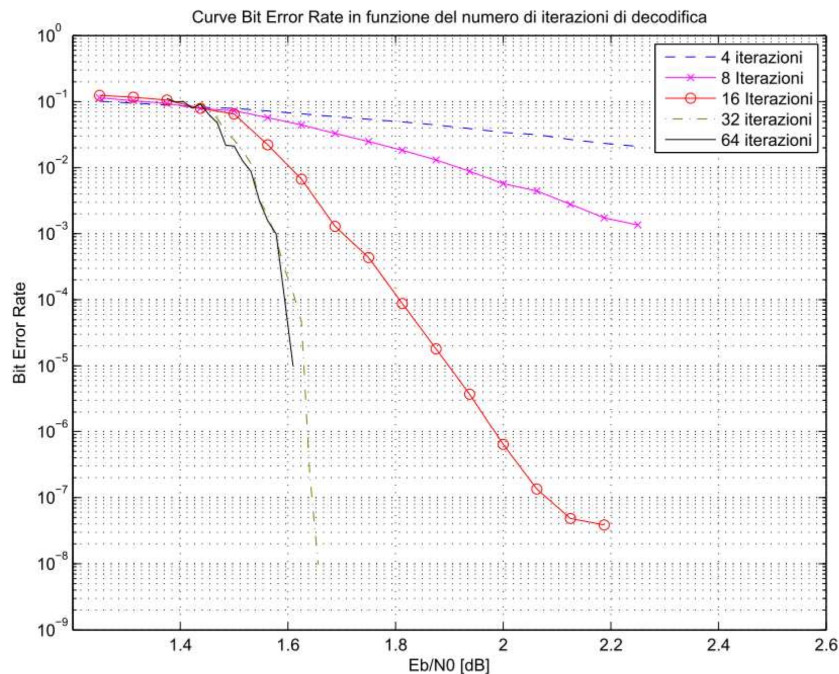


Figura 5.6: Variazione delle curve BER rispetto al numero di iterazioni (DVB-S2 normal FECFRAME Rate 1/2)

Osservando la figura si nota come aumentando il numero di iterazioni le curve tendano a diventare più ripide finché, raggiunte 20-30 iterazioni, queste coincidano.

Due curve coincidenti indicano che il numero di iterazioni impostato è maggiore di quello necessario perché l'algoritmo converga, perciò le curve ottenute corrispondono a quelle teoriche del codice LDPC. Negli altri casi, invece, l'algoritmo non riesce a convergere, risultando in numerosi errori di decodifica. Per questo motivo aumenta la probabilità di errore e quindi le curve tendono ad essere meno ripide.

La scelta del numero di iterazioni da eseguire dipende fortemente dalle prestazioni, in termine di probabilità di errore, che si vogliono ottenere.

Anche in questo caso è necessario un compromesso tra la probabilità di errore e la velocità di esecuzione.

Il tempo di decodifica, infatti, aumenta linearmente con il numero di iterazioni eseguite, com'è facile aspettarsi (Figura 5.7).

[15] Il sistema non implementa le modulazioni del DVB-S2, tuttavia le curve così ottenute corrispondono a quelle che si ottengono con modulazione BPSK in quanto ad ogni simbolo corrispondono due bit indipendenti tra loro dal punto di vista del rumore.

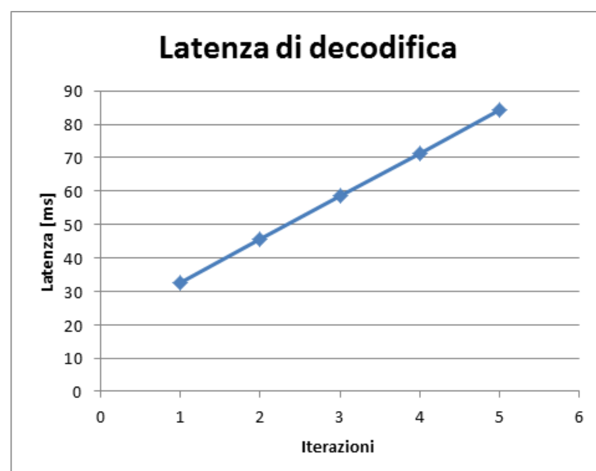


Figura 5.7: Dipendenza del tempo di decodifica dal numero di iterazioni per il DVB-S2 rate=1/2 normal FECFRAME

Dalla figura si nota che la retta ha un offset verso l'alto: questo è dovuto all'overhead, indipendente dal numero di iterazioni, rappresentato dall'inizializzazione e dall'hard decision finale.

Per caratterizzare questo overhead è stato introdotto il termine $3/2$ nelle formule (5.1), (5.2) e (5.3)[16].

5.6 Prestazioni finali e limitazioni

Seguendo le indicazioni del paragrafo precedente, sono state misurate le prestazioni per la decodifica di tutti i codici LDPC dello standard DVB-S2. Queste sono riportate nella Figura 5.8.

Purtroppo con l'hardware a disposizione le prestazioni ottenute non garantiscono la corretta decodifica di un flusso video digitale per ogni rate del codice. Infatti un flusso video codificato con lo standard MPEG-2 ha una bitrate compresa tra i 4 e i 9 Mbps.

La grande limitazione del sistema così costruito è la memoria.

Durante il profiling dei vari kernel, una delle statistiche osservate che più portano a questa conclusione è quella riguardo le ragioni di stallo delle istruzioni che, nella maggior parte dei casi, è per dipendenza di dato.

[16] Analizzando i valori ottenuti il termine corretto vale circa 1.55, è stato scelto 3/2 per semplificare la scrittura della formula che comunque è approssimativa.

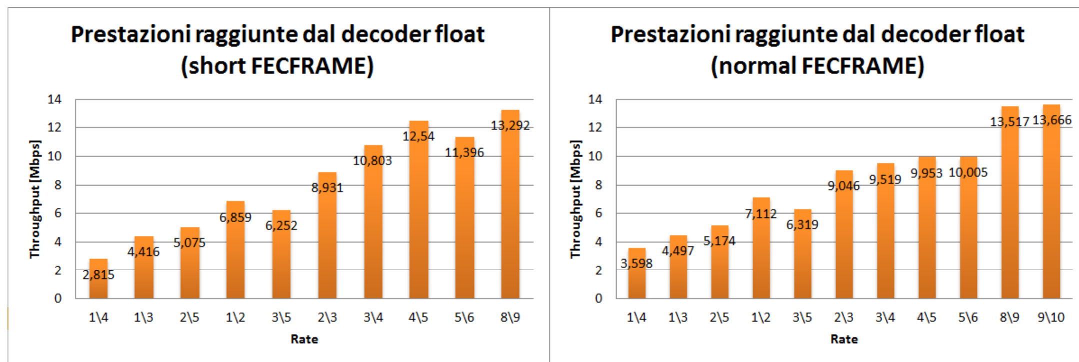


Figura 5.8: Prestazioni del decoder float per i codici del DVB-S2 e 10 iterazioni

Ciò significa che quando gli SM interrompono l'esecuzione perché non vi sono istruzioni "eseguibili" è perché stanno attendendo uno degli operandi.

Nel capitolo 3 si accennava al fatto che è possibile configurare la ripartizione dei 64KB di memoria a bordo di ogni SM tra Shared Memory e cache di primo livello; è stato eseguito un profiling del kernel LDPCker_dec_float_BNupdate in due casi: 48KB di Shared Memory e 16KB di cache, e viceversa. I risultati sono riportati nell'allegato 7.4.

Nell'allegato si può notare come l'aumento della quantità di memoria cache provochi una diminuzione del numero di istruzioni che stallano per dipendenza di esecuzione che, in proporzione, è pari alla diminuzione del tempo di esecuzione.

A conferma di quanto detto, si può notare anche un aumento significativo della Hit Rate della cache.

5.6.1 Confronto con lavori precedenti

Il principale lavoro già esistente è quello sviluppato da Gabriel Falcao nella sua tesi di dottorato [8], alla quale sono seguite altre pubblicazioni come per esempio [2].

Egli, oltre a sviluppare soluzioni per la decodifica di codici LDPC anche su architetture diverse, come FPGA o tipi diversi di schede video, ha realizzato un decoder a 8 bit in ambiente CUDA che è possibile reperire online [9].

Il funzionamento dei due decoder è del tutto simile: due kernel eseguono il processing orizzontale e verticale un numero prestabilito di volte, quindi viene misurato il tempo di esecuzione e il throughput.

Confrontando l'implementazione nei due casi si nota come i kernel del decoder di Falcao siano costituiti quasi completamente da istruzioni di calcolo e nessuna istruzione di salto.[17]

Il codice sorgente dei kernel è costituito da un gran numero di linee di codice, questo perché le caratteristiche della matrice di parità del codice LDPC usato sono inserite nel codice stesso.

Fissando la matrice di parità, si può sapere in anticipo quante iterazioni dovrà eseguire ogni ciclo for o a quali elementi accederà un kernel, quindi è preferibile scrivere queste istruzioni in modo sequenziale piuttosto che raggrupparle in un ciclo for che sarà eseguito inefficientemente a causa delle branch.

In [4] vengono mostrati i throughput ottenuti dagli autori con la loro implementazione: per i codici LDPC dello standard DVB-S2 a 10 iterazioni di decodifica variano da 36 a 87 Mbps. La scheda video utilizzata è una Fermi C2050.

È difficile confrontare le prestazioni ottenute nell'articolo sopracitato con quelle ottenuta in questo lavoro in quanto le due schede video hanno caratteristiche molto diverse, per esempio la banda della memoria disponibile è circa 10 volte superiore a quella della GeForce GT620.

Le prestazioni del decoder dipendono anche da molti altri fattori, come la quantità di cache, il numero di CUDA core o la massima velocità di esecuzione delle istruzioni (FLOPS[18]), perciò si è preferito compilare ed eseguire i sorgenti trovati in rete sulla macchina a disposizione e confrontarne i risultati con il decoder in esame.

Il decoder di Falcao raggiunge un throughput di 12.458 Mbps contro i 7.995 Mbps ottenuti da quello in esame utilizzando in entrambi i casi 10 iterazioni e una matrice di parità 8000x4000 con 24000 '1'.

Si può calcolare la latenza specifica di decodifica tenendo conto del fatto che il decoder di Falcao non esegue inizializzazione e Hard Decision che si traduce nella scomparsa dalle formule del termine $^{3}/_{2}$.

Il decoder di Falcao ha una latenza specifica di 1.338 $\left[\frac{ns}{Edge \cdot Iteration}\right]$ contro gli 1.813 $\left[\frac{ns}{Edge \cdot Iteration}\right]$ del decoder in esame che ha, quindi, delle performance inferiori del 36% circa.

---

[17] In realtà alcune delle macro usate eseguono dei confronti e quindi delle branch ma in numero decisamente inferiore rispetto alla soluzione presentata in questa tesi
[18] Floating Point Operations per Second

Questo è dovuto principalmente al numero ridotto di branch che una scheda video non è in grado di eseguire in modo efficiente.

Un software di questo tipo può, inoltre, evitare di eseguire gran parte dei calcoli legati all'indirizzamento e al recupero dei messaggi nei rispettivi vettori e, in generale, ha un pattern di accesso alla memoria molto regolare.

Il profiling dei kernel del decoder di Falcao ha rivelato inoltre una Hit rate della L1 cache di oltre il 90%.

Grazie a un accesso alla memoria così regolare si può ottenere una Hit Rate alta, che a sua volta aumenta le prestazioni del software limitato dalla memoria.

Il decoder qui presentato, pur avendo prestazioni inferiori, anche se paragonabili a quelle ottenute da Falcao, ha il vantaggio di essere flessibile.

La flessibilità è di sicuro un grande punto di forza: oltre a rendere le funzionalità implementate facilmente accessibili da altri sviluppatori e applicazioni, permette anche di ottenere un file eseguibile di dimensioni inferiori; pensando alla realizzazione di un terminale embedded, dove l'host avrà specifiche decisamente inferiori rispetto a quelle di un PC, utilizzare meno spazio in memoria per il codice significa poter ospitare sulla stessa macchina più algoritmi e più blocchi di elaborazione.

La filosofia della Software Defined Radio è proprio quella di poter realizzare una gran varietà di applicazioni in modo rapido e flessibile.

CAPITOLO 6

Conclusioni

6.1 Considerazioni sui risultati ottenuti

I risultati raccolti nel capitolo precedente, mostrano come il software venga eseguito in maniera efficiente.

La grande differenza con i risultati proposti da Falcao in [4] è dovuta allo stile di scrittura del codice che, a causa degli obiettivi diversi, rende questo progetto meno performante ma molto più flessibile.

Per realizzare un terminale di ricezione DVB-S2, oltre alla decodifica, sono necessarie molte altre elaborazioni che la scheda video utilizzata non è in grado di sostenere alle bitrate richieste per l'elaborazione di un flusso video.

L'elaborazione di codici LDPC con matrici così grandi implica l'utilizzo di un gran numero di thread, questo rende il sistema realizzato facilmente scalabile rispetto al numero di core di cui è dotata una scheda video o un sistema many-core.

Per migliorare le prestazioni in modo significativo si può sfruttare la scalabilità del software e, semplicemente, usare un hardware più evoluto: in particolare dotato di più memoria cache e di una banda più alta nel trasferimento di dati dai core alla memoria globale.

6.2 Lavori futuri

Un sistema che ricava le informazioni sul codice LDPC da un file e, a runtime, modifica il proprio comportamento secondo queste informazioni è sicuramente utile in ambiti in cui non si conosce a priori quale sia il comportamento che dovrà assumere, per esempio nell'ambito della Software Defined Radio quando, durante la progettazione, si vogliono esplorare diverse soluzioni.

Tuttavia per un terminale DVB-S2 lo standard prevede solo una fascia ristretta di codici LDPC; in questo caso può essere valido lo “stile” di programmazione adottato da Falcao, ovvero una serie di kernel per la codifica e decodifica che non contengono cicli e hanno “cablate” al loro interno tutte le informazioni sulla matrice di parità del codice LDPC.

La costruzione di un terminale che implementa uno standard è, infatti, un’applicazione intermedia tra un sistema statico e uno che può cambiare il suo comportamento con il massimo grado di libertà.

Queste funzioni potrebbero essere poi richiamate selettivamente, a runtime, in base alle esigenze dettate dalle condizioni di comunicazione del momento.

Questo lavoro potrebbe proseguire in futuro con la realizzazione di un framework di sviluppo che, dato un codice LDPC, crei il codice sorgente dei kernel per la codifica e decodifica e li integri automaticamente nel sistema compilando nuovamente la libreria. In questo modo, pur mantenendo le stesse API, la libreria usufruirà di kernel più veloci.

Nel corso dei capitoli precedenti si è studiata la possibilità di utilizzare diverse versioni di decoder; quelle a 8 bit e 16 bit sono le più performanti, in particolare quella a 16 bit offre una maggiore precisione, che si traduce in basse probabilità di errore, a throughput comunque eccellenti.

Tuttavia resta aperto il problema della saturazione delle variabili che impediscono una decodifica corretta del segnale ricevuto.

Vi è la necessità di analizzare più in profondità questo problema, inserendo un sistema di aggiustamento del guadagno o modificando l’algoritmo MSA per evitare che le variabili aumentino di magnitudine.

Può essere interessante, anche, studiare la possibilità di realizzare un decoder con variabili a doppia precisione (double) per valutare se l’inevitabile perdita di performance dovuta all’utilizzo di una quantità doppia di memoria rispetto ai float, e dovuta al tempo di esecuzione maggiore di istruzioni con questo tipo di operandi, possa essere giustificata da una maggior precisione e, quindi, dalla possibilità di eseguire meno iterazioni per ottenere la stessa BER.

Infine, nel corso di questa tesi si è dimostrato come l’utilizzo della memoria Texture abbia un effetto negativo sulle performance; tuttavia non è mai stata utilizzata la memoria condivisa in quanto non vi era necessità che i thread nello stesso TB cooperassero in alcun modo.

Tuttavia, anche configurando la cache in modo che ve ne sia il più possibile (48KB), parte della memoria a bordo degli SM è comunque occupata da quella condivisa (16KB).

È evidente come la quantità di cache influisca direttamente sulla velocità di esecuzione dell'algoritmo, quindi potrebbe essere interessante trovare un modo efficiente di usare la memoria condivisa, in modo da trarne un beneficio come se questa, in realtà, fosse una cache aggiuntiva.

Bibliografia

[1] W. Plishker, G. F. Zaki, S. S. Bhattacharyya, C. Clancy e J. Kuykendall, «Applying Graphics Processor Acceleration in a Software Defined Radio Prototyping Environment,» 2011.

[2] G. Falcao, *Parallel Algorithms and Architectures for LDPC Decoding,* Universidade de Coimbra, 2010.

[3] G. Falcao, V. Silva e L. Sousa, «How GPUs Can Outperform ASICs for Fast LDPC Decoding».

[4] G. Falcao, J. Andrade, V. Silva e L. Sousa, «Real-time DVB-S2 LDPC decoding on many-core GPU accelerators».

[5] Amin Shokrollahi - Digital Fountain, Inc., *LDPC Codes: An Introduction,* 2003.

[6] H. Jin, A. Khandekar e R. McEliece, «Irregular repeat-accumulate codes,» in *Int. Symp. Turbo Codes and Related Topics*, Brest, France, settembre 2000.

[7] ETSI, «Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications,» [Online]. Available: http://www.etsi.org/deliver/etsi_en/302300_302399/302307/01.01.01_60/en_302307v010101p.pdf. [Consultato il giorno Febbraio 2014].

[8] Nvidia, «Sito Web dell'Nvidia,» [Online]. Available: http://www.nvidia.com/.

[9] P. N. Glaskowsky, «NVIDIA's Fermi: The First Complete GPU Computing Architecture,» 2009.

[10] Nvidia, CUDA C Programming Guide, 2012.

[11] Nvidia, CUDA C Best Practices Guide, 2012.

[12] G. F. J Andrade, «Repository decoder 8bit OpenCL e CUDA,» 16 dicembre 2011. [Online]. Available: http://montecristo.co.it.pt/opencl_ldpc/OpenCL_LDPC_uint8/html/index.html.

[13] R. G. Gallager, *Low-Density Parity-Check Codes,* 1963.

[14] T. Ta, «A tutorial on low density parity-check codes,» *University of Texas at Austin.*

Indice delle Figure

Figura 2.1: Schema a blocchi di riferimento 3
Figura 2.2: Codice di esempio: parametri caratteristici e relativo grafo di Tanner 7
Figura 2.3: Diagramma a blocchi di un encoder RA 8
Figura 2.4: Grafo di Tanner di un codice IRA e schema a blocchi equivalente 9
Figura 2.5: funzione φ(x) 18
Figura 2.6: Associazione tra i thread e i messaggi che questo deve calcolare 21
Figura 2.7: Schema a blocchi di un sistema di trasmissione DVB-S2 [3] 24
Figura 2.8: Esempio di matrice H di un codice LDPC del DVB-S2 25
Figura 3.1: Confronto sull'utilizzo dell'area del chip tra una CPU e una GPU ALU e CUDA cores sono le porzioni di chip riservate ai calcoli aritmetici) 30
Figura 3.2: Architettura di un CUDA Streaming Multiprocessor 30
Figura 3.3: Esempio di come le Dispatch Unit assegnano ai vari EB le istruzioni. 32
Figura 3.4: Organizzazione dei thread 33
Figura 3.5: Le memorie della GPU e la loro visibilità 34
Figura 3.6. Principio di Località Spaziale in 2 dimensioni 35
Figura 3.7: Effetto della cache e del disallineamento sul numero di trasferimenti (i riquadri verdi). 39
Figura 3.8: Vantaggio dell'esecuzione asincrona e sovrapposta ai trasferimenti 41
Figura 4.1: Schema a blocchi del sistema 44
Figura 4.2: Spiegazione grafica del significato dei vettori Hcn e Hbn 48
Figura 4.3: Schema a blocchi dell'encoder 59
Figura 4.4: Schema a blocchi del simulatore di canale 60
Figura 4.5: Schema a blocchi del decoder (float) 61
Figura 4.6: Schema a blocchi del calcolatore di BER 63
Figura 4.7: Principio di funzionamento delle operazioni di riduzione 63
Figura 5.1: Visualizzazione grafica dell'esecuzione di una simulazione di trasmissione e ricezione nel tempo 66
Figura 5.2: Pattern di accesso in memoria variando il numero di parole decodificate in parallelo, nel caso c si ha coalescenza 69
Figura 5.3: Dipendenza delle prestazioni dal numero di parole elaborate in parallelo . 70
Figura 5.4: Grafici tratti dall'Occupancy Calculator che evidenziano le impostazioni ottimali per i kernel del progetto 73
Figura 5.5: Dipendenza delle prestazioni dall'Occupancy e dalla configurazione dei thread 73

Figura 5.6: Variazione delle curve BER rispetto al numero di iterazioni (DVB-S2 normal FECFRAME Rate 1/2) 75
Figura 5.7: Dipendenza del tempo di decodifica dal numero di iterazioni per il DVB-S2 rate=1/2 normal FECFRAME.. 76
Figura 5.8: Prestazioni del decoder float per i codici del DVB-S2 e 10 iterazioni....... 77

Indice delle Tabelle

Tabella 2.1: Numero di operazioni aritmetiche e di accessi in memoria per iterazione per MSA ottimizzato e codice regolare [2].. 21

Indice dei Codici

Codice 3.1: Come lanciare un kernel impostando il numero dei thread....................... 33
Codice 3.2: Esempio di stratagemma per evitare l'uso di branch................................ 40
Codice 4.1: Definizione della struttura CodeSpecs 46
Codice 4.2: esempio di iterazione tra i messaggi dell'insieme N(m) in pseudo-codice49
Codice 4.3: esempio di iterazione tra i messaggi dell'insieme N(m) in pseudo-codice con vettori regolarizzati 49
Codice 4.4: Algoritmo per calcolare il contenuto di Hcn e Hbn 50
Codice 4.5: definizione della struttura systemCtrl... 53
Codice 4.6: Esempio di inizializzazione di sistema per la simulazione 55
Codice 4.7:Esempio di utilizzo delle API per trovare la Bit Error Rate a un certo rapporto segnale rumore 55
Codice 4.8: Programma utilizzato per misurare il throughput dei vari blocchi............ 57
Codice 4.9: Esempio di decoder real-time embedded 58

Allegati

7.1 Soluzione Visual Studio 2010

Il codice sviluppato in questa tesi è raccolto in un unica soluzione di Visual Studio 2010.

Questa è suddivisa in diversi progetti: uno per la creazione della libreria LDPC.lib mentre gli altri consistono in dimostrazioni dell'utilizzo di questa libreria, dalla simulazione della catena con la relativa misurazione delle prestazioni, fino a un decoder real time.

Tramite il comando "Build" si possono creare i file eseguibili automaticamente; in seguito è illustrata la procedura per creare un nuovo progetto.

7.1.1 Progetti realizzati

BER_curve è un programma che esegue la simulazione di una trasmissione e ricezione di dati utilizzando tutta la catena a diversi valori di rapporto segnale rumore.

Restituisce, in un file di testo, un vettore contenente i valori di BER che è possibile copiare e incollare direttamente in uno script Matlab.

Si può impostare, inoltre, il numero minimo di bit sbagliati perché la simulazione sia valida e il numero massimo di simulazioni eseguibili.

ChainSimulation esegue una simulazione della catena a un certo valore di rapporto segnale rumore misurando, al tempo stesso, latenza e throughput dei singoli blocchi.

Per una misurazione esente il più possibile da overhead, ogni funzione viene lanciata più volte.

GPUspecsAndTest fornisce una serie di specifiche tecniche riguardo la scheda video a bordo del sistema, usando le API di CUDA, ed esegue una serie di misure sulla banda effettiva di trasferimento dalla memoria.

LDPC è la libreria oggetto di questa tesi che viene compilata in un file .lib.

RealTimeDecoder è un decoder real time che esegue un certo numero di volte la decodifica di dati presenti su un vettore in memoria principale inserendo il risultato in un secondo vettore, nel frattempo misura la bitrate corrente.

Il programma eseguirà un numero limitato di iterazioni, ma è possibile de-commentare la costante "DO_FOREVER" facendo sì che il programma continui il suo funzionamento indefinitamente.

È possibile specificare la bitrate desiderata e, se questa è minore o uguale alla bitrate massima possibile, ad ogni iterazione il programma attenderà un certo lasso di tempo in modo da mantenerla.

Infine si può specificare il rapporto segnale rumore.

7.1.2 Organizzazione dei File

La cartella principale della soluzione è divisa in più sottocartelle compatibilmente con il loro contenuto.

La cartella "bin", suddivisa in "Debug" e "Release" contiene gli eseguibili dei programmi sopracitati. Le versioni Release non contengono numerose istruzioni utilizzate dal debugger di Visual Studio per la gestione di eccezioni, risultando più veloci.

Ogni progetto ha una cartella corrispondente che contiene le sue configurazioni e i suoi sorgenti.

La cartella "CodeSpecs" contiene i file .cod di numerosi codici LDPC, tutti generati con lo script Matlab "CodFileGenerator.m"

"include" contiene gli header file della libreria LDPC da includere nel programmi principali.

"lib", a sua volta suddiviso in "Debug" e "Release", contiene il file .lib della libreria statica LDPC.

"src", infine, contiene i file sorgenti della libreria LDPC.

Nella cartella radice sono presenti i file della soluzione di Visual Studio 2010.

7.1.3 Come creare un nuovo progetto

Nel caso in cui si voglia creare un nuovo progetto all'interno di questa soluzione, rendendolo compatibile con la suddivisione delle cartelle sopracitata, è necessario aver installato sul sistema il CUDA toolkit, scaricabile gratuitamente dal sito dell'Nvidia, quindi è sufficiente seguire le istruzioni seguenti:

1. Creare il progetto aprendo la soluzione e cliccando col tasto destro del mouse su di essa nel solution explorer sulla sinistra; quindi cliccare su Add -> New Project

Selezionare come template CUDA, inserire il nome del progetto, e assicurarsi che la cartella di lavoro sia \LDPC_SDK_VS2010\ (cartella radice)

2. Accedere alle proprietà del progetto tramite Project -> Property e impostare come configurazione (il menù a tendina in alto) “All Configurations” Impostare in Configuration Properties -> General -> Output Directory “$(SolutionDir)\bin\$(Configuration)\
3. In VC++ Directories -> Library Directories, aggiungere la cartella “lib” della soluzione (quella contenente LDPC.lib) distinguendo i due casi Debug e Release con le relative sotto cartelle
4. Allo stesso modo aggiungere la cartella “include” in “Include Directories” insieme con la directory degli header file del CUDA toolkit: di default “C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.5\common\inc”
5. In Configuration Properties -> Linker -> Input assicurarsi che sia presente cudart.lib, quindi aggiungere LDPC.lib e curand.lib
6. Modificare CUDA C/C++ -> Device -> Code Generation in modo che risulti compute_20, sm_20

Di default viene creato un file kernel.cu contenente un programma di prova. Questo file può essere rinominato a piacimento anche con estensione .c o .cpp e conterrà il programma principale

7.2 Descrizione dei kernel

7.2.1 Kernel per la codifica

LDPCker_enc_CopyInfoBit

Estrae il k-esimo bit di informazione dal buffer ‘u’ e, convertendolo nel format “fat bit”, lo copia all’inizio del vettore ‘c’. Equivale alla codifica sistematica dove i primi *K* bit della parola di codice sono uguali ai bit originali di informazione.

```
__global__ void LDPCker_enc_CopyInfoBit(int K,int N,int KinByte,char *u,char *c){
        int cdw_numb = threadIdx.x;                                    //codeword number
        int k=threadIdx.y + blockIdx.x*THREADS_PER_BLOCK/CDW_IN_PARALLEL;  //k-th bit
        if(cdw_numb>=CDW_IN_PARALLEL|k>=K) return;   //kill kernel if out of bounds

        int byten = k/8;
        int bitn = k%8;
        c[cdw_numb*N+k]=(u[cdw_numb*KinByte+byten]>>bitn)&0x01;
        //=0x01 if bit under test is 1, 0x00 otherwise
}
```

LDPCker_enc_LimitedMatrixMul

Questo kernel esegue la moltiplicazione matriciale tra il vettore ‘u’ e la sottomatrice di ***H*** tale che ogni riga ha i primi *limit* ‘1’ delle righe di ***H***. Se *limit* vale *a* (notazione del paragrafo 2.2.1) che coincide con *wcmax-2*, questa sottomatrice coincide con G_1 e l’operazione equivale alla codifica di un codice IRA sistematico.

Ogni thread calcola un elemento del vettore risultante, eseguendo il prodotto scalare tra una riga della matrice ed u.

Le righe della matrice sono memorizzate come un vettore che ha per elementi gli indici ai quali si trova un ‘1’. Il bit di ‘u’ all’indice corrispondente viene analizzato e, se questo vale 1, la somma viene incrementata. L’istruzione che esegue l’incremento è scritta in modo da evitare una branch sfruttando il fatto che la quantità da incrementare è pari al valore del bit stesso.

Infine, eseguendo il modulo a 2 della somma, si trova il valore del bit risultato, che viene scritto sul vettore Gu.

```
__global__ void LDPCker_enc_LimitedMatrixMul(int limit,int KinByte,int M,int wcmax,char
*u,int *H,char *Gu){
        int TID = blockDim.x*blockIdx.x+threadIdx.x;        //1 Thread = 1 row*vec mult
        int cdw = TID%CDW_IN_PARALLEL;                      //codeword index
        int m = TID/CDW_IN_PARALLEL;                        //bit index
        if(m>=M) return;                        //kill Kernel if out of bounds

        int *row = &H[m*wcmax];       //the row this kernel have to multiply with u
        char *specificU = &u[KinByte*cdw];   //select right codeword
```

```
        int sum =0;
        for(int i=0;i<limit;i++){     //limit the number of '1'
                int but = row[i]-1;           //Bit Under Test
                int byten = but/8;
                int bitn = but%8;
                sum+=specificU[byten]>>bitn&0x01;
//equivalent to "if bit under test is 1, increment sum: specificU[byten]>>bitn&0x01=1
if bit under test=1, 0 otherwise
        }
        Gu[cdw*M+m] = sum&0x01;
        //copy in the destination vector and do modulus 2   (&0x01 ==  mod 2)
}
```

7.2.2 Kernel per la decodifica[19]

LDPCker_dec_<8|16>bit_BNupdate

Il kernel implementa la formula di aggiornamento dei nodi variabile (2.40)

Nel caso mostrato di seguito (8 bit) la sommatoria viene eseguita a 32 bit per poi ri-convertire il risultato a 8 bit. La macro _8bCLIP limita il risultato tra -128 e 127 in caso di overflow.

Il codice mostra il caso di decoder a 8 bit, nella versione a 16 bit cambia solo il tipo di dato usato per le variabili. Lo stesso discorso vale per i kernel seguenti.

```
__global__ void LDPCker_dec_8bit_BNupdate(int N,char *r,char *q,char *Pi,int *Hcn,int
*wb,int wbmax,char *hatc,int wcmax){
        int TID = blockDim.x*blockIdx.x+threadIdx.x;        //one thread each bit node
        int cdw = TID%CDW_IN_PARALLEL;       //codeword index
        int n = TID/CDW_IN_PARALLEL;         //bit index
        if(n>=N) return;             //kill kernel if out of bounds
        int from=n*wbmax;
        int to=from+wb[n];

        //compute Qn
        char sum = Pi[n*CDW_IN_PARALLEL+cdw];
        for(int i=from;i<to;i++){
                char temp = r[Hcn[i]*CDW_IN_PARALLEL+cdw];
                sum = _8bCLIP((int)sum + (int)temp); //do the 32 bit sum and clip again
        }
        //message calculation
        for(int i=from;i<to;i++){
                q[i*CDW_IN_PARALLEL+cdw] = sum - r[Hcn[i]*CDW_IN_PARALLEL+cdw];
        }
}
```

[19] Dal codice sono state escluse le parti relative alla Texture memory per semplicità

LDPCker_dec_<8|16>bit_CNupdate

Questo kernel implementa l’aggiornamento dei nodi di controllo seguendo la formula (2.39) e le considerazioni espresse alla fine del paragrafo 2.3.3.

In una prima iterazione lungo gli elementi dell’insieme N(m) vengono calcolati il valore minimo, il secondo valore più piccolo, e il prodotto dei segni.

Partendo da questi, nella seconda iterazione, si possono creare tutti i singoli messaggi che avranno segno pari a quello espresso dalla formula (2.41) e modulo pari al minimo, tranne nel caso del messaggio diretto al nodo variabile che ha generato questo al quale verrà inviato il secondo valore più piccolo.

Sono stati utilizzati alcuni stratagemmi per evitare operazioni onerose come le moltiplicazioni nel caso del calcolo del segno del prodotto, preferendo lo xor bit a bit il cui risultato avrà il medesimo bit di segno.

```
__global__ void LDPCker_dec_8bit_CNupdate(int M,char *q,int *Hbn,int *wc,char *r,int wcmax,int wbmax){
        int TID = blockDim.x*blockIdx.x+threadIdx.x; //one thread each check node
        int cdw = TID%CDW_IN_PARALLEL;               //codeword index
        int m = TID/CDW_IN_PARALLEL;                 //bit index
        if(m>=M) return;            //kill kernel if out of bounds

        int from=m*wcmax;
        int to=from+wc[m];

        //find: sign_product, min, 2ndmin
        char signProd = 0;
        char min = 127;
        char min2 = 127;
        int minindex=-1;
        for(int i=from;i<to;i++){
                char curr_q =q[Hbn[i]*CDW_IN_PARALLEL+cdw];
                char curr_qabs = _8bABS(curr_q);
                signProd ^= curr_q;
            //the sign bit is the sign of the multiplication (all the other bits don’t)
                if(curr_qabs<min){    //update min
                        min2 = min;
                        min=curr_qabs;
                        minindex=i;
                }else{
                        if(curr_qabs<min2) min2=curr_qabs;   //update second min
                }
        }

        //compute single messages
        for(int i=from;i<to;i++){
                char curr_q =q[Hbn[i]*CDW_IN_PARALLEL+cdw];
```

```
                char temp;
                if(i==minindex) temp = ((signProd^curr_q)&0x80)==0?min2:-min2;
                //equivalent to: sign(signProd*curr_q)*min2
                else temp = ((signProd^curr_q)&0x80)==0?min:-min;
                //equivalent to: sign(signProd*curr_q)*min
                r[i*CDW_IN_PARALLEL+cdw] = temp;
        }
}
```

LDPCker_dec_<8|16>bit_HardDecision

La hard decision viene eseguita calcolando prima gli $LQ_n^{(l)}$, quindi ponendo il bit corrispondente a 1 se $LQ_n^{(l)}$ è negativo, 0 viceversa.

Questa operazione è eseguita come un semplice “and” (sum=sum&0x80) sfruttando il fatto che il MSB di “sum” è anche il bit di segno e quindi assume esattamente il valore dell’output negli stessi casi descritti sopra.

Si può notare, confrontando l’ultima riga con quella in cui si accede ad ‘r’, che il kernel riordina anche i dati in modo conforme allo standard dei buffer principali del sistema.

Il codice mostra il caso di decoder a 8 bit, nella versione a 16 bit cambia solo il tipo di dato usato per le variabili.

```
__global__ void LDPCker_dec_8bit_HardDecision(int N,char *hatc,char *Pi,int *Hcn,int
*wb,int wbmax,char *r,int wcmax){
        int TID = blockDim.x*blockIdx.x+threadIdx.x;//one thread each output bit
        int cdw = TID%CDW_IN_PARALLEL;               //codeword index
        int n = TID/CDW_IN_PARALLEL;                 //bit index
        if(n>=N) return;             //kill kernel if out of bounds

        int from=n*wbmax;      //range of Hcn elements containing BN indexes in M(n)
        int to=from+wb[n];

        //compute Qn
        char sum = Pi[n*CDW_IN_PARALLEL+cdw];
        for(int i=from;i<to;i++){
                char temp = r[Hcn[i]*CDW_IN_PARALLEL+cdw];  //message in M(n)
                sum = _8bCLIP((int)sum + (int)temp); //do 32bit sum and back to 8bit
        }
        sum=sum&0x80;                //fat bit with info in MSB
        hatc[cdw*N+n] = sum;
}
```

LDPCker_dec_<8|16>bit_Init

Questo kernel si occupa di inizializzare i nodi bit calcolando il rapporto di verosimiglianza a priori di ogni bit ricevuto moltiplicandolo per un coefficiente di guadagno.

In seguito inizializza i messaggi $Lq_n^{(l)}$ con questo valore e riorganizza l'ordine degli elementi di questi buffer per sfruttare la coalescenza (si può notare come l'indice all'interno di 'Pi' e 'q' sia calcolato diversamente dall'indice all'interno di 'y')

```
__global__ void LDPCker_dec_8bit_Init(float K,int N,float *y,char *Pi,char *q,int
wbmax){
	int TID = blockDim.x*blockIdx.x+threadIdx.x;//one thread each bit node
	int cdw = TID%CDW_IN_PARALLEL;			//codeword index
	int n = TID/CDW_IN_PARALLEL;			//bit index
	if(n>=N) return;			//kill kernel if out of bounds
	char val = _8bCLIP(y[N*cdw+n]*K);	//scale by K and clip from -128 to 127

	Pi[TID]=val;
	int from=n*wbmax;
	int to=(n+1)*wbmax;
	for(int i=from;i<to;i++){//each message sent in the 1st round is equal to Pi
		q[i*CDW_IN_PARALLEL+cdw]=val;
	}
}
```

LDPCker_dec_float_BNupdate

Il principio di funzionamento è il medesimo delle versioni a 8 e 16 bit senza l'operazione di clipping delle variabili.

LDPCker_dec_float_CNupdate

Analogamente alle versioni a 8 e 16 bit questo kernel calcola i messaggi da inviare ai nodi variabile.

LDPCker_dec_float_HardDecision

Il principio di funzionamento è il medesimo della versione a 8 e 16 bit.

L'unica differenza consiste nel calcolo finale del bit della parola di codice.

Per le variabili a 8 e 16 bit il bit di segno coincide con il MSB perciò, per evitare la branch, si copia questo nel risultato. Nel caso di variabili float l'unica differenza consiste nell'utilizzo di un puntatore e una conversione di tipo di dato per poter accedere a quel bit.

```
float sum;
...
int *intptr = (int *)&sum;	//float -> binary representation -> sign bit mask
int intvar = *intptr;
hatcBuf[cdw*N+n] = (intvar>>24)&0x80;
```

LDPCker_dec_float_Init

Il funzionamento è analogo alle versione a 8 e 16 bit, il coefficiente di guadagno viene però calcolato basandosi sulla formula (2.32).

LDPCker_dec_SystDec
Ipotizzando un codice sistematico, questo kernel estrae i primi K bit dalla parola di codice e li copia nel vettore 'hat u' eseguendo, così, la decodifica.

Da notare che i bit nel vettore 'hat c' sono in formato "fat bit" ma, al contrario di altri buffer, l'informazione è contenuta nel MSB.

Il kernel produce in uscita un intero byte, perciò esegue 8 iterazioni riempiendo di volta in volta la variabile output.

```
__global__ void LDPCker_dec_SystDec(char *hatc,char *hatu,int Kinbyte,int N){
        int TID = blockDim.x*blockIdx.x+threadIdx.x;        //1 thread each output byte
        int cdw = TID/Kinbyte;                              //codeword index
        if(cdw>=CDW_IN_PARALLEL) return;            //kill kernel if out of bounds
        int byteIdx = TID%Kinbyte;
        char output = 0x00;
        for(int i=0;i<8;i++){
                output=0x7F&(output>>1);                    //right shift with 0 input
                char hatcval = hatc[cdw*N+(byteIdx*8+i)];
                output |= hatcval&0x80;                     //hatc has 0 or 1 in the MSB
        }
        hatu[TID] = output;
}
```

7.2.3 Kernel per altre elaborazioni

LDPCker_ber_bitwiseXor
Esegue lo xor bit a bit di due vettori usando un thread per ogni byte.

```
__global__ void LDPCker_ber_bitwiseXor(char *a,char *b,char *c,int len){
        int n = blockIdx.x*blockDim.x+threadIdx.x;
        if(n>=len) return;
        c[n] = a[n]^b[n];
}
```

LDPCker_ber_treeSum
Questo kernel somma due elementi (distanti tra loro $\left\lfloor\frac{N}{2}\right\rfloor$) di un vettore ponendo il risultato al posto del primo.

Richiamato iterativamente riducendo di volta in volta N si ottiene un'efficiente implementazione del calcolo della somma degli elementi di un vettore (Figura 4.7)

```
__global__ void LDPCker_ber_treeSum(int *tempSum,int N){
        int TID = blockIdx.x*blockDim.x+threadIdx.x;
        if(TID>=(N>>1)) return;
        tempSum[TID] += tempSum[TID+(N>>1)];
}
```

LDPCker_ber_wrongBitCountFirstStep

Questo kernel conta il numero di '1' in un certo byte di un vettore, scrivendo il risultato nell'elemento corrispondente di un secondo vettore.

Nel caso in cui K non sia multiplo di 8, vengono considerati i primi 'limit' bit del byte, valore che va calcolato precedentemente come il modulo tra K e 8 (il calcolo è eseguito all'esterno per evitare una ripetizione e un rallentamento inutile dell'algoritmo).

```
__global__ void LDPCker_ber_wrongBitCountFirstStep(char *xorRes,int KinByte,int lim-
it,int *sum){
	int TID = blockIdx.x*blockDim.x+threadIdx.x;		//1 Thread each byte
	int cdw = TID/KinByte;
	if(cdw>=CDW_IN_PARALLEL) return;		//kill if out of bounds
	int byteIdx=TID%KinByte;
	int forCount = 8;
	if(byteIdx==KinByte-1) forCount=limit;//last byte (if K is not multiple of 8)
	sum[TID]=0;
	for(int i=0;i<forCount;i++,xorRes[TID]>>=1){//count the number of ones
		sum[TID]+=xorRes[TID]&0x01;		//mask and right shift
	}
}
```

LDPCker_awgn_NoiseAdd

Il kernel esegue la modulazione BPSK dei bit da inviare nel canale, quindi vi applica il rumore bianco calcolato precedentemente e situato nel vettore n.

```
__global__ void LDPCker_awgn_NoiseAdd(int N,float *y,char *c,float *n){
	int k = blockIdx.x*blockDim.x+threadIdx.x;
	if(k>=N*CDW_IN_PARALLEL) return;	//kill if out of bounds
	//mapping (BPSK) & noise summation
	y[k] = 1.0-2.0*((float)c[k]) + n[k];		//0->1.0,  1->-1.0
}
```

7.3 Confronto tra le prestazioni del decoder variando il numero di parole decodificate in parallelo

Parametro		BN update				
		CDW = 32		CDW = 4		
		valore	% max	valore	% max	differenza
General	Bit sent	1036800	-	129600	-	-88%
	Execution Time [ms]	6,06	-	4,71	-	-22%
Occupancy	Active Warps	41,48	86,42%	43,6	90,83%	5%
Instruction Statistics	IPC (Executed)	1,48	37,00%	0,24	6,00%	-84%
	Serialization	16,81%	16,81%	56,42%	56,42%	236%
	SM Activity	100%	100,00%	100%	100,00%	0%
	Instruction Per Warp	112,2	-	112,2	-	0%
	Warp Launched	64800	-	8100	-	-88%
Global Memory (Logic)	Loads	1036796	-	129604	-	-87%
	Stores	226799	-	28351	-	-87%
	Loads (Trans. Per request)	1	3,13%	4,37	13,66%	337%
	Stores (Trans. Per Request)	1	3,13%	8	25,00%	700%
L1 Cache	Replay Overhead	0,00%	0,00%	30,50%	30,50%	-
	Hit Rate	66,02%	66,02%	28,18%	28,18%	-57%
	Load Transaction	1036796	-	567002	-	-45%
	Load Size [MB]	126,56	-	69,21	-	-45%
	Load BW [GB/s]	20,41	-	14,33	-	-30%
	Store Transactions	226799	-	226799	-	0%
	Store Size [MB]	27,69	-	27,69	-	0%
	Store BW [GB/s]	4,47	-	5,73	-	28%
L2 Cache	Hit Rate	4,62%	4,62%	1,87%	1,87%	-60%
	Load Transactions	1409632	-	1629852	-	16%
	Load Size [MB]	43,02	-	49,74	-	16%
	Load BW [GB/s]	6,94	-	10,3	-	48%
	Store Transactions	907196	-	226799	-	-75%
	Store Size [MB]	27,69	-	6,92	-	-75%
	Store BW [GB/s]	4,47	-	1,43	-	-68%
Device Memory (Physical)	Load Size [MB]	39,82	-	48,7	-	22%
	Load BW [GB/s]	6,42	46,90%	10,08	73,63%	57%
	Store Size [MB]	27,69	-	6,92	-	-75%
	Store BW [GB/s]	4,47	32,65%	1,43	10,45%	-68%
	Total BW [GB/s]	10,89	79,55%	11,51	84,08%	6%

7.4 Confronto tra le prestazioni del decoder variando la quantità di L1 cache

Parametro		BN update				
		Cache = 16KB		Cache = 48KB		
		valore	% max	valore	% max	differenza
	Execution Time [ms]	6,68	-	6,06	-	**-9%**
Instruction Statistics	IPC (Executed)	1,34	33,50%	1,48	37,00%	10%
	IPC (Issued)	1,66	41,50%	1,78	44,50%	7%
	Serialization	19,23%	19,23%	16,81%	16,81%	-13%
	SM Activity	100%	100,00%	100%	100,00%	0%
	Stall Reason (Execution Dependency)	80,12%	-	73,96%	-	**-8%**
	Stall Reason (Other)	19,88%	-	26,04%	-	31%
Global Memory (Logic)	Loads	1036796	-	1036796	-	0%
	Stores	226799	-	226799	-	0%
	Loads (Trans. Per request)	1	3,13%	1	3,13%	0%
	Stores (Trans. Per Request)	1	3,13%	1	3,13%	0%
L1 Cache	Replay Overhead	0	0,00%	0,00%	0,00%	-
	Hit Rate	53,71%	53,71%	66,02%	66,02%	23%
	Load Transaction	1036796	-	1036796	-	0%
	Load Size [MB]	126,56	-	126,56	-	0%
	Load BW [GB/s]	18,5	-	20,41	-	10%
	Store Transactions	226799	-	226799	-	0%
	Store Size [MB]	27,69	-	27,69	-	0%
	Store BW [GB/s]	4,05	-	4,47	-	10%
L2 Cache	Hit Rate	16,80%	16,80%	4,62%	4,62%	-73%
	Load Transactions	1918664	-	1409632	-	-27%
	Load Size [MB]	58,55	-	43,02	-	-27%
	Load BW [GB/s]	8,56	-	6,94	-	-19%
	Store Transactions	907196	-	907196	-	0%
	Store Size [MB]	27,69	-	27,69	-	0%
	Store BW [GB/s]	4,05	-	4,47	-	10%
Device Memory (Physical)	Load Size [MB]	44,08	-	39,82	-	-10%
	Load BW [GB/s]	6,44	47,04%	6,42	46,90%	0%
	Store Size [MB]	27,69	-	27,69	-	0%
	Store BW [GB/s]	4,05	30%	4,47	32,65%	10%
	Total BW [GB/s]	10,49	77%	10,89	79,55%	4%