UNIVERSITÀ DEGLI STUDI DI GENOVA
FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Elettronica

Tesi di laurea

Algoritmi per training di reti neurali SVM su piattaforma DSP

Relatore: Chiar.mo Prof. Rodolfo Zunino
Correlatore: Ing. Giovanni Parodi

Candidato: Sergio Decherchi

Genova, 26 luglio 2005

Alla commissione di Laurea e di Diploma
Alla Commissione Tirocini e Tesi

Sottopongo la tesi redatta dallo studente Sergio Decherchi dal titolo:

'*Algoritmi per training di reti neurali SVM su piattaforma DSP*'

Ho esaminato, nella forma e nel contenuto, la versione finale di questo elaborato scritto, e propongo che la tesi sia valutata positivamente, assegnando i corrispondenti crediti formativi.

Il relatore Accademico
Prof. Ing. Rodolfo Zunino

Vorrei dedicare questo lavoro
a tutti coloro che fin qui con amore,
mi hanno sostenuto ,
ed in particolar modo a mio fratello Carlo,
i miei familiari i miei amici.

Mantieni le cose semplici :
il più possibile , ma non più di così.
A.Einstein

Indice

Elenco delle figure

Elenco delle tabelle

Listings

Abstract

Nowdays one of the most attractive discipline is *Machine Learning* and in general Artificial Intelligence: the possibility of having good classifiers, regressors (et cetera) getting codes, relatively, simple and usable, allows an always getting bigger diffusion of the IA discipline itself. Today with a simple PC, with its well known computational skills, one can think to solve problems of big dimensions ($10^5, 10^6$ pattern) in reasonable times . In this work, in particular, will be studied the chance of porting a training algortihm over DSP platform. The application domain of a AI system ported over an embedded platform is large, and knows applications in those market sectors (Consumer the first one) where powerfull,simple, low-cost systems are needed. For reaching the objective to open this new prospective the SVM model will be used, where SVM is acronim for Support Vector Machine. SVM technology is based on a strong and consolidated mathematical theory. SVM, like most of AI systems, is based on two phases: in the first one, the machine learns the classification problem that it's desired to solve (training phase); in the second phase (test phase) the machine is used on the field, to test its acquired classification skills. The hardware system used will be Blackfin processor by Analog Devices: it is an entry level peripheral with good computational ability,low-[cost,power] . In this work will be implemented the training phase over this DSP: this task essentially consists in the minimization of a quadratic form. Different algoritms exist to solve this kind of problem: gradient descent, coniugate gradients, Baba, SMO and others. In this case SMO will be used: this algortihm, for its own nature, analitically solves the smallest solvable problem, therefore it results to be strongly well posed in situations (like this one) where poor resources are available. DSP context, and in general embedded systems context, is depicted by limited computational capabilities, low power

dissipation and poor memory subsystems. Like it will be discussed, under proper hipothesis due to memory capacity and numerical rappresentation, the developed systems will be able to perform a correct training; furthermost a promising SMO algorithm modification will be proposed. The exposition will start from a theorethical background for reaching step by step implementations,results and future developments.

Capitolo 1

Prefazione

Oggi una delle discipline di maggiore interesse e attrazione è il *Machine Learning* ed in generale l'Intelligenza Artificiale: la possibilità di avere buoni classificatori,regressori (et cetera) realizzando codici, relativamente, semplici e manutenibili sta consentendo sempre più una maggiore diffusione della disciplina IA. Oggi con un semplice PC, viste le attuali capacità computazionali, si può pensare di risolvere problemi anche di grosse dimensioni ($10^5, 10^6$ pattern) in tempi accettabili. In questo lavoro in particolare verrà esplorata la possibilità di portare algoritmi di training su piattaforma DSP. Il dominio applicativo di un'intelligenza artificiale portata su piattaforma embedded è assai vasto e conosce applicazione in tutti quei settori (consumer fra tutti) in cui si ha bisogno di un mezzo potente,semplice e a basso costo.Per raggiungere l'obbiettivo di aprire questa prospettiva il modello di apprendimento utilizzato sarà SVM, acronimo di Support Vector Machine, un'interessante e promettente tecnologia dotata di una forte e consolidata base teorico-matematica. In SVM, come in molti altri algoritmi, esistono due fondamentali fasi : il training, ed il test. Il training è la fase in cui la macchina impara a risolvere il problema a lei posto. Il test è la messa in opera della macchina, ovvero la fase di utilizzo vera e propria. Il supporto hardware sarà invece il processore Blackfin di Analog Device: si tratta del modello entry level di Analog Device ed è caratterizzato da discrete capacità computazionali,basso costo e consumo. Ritornando a SVM,in questa sede si analizzerà e realizzerà la fase di training: questa consiste sinteticamente nella minimizzazione di una forma quadratica sottoposta ad un vincolo lineare. Come noto esistono diversi algoritmi di soluzione a questo

problema: discesa del gradiente, gradiente coniugato, Baba, SMO e altri. Nella fattispecie verrà utilizzato SMO: questo algoritmo per sua natura risolve in modo analitico il più piccolo sottoproblema risolvibile, quindi risulta particolarmente vantaggioso nel momento in cui le risorse a disposizione siano limitate. Il caso del DSP, ed in generale di un qualsivoglia sistema embedded, è proprio caratterizzato da capacità computazionali limitate, sottosistemi di memoria e consumi ridotti. In questa cornice risulta interessante vedere come un sistema di questo tipo sia in grado di svolgere un compito di relativa complessità e delicatezza in vista anche della rappresentazione numerica da utilizzare. Come si avrà modo di notare, sotto opportune ipotesi restrittive, dovute alla rappresentazione numerica e alle capacità di memoria, i sistemi realizzati saranno in grado di compiere un training corretto, infine verrà proposta una variante di SMO di promettente utilità. Nell'esposizione si partirà da un'inquadratura teorica per poi dettagliare realizzazione, risultati e sviluppi.

Capitolo 2

Introduzione a SVM

2.1 Classificatori e SVM

Il tema della classificazione è ormai uno dei temi e branchie principali del settore dell'Intelligenza Artificiale ed il *Machine Learning*. I primordi della classificazione affondano le loro radici nella persona dell'abate Bayes che nel diciasettesimo secolo definì i concetti di probabilità *a priori*, probabilità *a posteriori* e i fondamentali rapporti che insistono fra loro, fra cui il famoso omonimo teorema. Solo dopo molto tempo (ventesimo secolo) in realtà nacquero in primi classificatori benché, difatto, già tutta la matematica di base da secoli fosse pronta, che nella mera sostanza basilare consiste in : teoria delle probabilità, algebra lineare e calcolo infinitesimale. Fra i primi tentativi ricordiamo il famoso *perceptron algorithm* di Rosenblath : si tratta di un classificatore lineare in cui opportuni pesi definiscono l'iperpiano separatore fra i pattern; questo algoritmo fa parte di una classe di algoritmi in cui il *training* (la fase in cui la macchina impara) avviene secondo il paradigma del *Supervised Learning*. Infatti i pattern vengono proposti alla macchina la quale risponde a che classe (*label* o *target*) questi appartengono, in base alla risposta fornita dalla macchina i pesi vengono di conseguenza cambiati e così il piano separatore definito da essi. Questo algoritmo risulta utile nei casi linearmente separabili (si vedrà meglio dopo) e si inserisce nel sottoinsieme del *Machine Learning* chiamato *Supervised Learning* in cui nel training i pattern sono forniti del *label* ovvero della classe di appartenenza. Nel caso in cui i *pattern* non fossero forniti di *label* si parla di *Unsupervised Learning*. Il vero salto di

qualità matematico avviene con la nascita della statistica : fare statistica significa fare delle inferenze sulle distribuzioni di probabilità associate ad una grandezza, fenomeno o evento. Conoscere le densità di probabilità associate ad un problema significa conoscere "tutto" del problema. Da ciò si vede come uno dei problemi e pilastri fondanti della statistica è la stima delle densità di probabilità (cfr. Fisher). In questa nuova cornice matematica si inquadrano le reti neurali, gli studi di Vapnik e la formulazione della *Statistical Learning Theory* (SLT). Questa teoria ha le sue basi nel concetto di rischio (*Structural Risk Minimization*) e in altri importanti concetti come la dimensione VC (Vapnik - Chervonenkis). SVM si inserisce in questo contesto come l'incarnazione della SLT applicata al tema della classificazione (regressione, stima di densità di probabilità e altre applicazioni sono possibili). Il paradigma è quello del *Supervised Learning* e oggi trova larga applicazione e successo grazie alle proprietà che di seguito verranno esposte. Esistono diverse formulazioni e varianti di SVM (RSVM, ASVM, LSVM et cetera): in questa sede verrà introdotta la formulazione originaria a cui ci si può sempre universalmente riferire (verrà usato il simbolismo di Vapnik).

2.2 Classifcazione lineare

2.2.1 Impostazione del problema

Partiremo dal caso più semplice quello in cui i dati sono linearmente separabili. In questo caso sarà sufficiente trovare l'iperpiano separatore ottimo dei *pattern* x_i tale da rendere massimo il margine (vedi dopo). Chiamata $\mathbf{w}$ la normale all'iperpiano, e b il bias,
i punti che giaciono sull'iperpiano soddisfano l'equazione :

$$\mathbf{w}^T\mathbf{x} + b = 0 \tag{2.1}$$

Adesso indichiamo con d_- e d_+ la minima distanza dall'iperpiano da , rispettivamente , i pattern negativi e i pattern positivi. Chiameremo margine infine la quantità $d_- + d_+$. Si supponga ora che tutti i dati di training soddisfino i seguenti vincoli:

$$\mathbf{w}^T\mathbf{x}_i + b \geq +1 \text{ per } y_i = +1 \tag{2.2}$$

$$\mathbf{w}^T\mathbf{x}_i + b \leq -1 \text{ per } y_i = -1 \tag{2.3}$$

che si possono unire nell'unica espressione :

$$y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right) - 1 \geq 0 \tag{2.4}$$

Ora considerando i punti che soddisfano l' Eq. (2.3) si vede che questi giacciono sull'iperpiano $H_1 : \mathbf{x}_i\mathbf{w} + b = 1$ con normale $\mathbf{w}$ e distanza perpendicolare
$|1 - b|/||\mathbf{w}||$. Analogamente stesso discorso vale per $H_2 : \mathbf{x}_i\mathbf{w} + b = -1$. Quindi $d_- = d_+ = 1/||\mathbf{w}||$ e il margine è $2/||\mathbf{w}||$. Si noti ancora che H_1 e H_2 sono paralleli e in mezzo ad essi non cadono punti del training. Il nostro scopo allora è trovare i due iperpiani che minimizzano $||\mathbf{w}||^2$ soggetto ai vincoli Eq.(2.4). Da cui il seguente problema di minimo (con NP si indicherà il numero di pattern ovvero la dimensione del problema)

$$\begin{cases} \min\limits_{\mathbf{w},b} \dfrac{1}{2}\|\mathbf{w}\|^2 \\ y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right) \geq 1 \quad i = 1, \ldots, n_p \end{cases} \tag{2.5}$$

I punti del training che soddisfano Eq.(2.4) sono chiamati Support Vector che nella figura (fig.2.1) sono indicati con dei cerchietti.

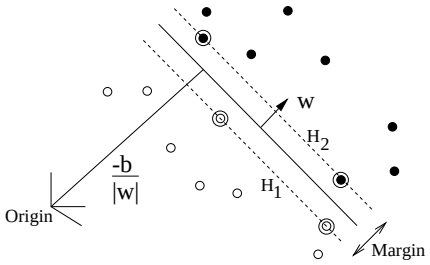

Figura 2.1 – *Iperpiano separatore in* $\Re^2$

Adesso verrà introdotta una formulazione Lagrangiana del problema per due importanti motivi: il vincolo Eq.(2.4) verrà sostituito da un vincolo sui moltiplicatori stessi, in più i dati di training appariranno solo in forma di prodotto scalare ; quest'ultima proprietà è fondamentale in quanto senza di essa il *kernel trick* (si veda dopo) non sarebbe utilizzabile. Introduciamo i moltiplicatori di Lagrange $\alpha_i, i = 1, ..., l$ ovvero uno per ogni vincolo di disuguaglianza (2.4). Questa nuova impostazione porta a questa equazione:

$$L_p(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{n_p} \alpha_i(-y_i(\mathbf{w}^T\mathbf{x}_i + b) + 1) \tag{2.6}$$

Ora bisogna minimizzare L_p rispetto $\mathbf{w}$,b e contemporaneamente richiedere che le derivate di L_p rispetto a tutti gli α_i vadano a 0 soggette ai vincoli $\alpha_i \geq 0$ (chiamiamo questo insieme di vincoli Γ_1). Questo si traduce in un problema di programmazione quadratica convesso dato che : le funzione obbiettivo è essa stessa convessa , e i punti che soddisfano i vincoli sono un set convesso. Alla luce di questo si ha che è possibile risolvere il problema duale associato : massimizzare L_p soggetto al fatto che il gradiente di L_p rispetto $\mathbf{w}$ e b vada a 0 e che vengano soddisfatti i vincoli $\alpha_i \geq 0$ (chiamiamo questo insieme di vincoli Γ_2). Questa doppia formulazione ha di particolare che il minimo di L_p soggetto a Γ_1 ha come soluzione gli stessi α_i b e $\mathbf{w}$ che avrebbe il massimo di L_p soggetto al vincolo Γ_2. Richiedere che il gradiente di L_p vada a zero rispetto $\mathbf{w}$ e b dà le seguenti condizioni:

$$\mathbf{w} = \textstyle\sum_{i=1}^{n_p} \alpha_i y_i \mathbf{x}_i \tag{2.7}$$

$$\textstyle\sum_{i=1}^{n_p} \alpha_i y_i = 0 \tag{2.8}$$

Sosituendo questi vincoli in Eq.(2.6) si ottiene una nuova e più conveniente formulazione:

$$L_d = \sum_{i=1}^{n_p} \alpha_i - \frac{1}{2} \sum_{i=1,j=1}^{n_p} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j \tag{2.9}$$

Si noti che in questo caso L_d è da massimizzare differentemente da L_p che era da minimizzare. Ancora si può notare che porre $b = 0$, che implica il passaggio degli iperpiani per l'origine, significa eliminare il vincolo Eq.(2.8) (inoltre questo avrà interessanti implicazioni quando si parlerà del Kernel RBF). Come detto prima i punti per cui $\alpha_i > 0$ sono detti Support Vectors , giaciono su uno degli iperpiani H_1 , H_2 e sono i punti critici per un SVM , infatti sono quelli che sono più vicini alla frontiera di decisione. Effettuato il training e trovati i $\mathbf{w}$ e b associati , per utilizzare la SVM, è sufficiente valutare $sign(\mathbf{w}\mathbf{x} + b)$ in cui gli x_i appartengono al test set.

2.2.2 Le condizioni di Karush Kuhn Tucker

Le condizioni KKT hanno un ruolo chiave nei problemi di ottimizzazione vincolata. Per il problema primale possono essere formulate così:

$$\frac{\partial}{\partial w_\lambda} L_p = w_\lambda - \sum_{i=1}^{n_p} \alpha_i y_i x_{i\lambda} = 0 \text{ con } \lambda = 1, \ldots, N_i \tag{2.10}$$

$$\frac{\partial L_p}{\partial b} = -\sum_{i=1}^{n_p} \alpha_i y_i = 0 \tag{2.11}$$

$$y_i(x_i \mathbf{w} + b) - 1 \geq 0 \text{ con } i = 1, \ldots, l \tag{2.12}$$

$$\alpha_i(y_i(x_i \mathbf{w} + b) - 1) = 0 \text{ per } \quad i = 1, \ldots, n_p \tag{2.13}$$

$$\alpha_i \geq 0 \quad i = 1, \ldots, n_p \tag{2.14}$$

Queste condizioni sono importanti perché sono soddisfatte per la soluzione vincolata del problema. Si può dimostrare sotto opportune ipotesi,che valgono per qualsiasi problema di ottimizzazione (convesso o meno).A maggior ragione essendo quello di SVM un problema convesso le KKT sono necessarie e sufficienti affinchè $\mathbf{w}$,b, α siano la soluzione. In questo modo si vede che trovare la soluzione di SVM significa trovare una soluzione vincolata dalle KKT. Adesso si consideri il caso di dati non linearmente separabili (fig.2.2). Il metodo fin qui elaborato non sarebbe efficace in quanto non sarebbe in grado di trovare il corretto iperpiano separatore. Ciò che possiamo fare è rilassare i vincoli introducendo delle variabili

$\xi_i > 0$ per $i = 1, \ldots, np$ (Cortes e Vapnik , 1995). I nuovi vincoli diventano:

$$\mathbf{w}^T\mathbf{x}_i + b \geq +1 - \xi_i \text{ per } y_i = +1 \tag{2.15}$$

$$\mathbf{w}^T\mathbf{x}_i + b \leq -1 + \xi_i \text{ per } y_i = -1 \tag{2.16}$$

$$\xi_i \geq 0 \text{ con } i = 1, \ldots, np. \tag{2.17}$$

Da questo si vede che se c'è un errore allora $\xi_i > 1$ da cui $\sum_{i=1}^{np} \xi_i$ è un maggiorante al numero di errori nel training. Allora la funzione costo verrà modificata e le verrà assegnato un costo aggiuntivo , in termini matematici :

$$\frac{||\mathbf{w}||^2}{2} + C(\sum_{i=1}^{np} \xi_i)^k \tag{2.18}$$

Dove il termine C è un iperparametro che pesa gli errori commessi. C è scelto dall'utente è viene scelto secondo la *Model Selection* (cfr.Smola). Con C crescente il peso degli errori cresce e viceversa. Per definizione il problema è ancora un problema di programazione quadratica se $k = 1, 2$: in base a questa scelta sono possibili diverse formulaizioni del problema. Se si sceglie $k = 1$, questa scelta dà il vantaggio che nella forma duale gli ξ_i non compaiono. Il problema diventa :

$$\min(L_d) = \min -(\sum_{i=1}^{np} \alpha_i - \frac{1}{2} \sum_{i=1,j=1}^{np} \alpha_i\alpha_j y_i y_j \mathbf{x_i}\mathbf{x_j}) \tag{2.19}$$

Soggetto ai vincoli :

$$0 \leq \alpha_i \leq C \tag{2.20}$$

$$\sum_{i=1}^{np} \alpha_i y_i = 0 \tag{2.21}$$

La cui soluzione è data da :

$$\mathbf{w} = \sum_{i=1}^{N_s} \alpha_i y_i x_i \tag{2.22}$$

dove con N_s si indica il numero di Support Vector. In $\Re^2$ si può visualizzare così la attuale situazione con la seguente figura. Quindi l'unica differenza dal caso dell'iperpiano ottimo è che gli α_i ora hanno un maggiorante che è l'iperparametro C. Adesso ci serviranno le KKT per il problema primale che in questo caso

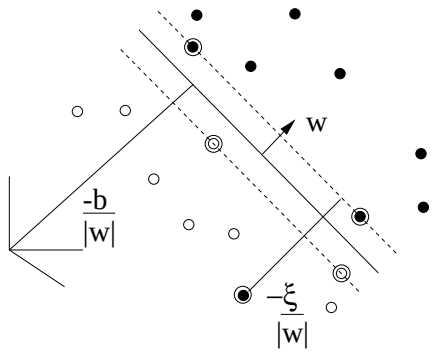

Figura 2.2 – *Dati linearmente non separabili in* $\Re^2$

rispetto al primale prima ottenuto avrà in più il costo associato agli errori pesati dall'iperparametro C, ovvero:

$$L_p(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{np}\xi_i - \sum_{i=1}^{n_p}\alpha_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 + \xi_i] - \sum_{i=1}^{np}\mu_i\xi_i \quad (2.23)$$

Dove μ_i sono i moltiplicatori di Lagrange introdotti per garantire che $\xi_i > 0$. Ecco dunque le KKT per il nuovo primale :

$$\frac{\partial}{\partial w_\lambda}L_p = w_\lambda - \sum_{i=1}^{n_p}\alpha_i y_i x_{i\lambda} = 0 \text{ con } \lambda = 1, \ldots, N_i \quad (2.24)$$

$$\frac{\partial L_p}{\partial b} = -\sum_{i=1}^{n_p}\alpha_i y_i = 0 \quad (2.25)$$

$$y_i(x_i\mathbf{w} + b) - 1 + \xi_i \geq 0 \text{ con } i = 1, \ldots, l \quad (2.26)$$

$$\alpha_i[y_i(x_i\mathbf{w} + b) - 1 + \xi_i] = 0 \text{ per } \quad i = 1, \ldots, n_p \quad (2.27)$$

$$\alpha_i \geq 0 \quad i = 1, \ldots, n_p \quad (2.28)$$

$$\frac{\partial}{\partial \xi_i}L_p = C - \alpha_i - \mu_i = 0 \quad (2.29)$$

$$\xi_i \geq 0 \quad (2.30)$$

$$\mu_i \geq 0 \quad (2.31)$$

$$\mu_i\xi_i = 0 \quad (2.32)$$

$$(2.33)$$

Si noti che mettendo insieme la Eq.(2.32) e la Eq.(2.29) si trova che $\xi_i = 0$ se $\alpha_i < C$. Quindi possiamo semplicemente prendere qualsiasi pattern del training per cui vale $0 < \alpha_i < C$ per poter usare la Eq.(2.27) per determinare b.

2.3 Classificazione non lineare

Se i dati fossero non linearmente separabili il modello creato fino ad ora risulta insufficiente . Nasce cioé l'esigenza di creare un classificatore che abbia maggiore flessibilità e consenta di trattare anche dati non linearmente separabili. Per rendere SVM in grado di fare ciò , verrà utilizzato il cosìddetto *Kernel Trick* : questa tecnica , non recentissima (Aizerman 1964) , permette , in maniera maticamente molto elegante , a SVM di trattare efficacemente questa nuova classe di problemi.Si ricordi che in Eqs.(2.19)-(2.21) i dati di training erano combinati fra loro solo in prodotti scalari e questo sarà fondamentale . Prima di tutto si veda graficamente sempre in $\Re^2$ cosa significa dati non lineramente separabili (in questo caso è stata disegnata una cubica come setto separatore).

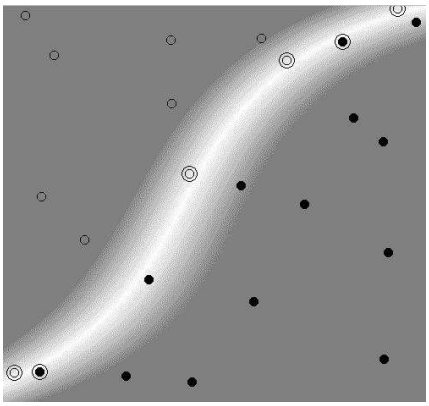

Figura 2.3 – *Dati linearmente non separabili in* $\Re^2$ *separati da una cubica.*

Ora si supponga di conoscere una certa funzione di mapping Φ così definita :

$$\Phi : \Re^d \rightarrow \Psi \tag{2.34}$$

In cui d=N_i e Ψ è uno spazio euclideo (di Hilbert) a un numero imprecisato di demensioni. Abbiamo tuttavia visto e sottolineato , che i dati compaiono solo in forma di prodotti scalari e questo vale sia nello spazio originari che in Ψ, con la differenza che qui il prodotto scalare avrà la forma $\Phi(\mathbf{x_i}) * \Phi(\mathbf{x_j})$. Da questa osservazione si nota che in realtà non serve conoscere singolaremente $\Psi()$, è invece sufficiente il suo prodotto scalare : $K(\mathbf{x_i}, \mathbf{x_j}) = \Phi(\mathbf{x_i}) * \Phi(\mathbf{x_j})$ dove con $K(x_i, x_j)$

indichiamo la funzione detta Kernel. Citiamo per esempio il kernel RBF (Radial Basic Function alias gaussiane) che si trova qui definito :

$$K(\mathbf{x_i}, \mathbf{x_j}) = exp[-\frac{||\mathbf{x_i} - \mathbf{x_j}||^2}{2\sigma^2}] \tag{2.35}$$

in questo caso Ψ ha infinite dimensioni e sarebbe complicato lavorare esplicitamente con Ψ. In sintesi è sufficiente mettere ovunque al posto di $\mathbf{x_j} * \mathbf{x_j}$ il $K(\mathbf{x_i}, \mathbf{x_j})$ nell'algoritmo di training : questo farà in modo che la SVM effettuerà il training in uno spazio ad infinite dimensioni (caso RBF). Tutto quello che è stato fin qui detto sul caso lineare vale ancora , infatti si sta effettuando ancora una separazione lineare ma in uno spazio diverso. Per quanto riguarda la fase di test, nella formulazione precedente, appariva ancora un prodotto scalare e si valutava il segno di $(\mathbf{wx} + b)$: come per il training, candidamente, si sostituirà il prodotto scalare con il kernel rendendo il test consistente con il training. Ricordando che: $\mathbf{w} = \sum_{i=1}^{Ns} \alpha_i y_i \mathbf{x_i}$ Si avrà:

$$f(x) = \sum_{i=1}^{Ns} \alpha_i y_i \Phi(\mathbf{s_i}) * \Phi(\mathbf{x}) + b = \sum_{i=1}^{Ns} \alpha_i y_i K(\mathbf{s_i}, \mathbf{x}) + b \tag{2.36}$$

dove con $f(x)$ si è indicata la funzione di decisione , ovvero la funzione il cui segno determina l'appartenenza all'una o all'altra classe e con s_i si indicano i support vector trovati nel training. Ovviamente non tutte le funzioni possono essere dei Kernel: per essere Kernel devono soddisfare le condizioni di *Mercer* che danno vita al seguente fondamentale teorema.

Teorema 2.3.1. *Esiste un mapping* Φ *e la relativa espansione :*
$K(x, y) = \sum_{i=1}^{Ns} \Phi(x)_i \Phi(y)_i$ *, se e solo se , per ogni* $g(x)$ *tale che* $\int g(x)^2 dx$ *abbia valore finito allora:* $\int K(x, y) g(x) g(y) dx dy \geq 0$

Questa condizione coincide in pratica, a garantire che l'hessiana associata sia definita positiva in modo da garantire il minimo globale. I kernel più usati sono :

lineare →

$$\mathbf{x}_i \mathbf{y}_i \tag{2.37}$$

polinomiale →

$$[1 + \mathbf{x}_i \mathbf{y}_i]^p \tag{2.38}$$

RBF →

$$exp[-\frac{||\mathbf{x_i} - \mathbf{x_j}||^2}{2\sigma^2}] \tag{2.39}$$

neurale →

$$tanh(\xi \mathbf{xy} - \delta) \tag{2.40}$$

Ne esistono molti altri e fra l'altro si può dimostrare che combinazioni lineari di kernel sono ancora kernel . Sul kernel neurale Vapnik ha notato sperimentalmente che ci sono alcuni problemi sull'hessiana associata che a volte risulta mal condizionata : fra le varie soluzioni citiamo la più semplice (adottata da Lin) ovvero quella in cui si forza con opportuni coefficienti correttivi l'hessiana ad essere semidefinita positiva.

Adesso si posseggono tutti gli strumenti minimi per costruire un SVM : ciò che in realtà ancora manca è la tecnica di minimizzazione della forma quadratica ovvero il cuore pulsante del training la cui efficacia determina insieme ad altre scelte (kernel, C , tolleranze..) le future capacità di classificazione della macchina. Ecco dunque che serve un algoritmo di minimizzazione: fra quelli proposti (e sono diversi) in questo lavoro si è utilizzato SMO per i motivi che di seguito verranno esposti, e con i relativi risultati.

Capitolo 3

Sequential Minimal Optimation

3.1 Versione originale

SMO è un algoritmo di ricerca di minimo che decompone l'intero problema QP in tanti sotto problemi QP in modo simile a quanto fatto da Osuna ed è stato elaborato da J.Platt, ricercatore Microsoft. Come già accennato nell'introduzione generale, SMO sceglie di risolvere il più piccolo problema ad ogni iterazione.Per la SVM standard SMO risolve il più piccolo problema di ottimizzazione che coinvolge 2 moltiplicatori di Lagrange, perché i moltiplicatori sono soggetti a un vincolo di uguaglianza lineare. Ad ogni passo, SMO sceglie quali siano i moltiplicatori da ottimizzare, ne trova il valore ottimo, e aggiorna il gradiente e b in modo da riflettere i cambiamenti avvenuti. Il vantaggio principale di SMO risiede nel fatto che il problema di ottimizzazione per 2 moltiplicatori può essere risolto analiticamente. Questo aspetto fa si che non ci sia un loop interno di calcolo numerico, cosa che tipicamente accade in algoritmi di ottimizzazione: questo rende il codice da realizzare abbastanza semplice.Essendo la velocità di soluzione dei singoli sottoproblemi alta, la velocità di soluzione generale è anch'essa alta rispetto ai metodi classici di chunking. Dato che per risolvere il problema si manipolano matrici $2x2$ questo fa si che l'algoritmo complessivamente sia meno sensibile a problemi di rappresentazione numerica. Sono dunque tre le parti principali di SMO:

- Un metodo analitico per ottimizzare i 2 moltiplicatori

- Un euristica per trovare i moltiplicatori candidati all'ottimizzazione

- Un metodo per calcolare b.

3.1.1 Soluzione analitica

Per trovare la soluzione dei due moltiplictori, SMO prima calcola i vincoli e dopo risolve per il massimo vincolato.Per convenzione tutte le quantità riferite al primo moltiplicatore avranno pedice 1, quelle riferite al secondo pedice 2. Graficamente nella seguente figura si vede quella che tipicamente è chiamata la box (ovvero il dominio in cui poter trovare i moltiplicatori) delimitata dal penalty C, e la retta che rappresenta il vincolo lineare e che limita ulteriormente alla retta stessa il dominio dei moltiplicatori.

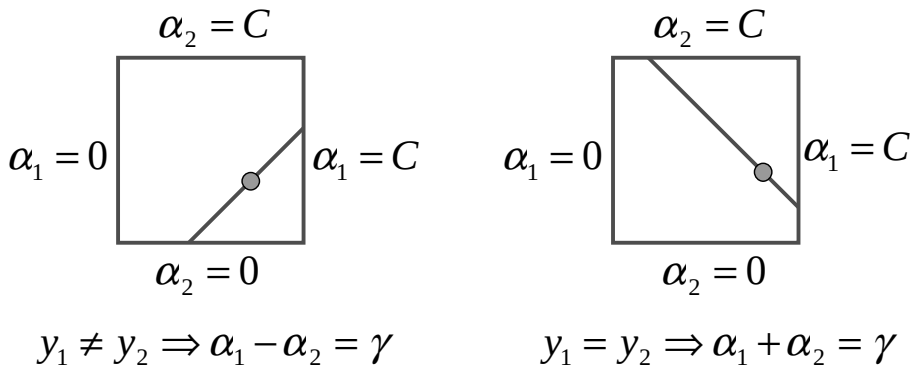

Figura 3.1 – *La* box *con il vincolo lineare.*

Proprio questo vincolo spiega perché 2 è il numero minimo di moltiplicatori che possono essere ottimizzati: se SMO ottimizzasse solo un moltiplicatore, non sarebbe in grado di soddisfare il vincolo di uguaglianza lineare ad ogni passo. Senza perdita di generalità, l'algoritmo prima computa il moltiplicatore α_2 poi computa le intersezioni della linea diagonale in termini di α_2. Se il target y_1 è diverso da il target y_2, allora si applicano i seguenti maggioranti ad α_2:

$$L = \max(0, \alpha_2^{old} - \alpha_1^{old}), \quad H = \min(C, C + \alpha_2^{old} - \alpha_1^{old}). \tag{3.1}$$

Se invece il target $y_1 = y_2$, sono applicati questi altri maggioranti:

$$L = \max(0, \alpha_2^{old} + \alpha_1^{old} - C), \quad H = \min(C, \alpha_2^{old} + \alpha_1^{old}). \tag{3.2}$$

Adesso invece cerchiamo una regola iterativa del calcolo dei nuovi α_i. La derivata seconda della funzione obbiettivo lungo la linea diagonale può essere espressa così:

$$\eta = 2k(\mathbf{x}_1, \mathbf{x}_2) - k(\mathbf{x}_1, \mathbf{x}_2) - k(\mathbf{x}_1, \mathbf{x}_2). \tag{3.3}$$

Data questa formulazione il passo successivo di SMO è calcolare l'ubicazione del massimo vincolato della funzione obiettivo Eq.(2.19) consentendo a solo due moltiplicatori di cambiare. In circostanze normali, ci sarà un massimo lungo la direzione del vincolo, e η sarà minore di 0. In questo caso, SMO calcola il massimo lungo la direzione del vincolo:

$$\alpha_2^{new} = \alpha_2^{old} - \frac{y_2(E_1 - E_2)}{\eta} \tag{3.4}$$

dove $E_i = f^{old}(\mathbf{x}_i) - y_i$ è l'errore nell'i-esimo pattern di training. Successivamente,il massimo vincolato è trovato saturando il massimo non vincolato alle intersezioni delle linea del vincolo con la box.

$$\alpha_2^{new_{saturato}} = \begin{cases} H, \; se \; \alpha_2^{new} \geq H; \\ \alpha_2^{new}, \; se \; L < \alpha_2^{new} < H; \\ L, \; se \; \alpha_2^{new} \leq L. \end{cases} \tag{3.5}$$

Ora, sia $s = y_1 y_2$. Il valore di α_1 è calcolato dal nuovo e saturato, α_2:

$$\alpha_1^{new} = \alpha_1^{old} + s(\alpha_2^{old} - \alpha_2^{new,saturato}). \tag{3.6}$$

3.1.2 Euristica di scelta dei moltiplicatori

SMO ottimizzerà sempre due moltiplicatori ad ogni passo, con uno dei due che ha precedentemente violato le KKT prima del passo attuale.Ciò significa che SMO modificherà 2 moltiplicatori per muoversi in *hill climbing* nella funzione obiettivo proiettata nel sottospazio *feasible* monodimensionale. SMO manterrà sempre un vettore *feasible* di moltiplicatori. Per questo la funzione costo aumenterà a ogni passo e l'algoritmo convergerà asintoticamente. Per velocizzare la convergenza, SMO utilizza un'euristica con cui scegliere i moltiplicatori di Lagrange da ottimizzare. Ci sono due separate euristica di scelta: una per il primo moltiplicatore, ed una per il secondo.La scelta della prima euristica riguarda il

loop più esterno di SMO.Il loop più esterno prima itera sull'intero training set, determinando quali pattern violino le KKT.Se qualche pattern viola le KKT,è candidato per immediata ottimizzazione. Appena si trova un candidato violatore, un secondo moltiplicatore è scelto usando la seconda scelta euristica, e sono ottimizzati congiuntamente.L'SVM è aggiornata (il gradiente,b) usando questi due nuovi moltiplicatori, e il loop più esterno ricomincia a cercare i violatori. Una volta scelto il primo moltiplicatore, SMO sceglie il secondo per massimizzare la grandezza del passo effettuato durante l'ottimizzazione congiunta. Calcolare il kernel è dispendioso in termini computazionali, così SMO approssima la grandezza del passo con il valore assoluto del numeratore di Eq.(3.4) : $E_1 - E_2$. Se $E_1 > 0$, SMO sceglie un pattern con errore minimo E_2. Se $E_1 < 0$, SMO sceglie un pattern con massimo errore E_2. In circostanze atipiche, SMO non può fare progressi usando la seconda scelta euristica appena descritta.Per esempio, non può esserci un progresso positivo se il primo e secondo pattern condividono le stesse componenti x_i, infatti questo causa che la funzione obiettivo diventi piatta nella direzione di ottimizzazione. Per evitare questo problema, SMO usa una gerarchia di seconde scelte euristiche fino a che non trova un paio di moltiplicatori che possono dare un progresso positivo. Per progresso positivo si intende un passo non nullo nell'ottimizzazione congiunta. La gerarchia della seconda scelta euristica consiste in :

- se l'euristica fallisce,allora SMO comincia a iterare i pattern non-bound cercando un secondo pattern che possa dare un passo non nullo
- se nessuno dei non-bound dà progressi, allora SMO itera su tutto il training set.

In entrambi i casi il punto di inizio di ricerca dei moltiplicatori è casuale. Il loop esterno itera finchè tutti i pattern soddisfano le KKT entro una tolleranza ϵ (valori tipici sono $10^{-2}10^{-3}$), a quel punto l'algoritmo termina.

3.1.3 Calcolo del bias b

Dopo ogni passo di ottimizzazione b è nuovamente computato in modo tale che le KKT siano soddisfatte per entrambi i pattern ottimizzati. La seguente soglia

b_1 è valida quando il nuovo α_1 non è ai limiti della box, perché forza l'uscita di SVM ad essere y_1 , quando l'ingresso è x_1:

$$b_1 = E_1 + y_1(\alpha_1^{new} - \alpha_1^{old})k(x_1, x_1) + y_2(\alpha_2^{new,saturato} - \alpha_2^{old})k(x_1, x_2) + b^{old} \quad (3.7)$$

La seguente soglia b_2 è valida quando il nuovo α_2 non è ai limiti della box, perché forza l'uscita di SVM ad essere y_2, quando l'ingresso è x_2:

$$b_2 = E_2 + y_1(\alpha_1^{new} - \alpha_1^{old})k(x_1, x_2) + y_2(\alpha_2^{new,saturato} - \alpha_2^{old})k(x_2, x_2) + b^{old} \quad (3.8)$$

Quando b_1, b_2 sono validi sono anche uguali.Quando i nuovi moltiplicatori di Lagrange sono al limite della box e L è diverso da H, allora i b_i appartenenti $[b_1, b_2]$ sono tutte soglie consistenti con le KKT. In tal caso SMO sceglie la soglia come: $(b_1 + b_2)/2$.

3.2 Variante di Keerthi

Questa variante consente attraverso una riformulazione delle KKT, di eliminare il calcolo di b_i ad ogni iterazione (come invece succede nell'algoritmo di J.Platt) e di definire un criterio di stop.Questa variante è la più utilizzata per gli incrementi di velocità che garantisce. Definendo $F_i = f(x_i) - y_i - b$ le KKT possono essere così riformulate:

$$\begin{aligned} y_i\,(F_i + b) &\geq 0 \qquad \text{per} \qquad \alpha_i = 0 \\ y_i\,(F_i + b) &= 0 \qquad \text{per} \qquad 0 < \alpha_i < C \\ y_i\,(F_i + b) &\leq 0 \qquad \text{per} \qquad \alpha_i = C \end{aligned} \tag{3.9}$$

Ora si definiscono degli opportuni insiemi di indici:

$$I_0 = \{i : 0 < \alpha_i < C\} \tag{3.10}$$

$$I_1 = \{i : y_i = +1, \alpha_i = 0\} \tag{3.11}$$

$$I_2 = \{i : y_i = -1, \alpha_i = C\} \tag{3.12}$$

$$I_3 = \{i : y_i = +1, \alpha_i = C\} \tag{3.13}$$

$$I_4 = \{i : y_i = -1, \alpha_i = 0\} \tag{3.14}$$

Inoltre chiamiamo I l'unione di tutti gli insiemi:

$$I = I_0 \cup I_1 \cup I_2 \cup I_3 \cup I_4 \tag{3.15}$$

Ponendo ora $b_i = -F_i$ e ponendo $y_i = -1$ e $y_i = +1$ nei diversi casi, le KKT si possono così riassumere:

$$\begin{aligned} b &\leq b_i \qquad \forall i \in I_0 \cup I_3 \cup I_4 \\ b &\geq b_i \qquad \forall i \in I_0 \cup I_1 \cup I_2 \end{aligned} \tag{3.16}$$

Ovviamente le relazioni devono essere soddisfatte per tutti gli i. Dunque i termini da tenere in considerazione sono rispettivamente:

$$\begin{aligned} b_{low} &= \max_i b_i \\ b_{up} &= \max_i b_i \end{aligned} \tag{3.17}$$

e per la validità delle KKT, deve risultare:

$$b_{up} > b_{low} \tag{3.18}$$

diversamente le KKT non sono soddisfatte e bisogna continuare l'ottimizzazione. Graficamente possiamo così definire la situazione attuale:

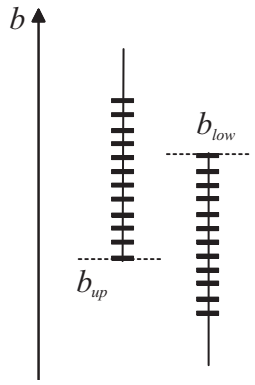

Figura 3.2 – *I valori di b durante l'ottimizzazione.*

Visto che se $b_{up} > b_{low}$ le KKT sono soddisfatte allora abbiamo immediatamente fornito un criterio di stop. Come al solito si può definire una tolleranza ϵ per il criterio di stop:

$$b_{up} - b_{low} \leq \epsilon \tag{3.19}$$

Ancora conoscere b_{low} e b_{up} ci consente di avere una semplicissima euristica di scelta dei moltiplicatori da ottimizzare.Essendo b_{low} e b_{up} i valori di b critici per la terminazione, sarà sufficiente prendere i 2 moltiplicatori associati a b_{low} e b_{up} e ottimizzarli.

3.3 Realizzazione di Linn

C.J.Lin con *LibSVM* ha dato una completa realizzazione di SVM atta a risolvere diversi problemi con diverse varianti di SVM. L'interesse per *LibSVM* nasce dal fatto che il codice originario in *fortran* è basato su quello di Lin però, rispetto ad esso, è semplificato perchè non è utilizzato lo *shrinking*(una tecnica usata per ridurre il numero di moltiplicatori possibili candidati all'ottimizzazione) .*LibSVM* utilizza SMO con la variante proposta da Keerthi: ciò che cambia è che tutto è riformulato e calcolato in termini del gradiente. Infatti trasponendo in termini del gradiente le KKT riformulate da Keerthi Eq.(3.9) si ottiene:

$$\begin{aligned} (\nabla f(\alpha) + b\mathbf{y})_i \geq 0 \qquad & \text{per} \qquad \alpha_i = 0 \\ (\nabla f(\alpha) + b\mathbf{y})_i = 0 \qquad & \text{per} \qquad 0 < \alpha_i < C \\ (\nabla f(\alpha) + b\mathbf{y})_i \leq 0 \qquad & \text{per} \qquad \alpha_i = C \end{aligned} \tag{3.20}$$

Considerando ora esplicitamente tutti i casi con $y_i = 1, -1$ ed il bias b, si ottengono le seguenti sintetiche condizioni di scelta dei moltiplicatori :

$$i = \arg\max([-\nabla f(\alpha_i)|y_i = 1, \alpha_i < C)], [\nabla f(\alpha_i)|y_i = -1, \alpha_i > 0]) \tag{3.21}$$

$$j = \arg\min([-\nabla f(\alpha_i)|y_i = -1, \alpha_i < C)], [\nabla f(\alpha_i)|y_i = +1, \alpha_i > 0]) \tag{3.22}$$

dove i e j sono gli indici dei moltiplicatori candidati all'ottimizzazione. Il criterio di stop risulta dunque così modificato:

$$g_i \leq g_j + \epsilon \tag{3.23}$$

dove ϵ è la solita tolleranza ed è sempre dell'ordine $10^{-2}, 10^{-3}$. Inoltre le regole di aggiornamento degli α diventano le seguenti :

$$\alpha_j^{new} = \begin{cases} \alpha_j + \frac{-G_i - G_j}{Q_{ii}+Q_{jj}+2Q_{ij}} \; se \; y_i \neq y_j \\ \alpha_j + \frac{G_i - G_j}{Q_{ii}+Q_{jj}-2Q_{ij}} \; se \; y_i = y_j \end{cases} \tag{3.24}$$

con ovviamente $G_i = \nabla f(\alpha)_i$ e $G_j = \nabla f(\alpha)_j$. Se α_j è fuori dalla box come al solito viene saturato.

Per esempio se $y_i = y_j$ e $C \leq \alpha_i + \alpha_j \leq 2C$ allora α_j^{new} deve soddisfare $L = \alpha_i + \alpha_j - C \leq \alpha_j^{new} \leq C = H$. Dato che α_i^{new} e α_j^{new} hanno come maggiorante C, allora definiamo $\alpha_j^{new} = L$ e $\alpha_i = \alpha_i + \alpha_j - \alpha_j^{new} = C$. Tuttavia numericamente

essendo $\alpha_i + \alpha_j - (\alpha_j + \alpha_i - C) \neq C$ spesso accade che $\alpha_i \geq C - \epsilon_{new}$ siano considerati bounded support vector e gli $\alpha_i \leq \epsilon_{new}$ siano considerati zero, dato che tipicamente ϵ_{new} è molto piccolo. Hsu e Lin hanno dimostrato che se tutti gli α_i bounded ottengono i loro valori per assegnazione diretta allora la tolleranza ϵ_{new} è eliminabile.

Con questa formulazione di Lin e la sua successiva semplificazione è nato il codice *fortran* originario che costituisce il punto di partenza di questo lavoro.

Capitolo 4

Digital Signal Processing

4.1 Caratteristiche generali dei DSP

Questo tipo di processori per le loro peculiarità di velocità,efficienza, basso costo e consumo da anni ormai domina il mercato delle periferiche e dei sistemi embedded. Il nome DSP indica in modo preciso la natura di questi componenti hardware : Digital è sinonimo di robustezza, Signal significa che le operazioni svolte sono tipicamente su flussi di dati e che quindi si tratta generalmente di applicazioni *number crunching*, Processor indica la ovvia natura dell'elaboratore. Questo tipo di hardware offre un ottimo supporto, ad alcuni problemi tipici che questo sistema è in grado di risolvere efficientemente. Casi classici di applicativi sono : filtri FIR,IIR, FFT, elaborazione audio/video. Adesso anche i O.S. Real Time (e.g. μLinux) di un certo livello in termini di layers di astrazione software (ovviamente non si parlerà di ambienti di esecuzione gestiti, ma di μKernel). Questi dispositivi, adesso, anche nelle fasce basse di mercato, esibiscono velocità di tutto rispetto in termini computazionali e frequenze di clock. Utilizzando codici altamente specializzati (intrinsic), e per precisi domini applicativi, questi sistemi offrono prestazioni di tutto rispetto e del tutto adeguate per il tipo di applicazioni a cui sono destinati. In generale un DSP ha, di fatto, una notevole serie di tecnologie che lo rendono particolarmente adatto negli applicativi a flusso di dati intenso e scarso controllo. La quantità e l'estensione pervasiva di queste tecnologie nell'architettura del DSP, dipende dal produttore, dal costo, ed ovviamente dal settore di mercato: nel nostro caso verrà utilizzato il Blackfin, processore entry

level in termini architetturali,prestazionali e di costo, della Analog Device. Se volessimo sinteticamente esporre le differenze architetturali fra DSP e general purpouse si potrebbe seguire come riferimento la seguente tabella.

Dsp	**General purpouse**
VLIW	VLIW presente solo negli ultimi modelli
Harvard	no Harvard
sistema MAC	no sistema MAC
aritmetica saturata	aritmetica non saturata
scheduling a ct	scheduling [c,r]time.
hardware loop control	no hardware loop control
ridotta cache	amplia cache
no esecuzione speculativa	si esecuzione speculativa
no branch prediction	si branch prediction
assembly semplice	assembly complesso
JTag	no JTag
Costo: 1-100	Costo: 100-800
bassi consumi	alti consumi
Alta longevità (10-20 anni)	Bassa longevità (1-2 anni)
Rappr. fixed-point	Rappr. floating-point

Tabella 4.1 – *Comparativa Dsp, GP.*

Esplicitiamo ora una ad una le caratteristiche dei DSP.

VLIW

VLIW è acronimo di *V*ery *L*ong *I*nstruction *W*ord: questa tecnologia consente di impacchettare in un unico blocco più istruzioni (tipicamente da 32 bit) e di mandarle dalla memoria istruzioni, al core del DSP in una singola ciclo di clock (questo è tipico, esistono tuttavia diverse soluzioni). Questo fa si che l'elaboratore possa, ove possibile, estrarre il massimo parallelismo: per fare questo generalmente ogni istruzione è dotatata di un *linking bit* ovvero un bit il cui valore determina la possibilità o meno di parallelismo fra l'istruzione corrente e quella che segue. In alcune architetture, accade anche che esista un'intera istruzione

fittizia in più nel pacchetto che funge da pacchetto di flag aggiuntivi per dare ulteriori indicazioni computazionali al core dell'elaboratore. Il VLIW è tipicamente presente su tutti i più recenti DSP: una forma tipica è il 8x32, ovvero un pacchetto ha 8 istruzioni, ognuna da 32 bit; oppure 8x16 o anche, l'atipico e più unico che raro, 2x16+32 del Blackfin, cioé due istruzioni a 16 bit ed una a 32 bit (il Blackfin ha un instruction set misto a 32 e 16 bit). I General Purpouse solo recentemente hanno cominciato ad utilizzare questa tecnologia (P III). Chi ne ha fa recentemente un uso massiccio con un'architettura profondamente innovativa rispetto a tutti i processori di classe x86 è l'Itanium di Intel Inside che per la sua architettura è di difficile classificazione, visto che prende il meglio da molti paradigmi architetturali diversi.

Architettura Harvard

Si tratta di un'architettura di memoria che consente l'accesso simultaneo a memoria dati e memoria istruzioni (diversamente dello schema classico Von Neumann). L'accesso può essere effettivamente simultaneo (stesso ciclo e fronte) o simultaneo in senso lato: molto spesso per simultaneo in senso latoi intendono 2 letture in memoria nello stesso ciclo ma su fronti diversi. Generalmente l'architettura Harvard è realizzata così per evitare di dover utilizzare, costose memorie doppia porta. I DSP godono di questa tecnologia, i GP no(sarebbe troppo costoso). Esiste inoltre la così detta Super Harvard ovvero una Harvard con in più un bus dedicato alle periferiche I/O che permette un ulteriore livello di parallelismo (lo SHARC,fratello maggiore del Blackfin adotta questa tecnologia, infatti SHARC è acronimo di *S*uper *H*arvard *Arc*hitecture).

MAC

La sigla MAC indica invece la capacità di fare una moltiplicazione ed una somma in un singolo ciclo di clock. In pratica una MAC altro non è che la singola iterazione di un for che esegue un prodotto scalare :$\sum_i x_i y_i$. Generalmente i processori GP non sono in grado di effettuare quest'operazione in un singolo ciclo di clock.

Aritmetica saturata

L'aritmetica saturata è sicuramente uno dei fattori qualitativi più apprezzabili dei DSP. Il concetto di base è molto semplice: questa aritmetica consente nel caso di overflow di pagare nel risultato il minor errore possibile. Si immagini di avere 4 bit a disposizione per la rappresentazione di una grandezza unsigned: su un normale GP l'operazione 0xF +0x1 causerebbe un overflow dando come risultato 0x0, nel caso del DSP la saturazione garantisce che 0xF+0x1 dia come risultato quello che garantisce il minore errore ovvero 0xF.Questa tecnologia non è presente nei GP e sembra, per ora, abbastanza difficile che venga realizzata, in vista soprattuto del range dei floating-point che è estremamente ampio. Si ricorda che nei DSP l'aritmeitca è generalmente fixed-point, quindi la saturazione a maggior ragione risulta utile in questi sistemi.

Scheduling

Lo scheduling del codice è effettuato nel caso del DSP a compile time dal compilatori. Nei GP il compilatore effettua un soft scheduling cioé non si preoccupa di ottenere la massima efficienza, in quanto il processore stesso con le sue strutture hardware (buffer di riordino,Scoreboard, algoritmo di Tomasulo,rinomina dei registri,esecuzione speculativa et cetera..) è ampliamente in grado di capire la configurazione migliore di esecuzione. L'hardware scheduling è costoso ed è difficile da realizzare; questi fattori non consentono di realizzare queste soluzioni in hardware, visto che il DSP dovrà avere un costo contenuto.

Gestione dei loop

I loop nel caso dei DSP sono solitamente eseguiti in hardware: esiste un buffer chiamato *repeat buffer* che si occupa di memorizzare le istruzioni da ripetere, inoltre la gestione dello stop del ciclo è gestita interamente con strutture hardware. Questa tecnologia ha notevolissimi vantaggi in quanto non esiste più il problema di aggiornamento dell'indice, del controllo di fine loop: in particolare ogni volta che si fa un'iterazione normalmente bisognerebbe controllare con una variabile ed un if lo stato del loop, tuttavia fare un if significa interrompere la

pipeline con delle bolle diminuendo drasticamente l'efficienza computazionale. I GP generalmente non adottano questa tecnologia.

Cache

La memoria cache è tipicamente un componente molto costoso e molto ingombrante in termini di area del wafer.Si ricorda infatti che il costo di un chip aumenta con il cubo dell'area, ed essendo la cache fortemente *area consuming* si vede immediatamente come una soluzione con grandi cache sia poco praticabile in un DSP a basso costo. Nei GP attuali le cache (con le diverse gerarchie) si sono fortemente espanse andando per raddoppi ogni due anni mediamente. Queste avranno sempre maggiore influenza e importanza visto che, allo stato attuale, cominciano a nascere problemi per l'innalzamento delle frequenze, da cui godere di un'architettura efficiente è l'unico modo per rimanere competitivi.

Esecuzione speculativa

è una tecnica avanzata di esecuzione usata dai GP. Questa tecnica consente di seguire contemporaneamente due token di controllo nati da un fork dovuto ad un if: non appena l'if è risolto la parte di codice eseguita e scorretta si butta, mantenendo e continuando l'esecuzione della parte corretta.Tipicamente questi GP hanno almeno due pipeline che si occupano di seguire i due token, ovviamente una delle due subirà un pipe flush. Questa tecnologia è estremamente utile quando l'applicazione è di tipo *data management* (db) e questo non è il caso dei DSP, inoltre anche questa tecnica in hardware è complessa da realizzare, per cui viene realizzata solo in alcune famiglie particolarmente avanzate di DSP.

Branch Prediction

I DSP tipicamente non posseggono una tecnica di previsione dei salti (dinamica), cosa che i GP hanno. Nel migliore dei casi, con delle direttive del compilatore, si può suggerire al processore se il branch è probabile o meno.

Assembly

Un altro fondamentale aspetto riguarda la semplicità dell'assembly: per quanto programmare in assembly non sia mai banale, non di rado per ottenere il massimo delle prestazioni i programmatori di DSP, usano l'assembly. Proprio per questa larga diffusione, l'assembly per DSP è effettivamente molto pulito ed è sempre basato su un'architettura RISC stile ARM (T5,T6,T7 et cetera..). Nel caso dei GP questi sono tipicamente dei CISC con codice assembly molto complicato. Altri GP in effetti sono basati su architetture RISC (Motorola G3,G4,G5) e per questo pur essendo dei GP hanno degli assembly che, per il settore di appartenenza, possono essere definiti semplici. Non è casuale che le architetture RISC, abbiano a parità di frequenza rispetto ai CISC, tipicamente, un'efficienza doppia (cfr.PIII,G3).

JTag

è un protocollo di comunicazione fra il DSP ed il mondo esterno, dove per mondo esterno si intende tipicamente il debugger. Con questa tecnologia si è grado di sapere istante per istante lo stato dei registri di architettura, contatori, watchdog, convertitori ovvero lo stato del core del processore e le periferiche ad esso collegate che supportano il protocollo. Anche questa tecnologia è appannaggio dei soli DSP.

Costo unitario

Generalmente esiste un ordine di grandezza fra la il costo di un DSP e di un GP. Se vengono venduti milioni di pc è altrettanto certo che vengono venduti miliardi dei soli cellulari: ciò significa che il DSP nasce, categoricamente con l'intento di servire un pubblico vastissimo tutto incentrato su sistemi embedded. Nei GP questa gara al ribasso in termini di prezzo non esiste in quanto c'è una situazione di evidente semi-monopolio da parte di Intel.

Consumi

Dato che i DSP tipicamente vivono su periferiche compatte e sopratutto *mobile*, consumo e dissipazione di potenza devono essere particolarmente ridotti.

Nei GP questo problema non esiste, visto che si tratta di un prodotto destinato ad un pubblico che generalmente non pone nel *Total Cost of Ownership* (TOC) i consumi dei PC.

Longevità

I GP inoltre, hanno altre salienti caratteristiche: molto spesso capita che i nuovi modelli di vengono concepiti con sempre nuove piedinature (e nuove specifiche elettriche, di bus ecc) incompatibili con i modelli precedenti. Questa politica è applicabile ad un pubblico che non ha bisogno di continuità tecnologica: in un sistema industriale non è neanche lontanamente pensabile un atteggiamento del genere, visto che spesso bisogna garantire continuità e supporto tecnologico per ventenni.I DSP ben rispondono a questa necessità con alti livelli di longevità, intesa come compatibilità hardware e supporto software.

Rappresentazione numerica

Ultimo elemento ma non per importanza riguarda la rappresentazione numerica. I GP sono ottimizzati per calcoli a interi e floating-point (IEEE 754). Nel caso dei DSP si ha per elaboratori di fascia bassa il supporto interi fixed-point, per quelli di fascia alta interi e floating-point. In entrambi casi la saturazioni è presente. Il costo di realizzazione di unità floating-point è generalmente non minimo, quindi spesso la presenza o meno del floating-point determina la fascia di appartenenza dell'elaboratore.

4.2 Il Blackfin

Il Blackfin è il DSP che è stato utilizzato per questo lavoro. Il sistema ADSP-BF533 EZ-Lite è la scheda di sviluppo che è stata utilizzata. Le caratteristiche principali del Blackfin revisione 0.3 sono le seguenti

- Frequenza operativa del core fino a 750 Mhz (rev. 0.3)
- 2 MAC a 16 bit, 2 ALU a 40 bit, 4 ALU video a 8 bit, 40 bit shifter
- Modello di istruzioni tipo RISC
- 0.8-1.2 V tensione V_{dd} del core con sistema on-chip di regolazione della tensione
- 148 Kb di memoria nel chip
- Doppio canale DMA verso la memoria
- Supporto per memorie esterne SDRAM,SRAM,FLASH e ROM.

inoltre sono disponibili le seguenti porte e protocolli: PPI/GPIO,12 canali DMA sulle periferiche,SPI,UART con supporto IrDA; ancora si hanno : RTC, un Watchdog Timer , JTAG , PLL on-chip 1x 63x fattore di moltiplicazione, timer del core.

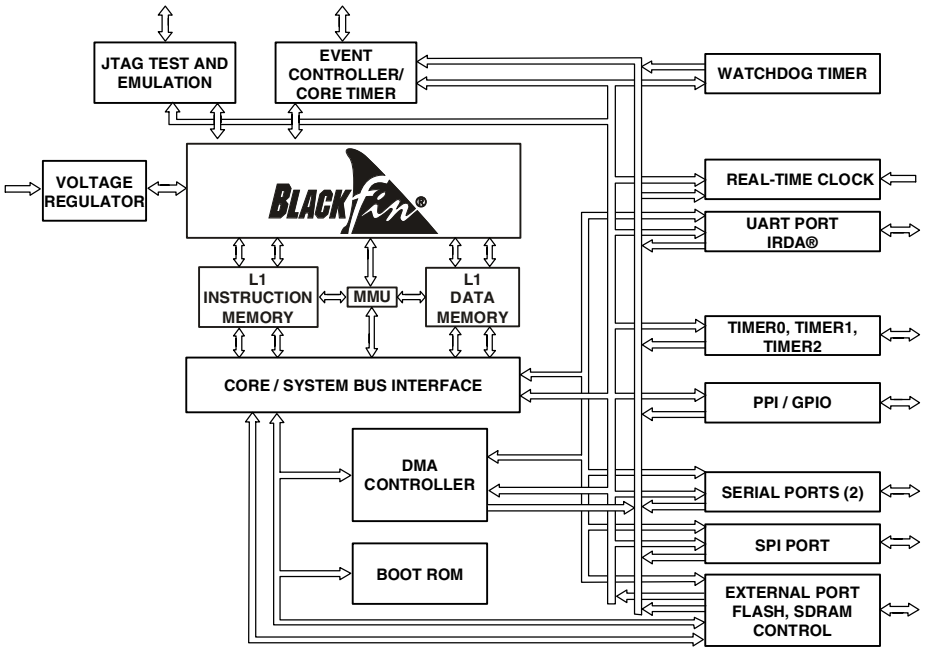

Figura 4.1 – *Schema a blocchi funzionale di Blackfin*

Esistono diversi chip Blackfin, qui si trova una comparativa: il core utilizzato sarà il BF533

	ADSP-BF531	ADSP-BF532	ADSP-BF533
Maximum Performance	400 MHz 800 MMACs	400 MHz 800 MMACs	600 MHz 1200 MMACs
Instruction SRAM/Cache	16K bytes	16K bytes	16K bytes
Instruction SRAM	16K bytes	32K bytes	64K bytes
Data SRAM/Cache	16K bytes	32K bytes	32K bytes
Data SRAM			32K bytes
Scratchpad	4K bytes	4K bytes	4K bytes

Figura 4.2 – *Comparativa Blackfin*

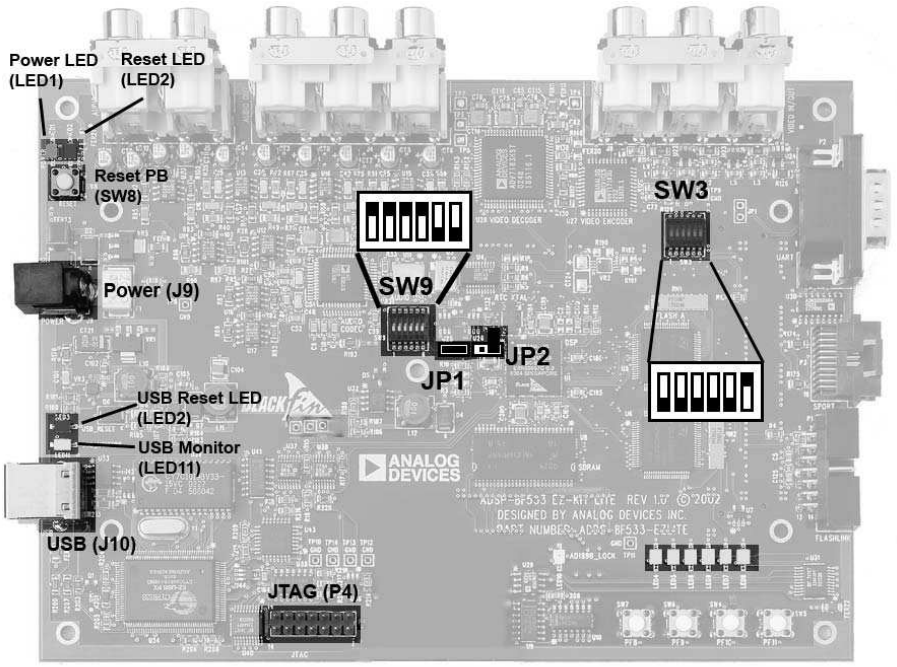

Figura 4.3 – *Bf533 EZ-Lite con evidenziati usb,alimentazione e switch di configurazione*

4.2.1 Il core

Come mostrato in figura (fig 4.4), il core del processore Blackfin, contiene 2 moltiplicatori a 16-bit, 2 accumulatori a 40-bit, 2 ALU a 40-bit, e uno shifter a 40-bit. Le unità computazionali possono gestire dati a 8,16,32 bit da il register file. Il register file contiene 8 registri a 32-bit. Quando si effettuano operazioni su operandi a 16-bit,il register file funziona come se avesse 16 registri indipendenti a 16-bit. Tutti gli operandi per le operazioni di calcolo provengono dal register file multiports e da i campi costanti delle istruzioni. Ogni MAC esegue una moltiplicazione 16-bit x 16-bit in ogni ciclo, accumulando i risultati negli accumulatori a 40-bit.Le ALU consentono di eseguire un insieme tradizionale di operazioni aritmetiche e logiche su dati a 16,32 bit. In più, ci sono alcune instruzioni speciali fatte per accelerare diverse operazioni di elaborazione di signali.Queste includono operazioni su bit,come estrazione e conto di popolazione,moltiplicazione modulo 2^{32},primitive di divisione,saturazione e arrotondamento,esplcitazione segno/e-

sponente. Il set delle istruzioni video include allineamento a byte,operazioni di packing, somme a 16 e 8 bit con clipping, media a 8 bit, SAA (Subtract Absolute Accumulate) a 8 bit.Sono anche incluse le compare/select e istruzioni di ricerca su vettore.

Per alcune istruzioni,2 operazioni su operandi a 16 bit possono essere eseguite contemporaneamente su coppie di registri. Usando poi anche la seconda ALU, quattro operazioni a 16 bit sono possibili. Lo shifter a 40 bit può fare shifts e rotazioni ed è usato per eseguire, normalizzazioni, ed istruzioni di estrazioni/deposito operandi.

Il sequenziatore di programma controlla il flusso di esecuzione delle istruzioni e l'allineamento e decodifica delle istruzioni. Nel controllo del flusso di programma, il sequenziatore supporta salti PC relativi,indiretti, condizionali (con predizione statica di salto) e chiamate di funzioni. Viene fornito supporto per l'hardware looping. L'architettura è completamente interlocked: il programmatore non deve preocupparsi mai di gestire manualmente possibli situazioni di stallo della pipeline, dovuti ad hazards della pipeline stessa.

L'unita aritmetica di calcolo degli indirizzi fornisce 2 indirizzi per simultanei doppi fetch dalla memoria. Contiene un file register multiporta che consiste in :4 gruppi di registri di tipo Idice,Modifica,Lunghezza e Base; 8 registri puntatori a 32 bit addizionali (per la manipolazione in stile C degli stack indicizzati).

I processori Blackfin supportano una architettura Harvard combinata con una struttura gerarchica della memoria: al primo livello (L1) abbiamo una memoria istruzioni, due banchi di memoria per i dati e una memoria di scratchpad che memorizza le informazioni di stack e delle variabili locali.

In più sono forniti multipli blocchi di memoria L1 che offrono la possibilità di configurare la memoria alternativamente come cache o SRAM. Il Blackfin emula, con i registri CLB, un'unità di gestione della memoria (MMU) che consente la protezione della memoria per task individuali che potrebbero operare sul core, e protegge i registri di sistema da accessi scorretti. Si ricorda che questa è una emulazione hardware tramite registri e non esiste una realizzazione hardware vera e propria di una MMU: a maggior ragione questo è vero se si considera che in μLinux per renderlo compatibile con il Blackifn, c'é stato un lavoro di revisione proprio per farlo funzionare anche senza MMU che il Blackfin di fatto

non possiede.

L'architettura consente tre modi operativi: modalità utente, modalità super utente, e modalità emulata. La modalità utente ha accesso ristretto, quindi si ha disposizione un ambiente software protetto, mentre la modalità super utente consente un accesso illimitato a tutte le risorse del core.

Il set di istruzioni del Blackfin è stato ottimizzato in funzione dei codici operativi delle istruzioni 16 bit (le più comuni) in modo da garantire la compattezza del codice compilato. Le istruzioni complesse sono codificate in codici operativi da 32 bit, e rappresentamo in tutto e per tutto istruzioni multifunzione. Il Blackfin ha una capacità multi-issue limitata, in cui un istruzione a 32 bit può essere issued insieme a due da 16 bit.

La sintassi dell'assembly del Blackfin usa una sintassi algebrica per facilità di lettura e codifica.L'architettura è stata ottimizzata per l'uso in congiunzione con il compilatore fornito C/C++ per dare implementazioni software veloci ed efficienti.

4.2.2 Architettura di memoria

Nel Blackfin tutto è *memory mapped* in uno spazio di indirizzamento di 4 Gb,usando indirizzi a 32 bit.Tutte le risorse, incluse memoria interna, memoria esterna, registri di controllo I/O, occupano sezioni diverse dello stesso comune spazio di indirizzamento. Le porzioni di memoria di questo spazio di indirizzamento sono organizzate in una struttra gerarchica per consentire un buon compromesso prezzo/prestazioni di alcuni sottosistemi di memoria veloci come cache e SRAM e altri, lenti e off-chip (le porte di comunicazione).

Il sistema di memoria L1 è il sistema primario con le prestazioni più alte che il Blackfin può fornire.Il sistema di memoria off-chip, a cui si accede tramite l'Unità di Interfaccia Esterna (EBIU), consente espansioni con SDRAM, memorie FLASH e SRAM opzionalmente accedendo fino a 132 MB di memoria fisica.

Il controller della memoria DMA consente scambi di dati con ampia banda. Esso è in grado di effettuare trasferimenti a blocchi di codice o dati fra la memoria esterna e quella interna.

Il Blackfin ha tre blocchi di memoria nel chip consentendo una larga banda

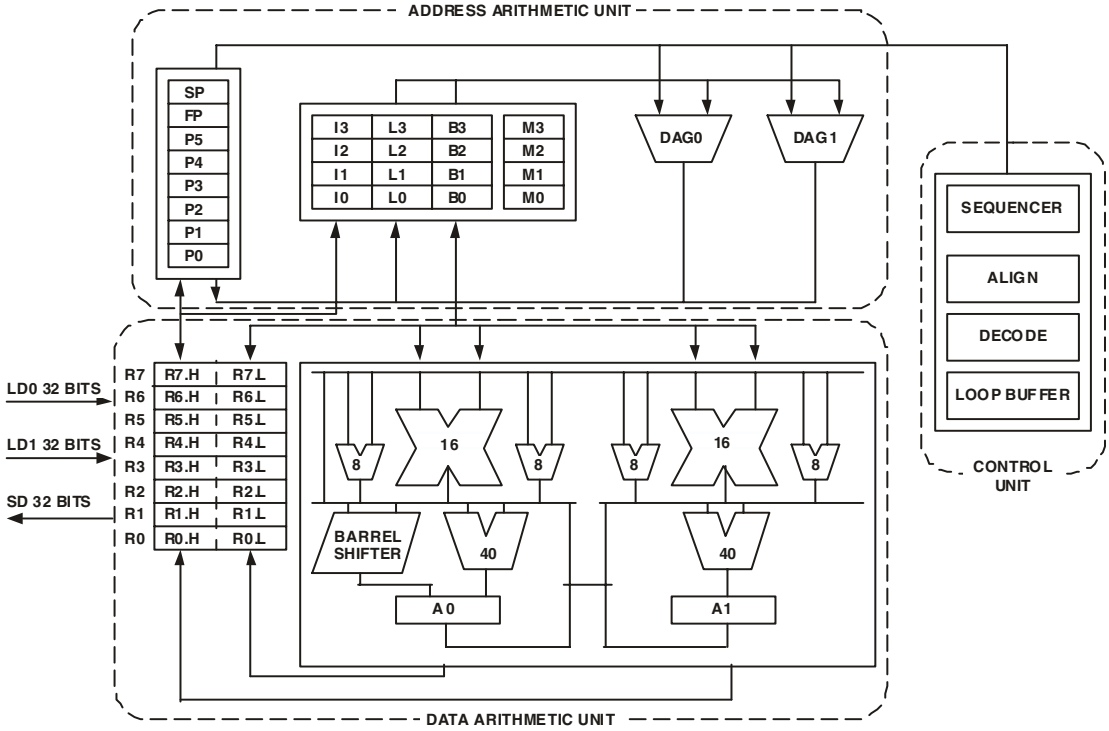

Figura 4.4 – *Core del Blackfin*

di accesso al core. Il primo blocco è la memoria istruzioni L1 , composta da 80 kb di SRAM, di cui 16kb possono essere configurati come una cache a 4 vie set associativa. L'accesso è effettuato alla velocità di clock del core. Il secondo blocco è la memoria dati L1, costituita a sua volta da due banchi ognuno di 32 kb. Ogni banco di memoria è configurabile, offrendo funzionalità sia di SRAM che di cache. Il terzo blocco di memoria è lo Scratchpad di 4kb che va alla stessa velocità della memoria L1, ma è accessibile come SRAM per dati e non può essere riconfigurata.

Interfaccia esterna

L'interfaccia esterna per la memoria può essere usata insieme a periferiche di tipo SRAM,FLASH,EEPROM,ROM,periferiche di I/O e periferiche sincrone come SDRAM. La larghezza del bus è sempre 16 bit, dove soltanto gli 8 bit meno significativi dovrebbero essere usati per i dati. Il controller è in standard PC 133 e può essere programmato per interfacciarsi fino a 128 MB di SDRAM. Il

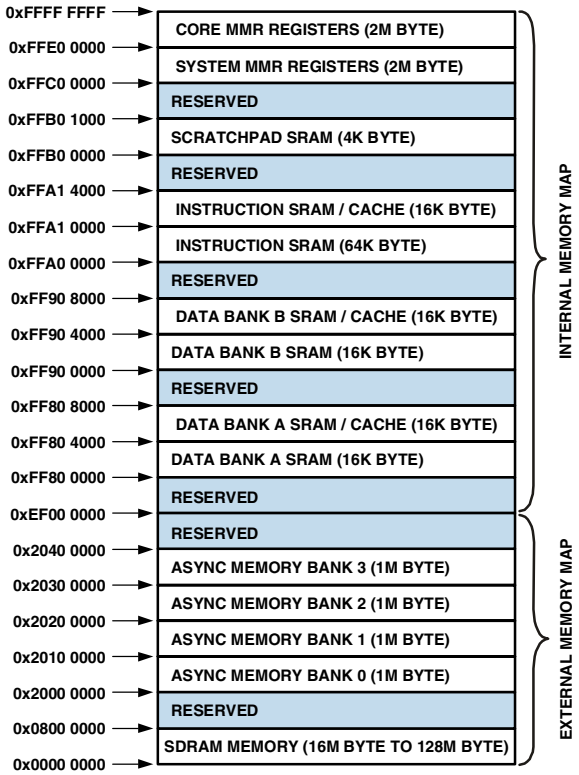

Figura 4.5 – *Mappa di memoria del BF533*

controller della SDRAM permette di attivare una riga per ogni banco di SDRAM memory mapped (fino a 4), aumentando le prestazioni complessive. Il controller della memoria asincrono può essere prgrammato per controllare fine a 4 banchi di periferiche con parametri di temporizzazione molto flessibili che consentono l'utilizzo di un'ampia gamma di periferiche. Ogni banco occupa un segmento da 1MB indipendentemente dalle capacità di memoria della periferica, questi banchi saranno contigui solo se completamente popolati.

Spazio di memoria I/O

I processori Blackfin non definiscono uno spazio di memoria I/O diverso dallo spazio di memoria principale. I registri associati al controllo dei porti (MMR, memory mapped registers) sono in zone di memoria riservate all'inizio dei 4Gb di spazio di indirizzamento. Questi sono separati in due ulteriori blocchi: il primo contiene i MMR, il secondo contiene i registr usati per gestire le periferiche

off-chip.

Il chip fornisce un'efficace e molto articolata politica di gestione degli eventi (fra cui gli interrupt) con un controller che si occupa della politica di arbitrazione.

4.2.3 Accesso diretto alla memoria (DMA)

IL BF533 ha diversi controller indipendenti che forniscono il trasferimento automatico di dati con minimo sovraccarico per il core del processore. I trasferimenti DMA possono avvenire fra qualsiasi delle periferche o memoria capaci di supportarlo fra cui: SDRAM,memoria interna,SPI,UART,PPI. Ogni periferica che supporta il DMA ha un suo canale dedicato. Si parla di MDMA (Memory DMA) se il trasferimento è fra memoria e memoria e di PDMA (Peripheral DMA) se il trasferimento è fra le periferiche e la memoria. Sia il trasferimento monodimensionale, sia quello bidimensionale sono supportati. L'inizializzazione del canale si può fare tramite dei blocchi descrittori o impostando opportuni registri. Il DMA 2D supporta trasferimenti di blocchi fino a 64K x 64k elementi, e larghezza di step di colonna/riga fino a 32 k elementi. Esempi di tipi di DMA supportati sono:

- Un singolo buffer lineare.
- Un buffer circolare, che si aggiorna automaticamente e che lancia un'interrupt appena è pieno.
- trasferimento mono-bidimensionale tramite liste concatenate di descrittori

esistono anche delle situazioni ibride oltre quelle citate. Oltre al supporto del DMA per le periferiche, ci sono 2 canali DMA per i trasferimenti memoria : interna/esterna , interna/interna ecc.... Ogni sorgente e destinazione forma rispettivamente un flusso, e questi due flussi (flusso sorgente, flusso destinazione) di dati sono gestiti da un sistema a priorità:

- Priorità 8 : Flusso di destinazione in memoria D0
- Priorità 9 : Flusso sorgente memoria S0
- Priorità 10 : Flusso di destinazione in memoria D1
- Priorità 11 : Flusso sorgente memoria S1

Il canale DMA 0 (flussi S0,D0) ha precedenza rispetto al canale DMA 1 (flussi S1,D1) a meno che non venga esplicitamente utilizzato uno schema di round robin

fra i canali. Anche se non ortodosso, è consentito scrivere su un flusso sorgente o leggere un flusso destinazione: in casi tipici il flusso sorgente è in lettura, e la destinazione in scrittura, tuttavia impostando opportuni registri è possibile il viceversa, benché questo non sia tipico. I canali supportano trasferimenti a 8 16 e 32 bit, tuttavia il singolo trasferimento deve essere programmato con un solo tipo di dato. In altre parole l'MDMA (Memory DMA) non effettua nessun packing o unpacking dei dati; ogni lettura si conclude in una scrittura. Ogni coppia di canali condivide una FIFO a 16 bit profonda 8 parole. La logica sul sorgente DMA riempre la FIFO, mentre la logica che gestisce la destinazione la svuota.La profondità della FIFO consente trasferimenti burst dal Bus di Accesso Esterno (EAB) e dal Bus di Accesso DMA (DAB) contemporaneamente, aumentando in modo significativo il throughput nel trasferimento dei blocchi fra memoria interna ed esterna. Due separati blocchi di descrizione sono necessari per consentire l'operatività dei canali. Dato che la logica che gestisce la sorgente e la destinazione condivide la stessa FIFO, i blocchi di descrizione devono essere configurati in modo da avere la stessa quantità di dati da trasferire.

Modalità registro

per far partire un trasferimento MDMA gli MMR della sorgente e destinazione devono essere scritti con i parametri corretti. Il registro di riferimento è chiamato DMA_CONFIG ed è associato al flusso di sorgente e destinazione.Per un corretto funzionamento è necessario configurare prima il DMA_CONFIG di sorgente e poi quello di destinazione. Non appena quest'ultimo è configurato il trasferimento inizia dopo 3 cicli di clock;

Modalità a descrittori

se invece si è in modalità di trasferimento tramite descrittori, questi sono per prima cosa recuperati dalla memoria, in particolare i descrittori del flusso di destinazione sono recupearati per primi. Successivamente, con una latenza di 4 ciclci di clock non appena l'ultima parola del descrittore è stata trasferita, il sorgente MDMA comincia a recuperare i dati dal buffer sorgente. I dati risultanti

sono messi nella FIFO e, dopo altri 2 cicli di clock di latenza, la destinazione MDMA comincia a scrivere i dati nel buffer di memoria di destinazione.

Sincronizzazione

un elemento critico della gestione del DMA è la sincronizzazione del completamento del trasferimento con il resto del codice. Questo può essere fatto al meglio usando gli interrupts, o il polling sul MMR IRQ_STATUS, o la combinazione delle due tecniche. Esitono diversi registri su cui andare in polling per il controllo del trasferimento. Il polling sui registri CURR_ADDR, CURR_DESC_PTR, o CURR_X/Y_COUNT non è consigliabile come metodo per ottenere una precisa sincronizzazione con l'elaborazione dei dati in arrivo dai canali DMA, questo a causa del pipelining delle FIFO del DMA. Per essere più chiari bisogna notare che i registri citati, (che corrispondono a Indirizzo Corrente, Puntatore e Contatore) sono aggiornati dal controller diversi cicli di clock in anticipo rispetto all effettivo completamento delle operazioni. Per esempio, nel caso di un'operazione di scrittura DMA sulla memoria esterna, se il canale (si supponga il canale A) viene inizializzato, questo farà si che la SDRAM effettui un'operazione di apertura di pagina, che impiegherà diversi cicli di clock per essere completata. Il controller del DMA potrebbe, allora, iniziare un'altra operazione DMA sul canale B, che per esempio, non ha latenza, ma che sarà bloccata dall'operazione più lenta sul canale A. Un software che controlla il canale B potrebbe concludere in maniera scorretta sull'effettiva fine del trasferimento di dati: infatti non può concludere con sicurezza se la locazione di memoria puntata da CURR_ADDR del canale B sia stata scritta o meno, basandosi sull'analisi del contenuto del medesimo registro. Il Polling su gli MMR riguardanti Indirizzo Corrente, Puntatore e Contatore non consentono pertanto una precisa sincronizzazione. Si ricorda che la lunghezza delle FIFO per una periferica DMA è di quattro locazioni (4 da 8-16 bit, o 2 da 32 bit) e per una FIFO MDMA è otto locazioni (4 elementi da 32 bit). Il DMA non aggiornerà i suddetti registri se queste FIFO sono riempite con dati incompleti (incluse letture che non sono state concluse). Ancora, la lunghezza complessiva delle pipelines sulla memoria L1 è di circa 6 elementi a 8 o 16 bit (MDMA). La lunghezza complessiva delle pipelines della memoria esterna (EBIU) è di circa 4 elementi (MDMA). Se si sommano al lunghezza delle FIFO

e delle pipelines, si può ottenere una stima del massimo numero di operazioni memoria incomplete in un dato momento. Per esempio, si assuma che una periferica DMA stia trasferendo 100 elementi nella memoria interna e il suo registro CURR_X_COUNT legga un valore di 60 elementi rimanenti; quindi, per lo meno, l'elaborazione dei rimanenti 40 elementi è iniziata. La lunghezza totale di pipeline è minore della somma di 4 (PDMA) più 6 (MDMA), o 10 elementi, per questo cautelativamente si può concludere che solo di 40-10=30 elementi è effettivamente completato il trasferimento. Per una sincronizzazione precisa, il programma deve o attendere un interrupt, o consultare il registro IRQ_STATUS associato al canale da controllare. Quando il controller DMA lancia un interrupt o cambia un bit del registro IRQ_STATUS, è garantito che l'ultima operazione DMA è stata completata con successo ed è visibile al core del DSP. Risulta superfluo dirlo, ma ovviamente in questo lavoro è stata usata quest'ultima tecnica.

Gestione del traffico DMA

La precedenza ai canali DMA normalmente è data in base alla loro priorità. La priorità di un canale è semplicemente il numero di quel canale, dove numeri più bassi indicano alta priorità .Per questo le periferiche con flussi di dati veloci dovrebbero essere associate a bassi numeri di canale (alta priorità) usando il MMR DMAx_PERIPHERAL_MAP. I flussi MDMA sono sempre a minore priorità delle altre periferiche, ma dato che richiedeno dati continuamente, è assicurato che possano usare tutti i time slot lasciati vacanti dalle altre periferiche.

Qui si trova lo schema del registro di configurazione MDMA:

Normalmente, quando più di un flusso MDMA è attivo e pronto, è garantito il trasferimento MDMA solo al flusso MDMA a più alta priorità.Se si desidera che i flussi MDMA condividano la banda disponibile si deve impostare il registro MDMA_ROUND_ROBIN_PERIOD per selezionare quanti trasferimenti a turno i due canali debbano effettuare. Il DMA è gestito tramite due processi di priorizzazione simultanei e completamente indipendenti: la priorizzazione del bus DAB, e la priorizzazione del bus di memoria DCB e DEB. Le periferiche che richiedono il DMA tramite il bus DAB, e le cui FIFO sono pronte al trasferimento, si spartiscono i cicli disponibili sul bus stesso. Analogamente, i canali le cui FIFO richiedono servizi di memoria, si sparitiscono i cicli disponibili per acced-

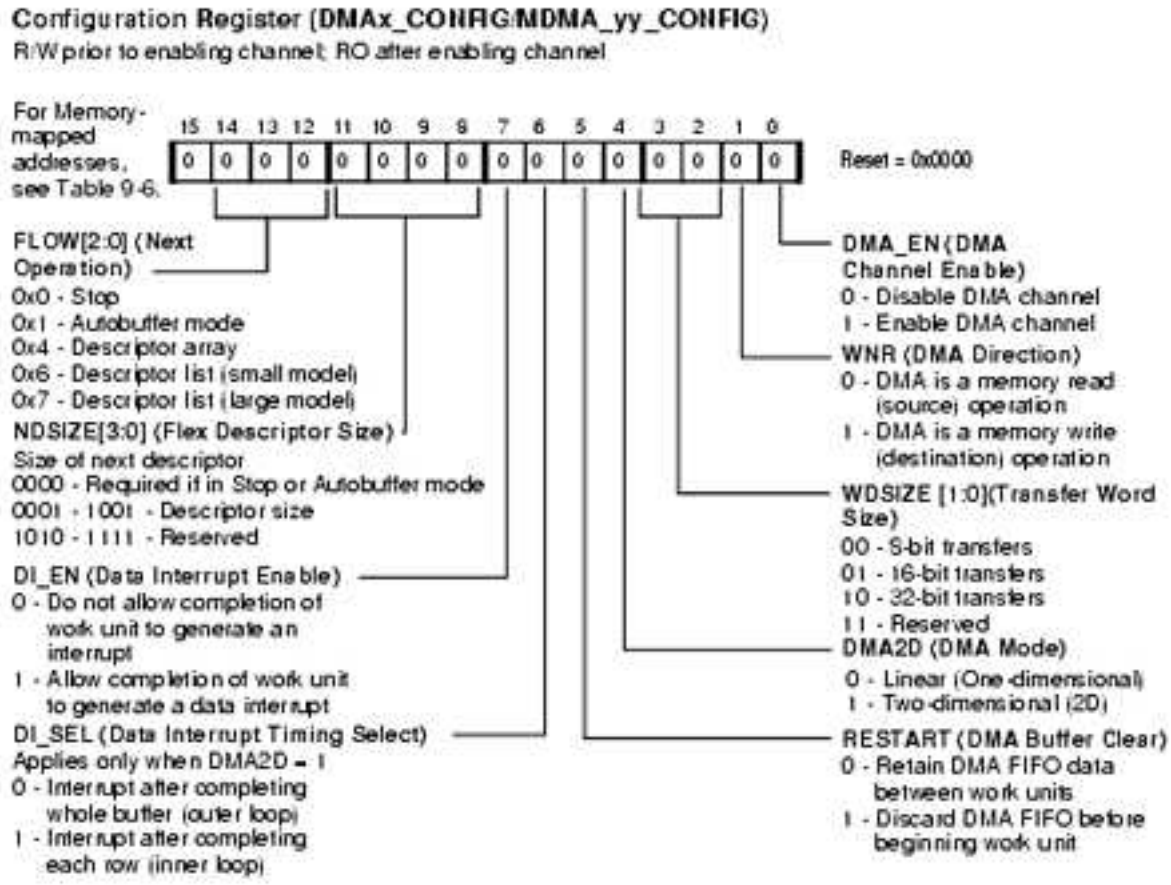

Figura 4.6 – *DMA_CONFIG in tutti i dettagli*

ervi. I flussi MDMA si comportano come un'unità singola e, se i trasferimenti che effettuano non sono in conflitto, si può garantire il trasferimento simultaneo. In tal modo, i trasferimenti fra memoria interna ed esterna e viceversa possono venire alla frequenza di clock di sistema (SCLK) overo la velocità di bus sulla memoria. Il controllo del traffico è un'importante considerazione nell'ottimizzazione delle risorse fornite dal DMA. Il controllo del traffico è un modo per influenzare quanto spesso le direzioni di traffico sui bus possono cambiare, in modo da raggruppare automaticamente i trasferimenti nelle stesse direzioni. Il traffico (anche tramite alcuni registri : DMA_TC_PER e DMA_TC_CNT) può essere controllato indipendentemente per ognuno dei bus (DAB,DCB,DEB) con semplici contatori. Usando le caratteristiche di controllo del traffico, il sistema DMA preferenzialmente garantisce trasferimenti di dati sul bus DAB che van-

no nella stessa direzione di lettura/scrittura del trasferimento precedente, fino a che o contatori di traffico terminano il conteggio, o fino a che il traffico cambi necessariamente direzione. Nel caso del contatore, non appena giunge a zero la direzione di trasferimento è invertita.Queste preferenze direzionali funzionano come se la priorità dei canali della direzione opposta fosse diminuita di 16. Per disabilitare la priorizzazione preferenziale è sufficiente programmare il registro DMA_TC_PER a 0x0000.

Scheduling dei trasferimenti

Tutte le operazioni MDMA hanno minore priorità che qualsiasi altra operazione DMA. Per questo MDMA fa un uso efficace di tutta la larghezza di banda lasciata libera dalle altre periferiche. Se due canali MDMA sono usati (S0-D0 e S1-D1) l'utente potrebbe scegliere di allocare banda con uno schema a priorità fissa o in round robin: questa modalità è selezionabile con il campo MDMA_ROUND_ROBIN_PERIOD nel registro DMA_TC_PER. Se questo campo è posto a 0, MDMA è organizzato secondo una schema a priorità fissa. Il canale 0 ha precedenza sul canale 1. Dato che un flusso MDMA è in grado di trasferire dati in ogni ciclo disponibile, questo potrebbe far si che il canale MDMA venga bloccato finchè il canale 0 non ha eseguito l'intero trasferimento. Questo schema è appropriato in sistemi in cui sono richiesti buffer sensibili alle latenze che bisogna trasferire velocemente. Se il campo MDMA_ROUND_ROBIN_PERIOD è impostato ad un certo valore non nullo $1 < l < 31$, allora lo scheduling round robin è utilizzato e il DMA effettua le operazioni in gruppi di l trasferimenti di un canale e poi di i trasferimenti dell'altro e così via in un classico schema di *time sharing*. Questo schema è appropriato quando dei trasferimenti devono il più possibile coesistere.

4.2.4 Registri

Le unità di elaborazione del processore operano su tre gruppi di registri:

- *Data Register File*: esso consiste di otto registri da 32 bit ciascuno i quali ricevono gli operandi dai bus e memorizzano i risultati delle unità computazionali. Ciascun registro può essere visto come una coppia di registri da 16 bit. Il Data Register File è connesso alla memoria L1 (è interna e opera alla stessa velocità del processore) tramite tre bus: due in lettura ed uno in scrittura, tutti da 32 bit. Sono inoltre presenti due indipendenti registri accumulatori da 40 bit. Ognuno di essi può essere visto come l'insieme di una parte bassa e una parte alta entrambe da 16 bit, più un'estensione da 8 bit.

- *Pointer Register File*: contiene i puntatori per le operazioni di indirizzamento. I general-purpose Address Pointer Register sono tutti da 32 bit e sono:

  - sei registri puntatori
  - una Frame Pointer (FP) usato per puntare il record di attivazione delle procedura corrente
  - uno Stack Pointer Register (SP) usato per puntare l'ultima locazione del Runtime Stack utilizzata.

- DAG(*Data Address Generator*) Register: il DAG genera degli indirizzi per il movimento dei dati da e per la memoria. Generando gli indirizzi DAG permette ai programmi di riferire gli indirizzi stessi in maniera diretta, utilizzando i DAG Register invece che valori assoluti. I DAG Register sono dunque registri dedicati che amministrano dei buffer circolari per le operazioni del processore. Le istruzioni del DSP usano principalmente questi registri per le operazioni di indirizzamento. Essi sono tutti a 32 bit e sono organizzati come:

  - Index Registers(I): contengono l'effettivo indirizzo di un byte in memoria

- Modify Registers(M): il loro contenuto rappresenta l'offset da sommare o sottrarre al contenuto di un Index Register
- Length Registers(B,L): operano l'indirizzamento dei buffer circolari. In particolare, B indica l'indirizzo di partenza del buffer e L la sua lungheza.

4.2.5 Program Sequencer

Il Program Sequencer controlla il flusso di programma ed indica la prossima istruzione da eseguire alle altri parti del processore. Tale flusso di programma è in genere lineare, il processore cioè esegue le varie istruzioni sequenzialmente. Tuttavia essò può variare divenendo non più sequenziale, questo accade quando il processore esegue un'istruzione che non si trova all'indirizzo successivo di quella precedente. Le strutture di programma non sequenziali sono:

- loops - parte del programma che viene ripetuta più volte
- subroutines - sono funzioni, che passati dei valori li elaborano e ritornano un risultato
- jumps - salti all'interno del codice generati da condizioni(if,else,while,/ldots)
- interrupts - interruzione non generata dal codice stesso, ma da periferiche o dal processore.
- exceptions - interruzione generata dal codice stesso
- idle - stato di attesa del processore

E' il Program Sequencer a gestire l'esecuzione di tali strutture indicando l'indirizzo della prossima istruzione da eseguire, esaminando sia l'istruzione che si sta eseguendo in quel momento, sia il corrente stato del processore. Una parte del Program Sequencer, denominata Event Controller, gestisce gli eventi. Gli indirizzi a 32 bit delle istruzioni che in quel momento sono state prelevate, decodificate, ed eseguite vengono inserite in una struttura di tipo pipeline, che è organizzata in dieci stadi.

In particolare l'Event Controller del processore gestisce cinque diverse condizioni operative:

- Emulation
- Reset
- Non-Maskable Interrupts (NMI)
- Exceptions
- Interrupts

Ognuna di queste condizioni operative può essere denominata evento. Visto che si possono avere al più undici Interrupts, in definitiva l'Event Controller si trova a gestire quindi in totale quindici eventi.
Si definisce interrupt un evento che cambia il normale flusso di esecuzione delle istruzioni del processore ed è asincrono con il flusso del programma stesso.
Si definisce exception un evento causato via software i cui effetti sono sincroni col flusso di programma. Il sistema di gestione degli eventi è strutturato tramite annidamento e priorità, ciò significa che più di un evento può essere attivo ad ogni istante. Il meccanismo di controllo degli eventi è inoltre a due livelli:

- *System Interrupt Controller (SIC)* - gestisce le interruzioni proveniente dal sistema processore. Il core prevede la possibilità di gestire un certo numero di interruzioni general-purpose, assegnandovi una certa priorità. Il SIC si occupa dunque di mappare le varie interruzioni provenienti dalle periferiche sugli interrupt general-purpose del processore. Tale mapping è programmabile ed inoltre è possibile mascherare nel SIC l'interruzione proveniente da una determinata sorgente.

- *Core Event Controller (CEC)* - gestisce le interruzioni provenienti dal core. Questa unità supporta, oltre ad alcune Interrupt ed Exception dedicate, le nove Interrupt general-purpouse su cui vengono mappate quelle generate dal SIC.

4.2.6 Le porte

Porta PPI

Il processore ha a disposizione una PPI (Parallel Peripheral Interface) che può essere collegata direttamente e parallelamente ai convertitori A/D e D/A,al decoder e encoder video ITU-R601/656 e ad altre periferiche general purpouse. La PPI ha un clock di ingresso dedicato, fino a tre pin per la sincronizzazione dei frame e fino a 16 pin per i dati. Il Clock di ingresso supporta dati paralleli fino ad una frequenza pari alla metà del Clock di sistema. In modalità ITU-R 656 l'interfaccia può rivecere e analizzare un flusso di dati di 8 o 10 bit che vengono inviati dalla memoria o dalla periferica. Sul chip di decodifica della PPI viene eseguito un controllo e una sincronizzazione delle informazioni. Sono supportate tre distinte modalità in ITU-R 656:

- Active video only - La PPI non legge i dati che vengono inviati tra la fine delle righe attive (EAV - End of Active Video) e quelle del frame successivo(SAV - Start of Active Video). In questa modalità l'immagine in ingresso viene filtrata dalla PPI salvandone in memoria solo una parte.

- Vertical blanking only - La PPI trasferisce soltanto le righe non attive, cioè quelle che non contengono l'immagine, ma soltanto informazioni di controllo.

- Entire field - In questa modalità la PPI trasferisce l'intera struttura del frame(righe attive e di controllo).

Per facilitare il trasferimento e permettere un buffer statico per il frame in memoria può essere usato il DMA a 2 canali, così che si salva in memoria un frame base e poi vengono solamente aggiornate le informazioni attive del video ogni volta che si ha un nuovo frame. Per impostare tale periferica vi è il MMR PPI_CONTROL. Questo registro consente di impostare le modalità di trasmissione e ricezione e l'estensione dei dati da trasmettere.

Porta SPI

IL processore dispone di una porta SPI (Serial Peripheral Interface) sincrona, full duplex. Consiste in quattro connessioni (due per i dati, un Clock ed un segnale di device select). La baud rate è completamente programmabile, così come fase e polaritità del Clock. SPI può operare in uno scenario *multi-master*, interfacciandosi (tramite Pin open drain) con più interfacce compatibili:

- altri CPU o microcontrollori
- codec
- convertitori A/D o D/A
- trasmettitori e ricevitori audio digitali
- display LCD
- shift registers
- FPGA con emulatore della SPI

Ognuna delle interfaccie può essere configurata sia come Master che come Slave. Quando viene trasmesso un dato tutte le interfacce ad essa collegate devono essere Slave e anche se tutte posso ricevere i dati simultaneamente, posso mandare i dati al Master solo una alla volta. E' possibile gestire l'interfaccia SPI in DMA. Quest'ultimo può essere configurato o per ricevere i dati o per trasmettere i dati, non è possibile compiere le due operazioni simultaneamente.

UART

UART (Universal Asynchronous Receiver/Transmitter) è una periferica full-duplex, ovvero ha due canali: uno per trasmettere e uno per ricevere. Queste canali possono essere usati simultaneamente. La funzione di questa periferica è quella di comunicare con periferiche esterne alla scheda. La trasmissione o la ricezione avvengono in modo asincrono e devo avere sempre un bit di start e un bit di stop, mentre il bit di parità è opzionale. Questa periferica supporta due modalità:

- Programmed I/O - il processore manda e riceve dati che vengono letti e scritti su appositi registri(I/O mapped UART registers), in questo caso la gestione avviene attraverso interrupt.

- DMA (Direct Memory Access) - in questo caso è il DMA a controllare il flusso di dati che vengono trasmessi e ricevuti. Questo ne riduce il numero e la frequenza con cui vengono inviati gli interrupt per trasferire dati da o su la memoria. LA UART ha due canali DMA dedicati, uno per trasmettere e uno per ricevere.

Si può impostare questa perifeica attraverso i UART Control and Status Registers.

S PORT

Il processore dispone di due porte seriali sincrone identiche tra loro e denominate SPORT0 e SPORT1. Le SPORT hanno un gruppo di Pin (primary data, secondary data, Clock e frame Sync) dedicati alla trasmissione, ed un gruppo di Pin dedicati alla ricezione. Le funzioni di trasmissione e ricezione sono programmabili separatamente. Ogni SPORT è full-duplex, ovvero capace di trasmettere e ricevere dati simultaneamente e configurabile indipendentemente dall'altra tramite un set di registri dedicati. Le SPORT sono accessibili in DMA e sono dotate di due canali allo scopo di aumentare il Throughput, esse infatti si comportano come se fossero due SPORT diverse, ma con segnali di Clock e frame Sync uguali. Questi segnali posso essere generati internamente dal sistema o ricevuti da una sorgente esterna. Le SPORT possono operate con un formato di dati LSB o MSB, dove la lunghezza della parola può variare da 3 a 32 bits. Questo porte vengono impostate dai Transmit Configuration Registers (SPORTx_TCR1, SPORTx_TCR2).

4.2.7 Clocking

Il Clock di ingresso al processore, denominato CLKIN, deve avere frequenza, duty cycle e stabilità tali da permettere una sua accurata moltiplicazione all'interno del DSP tramite una Phase Locked Loop (PLL) on-Chip. Tale PLL è analogica ed è controllata tramite una macchina a stati programmabile. L'utente può programmare la PLL in modo da moltiplicare il CLKIN del fattore desiderato. Il segnale che risulta da questa moltiplicazione è il Voltage Controlled Oscillator (VCO) Clock. L'utente può agire su tale segnale dividendolo per opportuni fattori al fine di generare il Clock per il Core (CCLK) ed il Clock di sistema (SCLK). Il segnale SCLK fornisce il Clock al bus delle periferiche (Peripheral Access Bus, PAB), al bus del DMA (DMA bus, DAB), al bus per le memorie esterne (External Address Bus, EAB) ed alla unità di interfacciamento esterno (External Bus Interface Unit, EBIU). Per ottimizzare le prestazioni e la dissipazione di potenza, il processore permette di cambiare dinamicamente frequenza di Core e di sistema. Per un aggiustamento fine, può essere variata anche la frequenza del Clock agendo sulla PLL. Grazie all'utilizzo congiunto di alcuni divisori programmabili nel circuito di reazione del PLL, e di un blocco configurabile alla sua uscita, si riescono ad ottenere una grande varietà di fattori moltiplicativi.

4.2.8 RTC

L'RTC (Real Time Clock), che si trova sulla scheda, con le impostazioni standard viene prescalato ad una frequenza di 1 Hz. La sua funzione primaria è quella di tenere conto del tempo. Le altre funzioni sono quelle di generare interrupt. Al reset della scheda gli interrupt sono disabilitati, essi possono essere attivati e mascherati dal registro Interrupt Control(RTC_ICTL), mentre il loro stato può essere letto dal registro Interrupt Status (RTC_ISTAT).

4.2.9 EBIU

L'external Bus Unit Interface (EBIU) è un bus che permette di collegare al processore memorie esterne. L'EBIU serve le richieste di accesso alle memorie

esterne provenienti dal Core o dal DMA. Tale bus funziona alla frequenza del SCLK Clock, così come tutte le memorie sincrone collegate al processore.

4.2.10 Gestione dinamica dei consumi

La gestione dinamica dei consumi si occupa delle funzioni di controllo per la regolazione dinamica della tensione di alimentazione del processore, in modo da ridurre la dissipazione di potenza. Lo stesso obiettivo viene inoltre perseguito, regolando il clock delle varie periferiche che si interfacciano con il processore. In aggiunta a ciò, il BF533 è caratterizzato da cinque modi di funzionamento, ciascuno con un differente rapporto tra prestazioni e consumi:

- Full-on operating mode
- Active operating mode
- Hibernate operating mode
- Sleep operating mode
- Deep sleep operating mode

Full-on operating mode

Rappresenta lo stato di default del sistema e consente il raggiungimento delle prestazioni massime. In questa situazione il processore e le periferiche abilitate funzionano alla velocità più elevata possibile, la PLL è abilitata e non può essere bypassata.

Active operating mode

In questo stato la PLL risulta abilitata, ma bypassato. Tale situazione, detta di *moderate power saving*, prevede che CCLK e SCLK viaggino alla frequenza del clock di ingresso CLKIN. È inoltre possibile, non solo bypassare la PLL, ma anche disabilitarla attraverso il registro di controllo della PLL (PLL_CTL). Una volta che la PLL risulta disabilitata, è necessario riattivarla prima di effettuare la transizione verso la modalità Full-on o verso quella Sleep.

Hibernate operatine mode

In questo stato vengono disabilitate le tensioni e i segnali di clock del processore e delle periferiche sincrone, riducendo al massimo i consumi statici. Per fare ciò occorre scrivere b#00 nei bit FREQ del registro VR_CTL, in modo che vengano disabilitati sia CCLK che SCLK e la tensione di alimentazione interna (V_{DDINT}) sia azzerata. Prima che ciò avvenga però, è necessario scrivere tutte le informazioni critiche, contenute nelle memorie e nei registri interni, in un dispositivo di memoria non volatile, in modo da non perdere lo stato del processore. È opportuno precisare che l'azzeramento di V_{DDINT} non influisce sulla tensione di alimentazione esterna V_{DDEXT}, cosicché tutti i pin esterni risultano mantenuti al livello tri-state, consentendo ai dispositivi che vengono connessi al processore di ricevere l'alimentazione necessaria.

Sleep operating mode

In questa modalità viene ridotta la dissipazione dinamica di potenza disabilitando il segnale di clock del processore, mentre la PLL e il clock del sistema continuano a operare normalmente. Se si verifica un evento esterno o una qualche attività da parte del Real Time Clock (RTC), il processore verrà riattivato. In particolare, se il bit di bypass nel registro PLL_CTL è settato a 0 si passerà alla modalità Full-on, viceversa alla modalità Active. Quando il sistema si trova in Sleep mode, inoltre, non è supportato l'accesso tramite DMA alla memoria L1, mentre è consentito verso la memoria esterna.

Deep sleep operating mode

Con la modalità di funzionamento deep sleep vengono ridotti al massimo i consumi dinamici di potenza disabilitando il clock del processore e quello delle periferiche sincrone. Le periferiche asincrone, quali ad esempio il real time clock, continuano a funzionare, anche se non sono in grado di accede alle risorse interne e alla memoria esterna. Per uscire da questa modalità di massimo risparmio energetico deve verificarsi un interrupt di reset oppure un interrupt asincrono generato dall'RTC. Se l'interrupt proviene dall'RTC il sistema passa alla modalità Active, mentre se proviene dal reset si ha la transizione al Full-on mode.

Capitolo 5

Realizzazione e ottimizzazioni

5.1 Porting del codice

5.1.1 Codice originale

Questo lavoro è partito dal codice originario che corrisponde al file in linguaggio *Fortran* chiamato qpLinSemplificata.f90 di cui è autore il Chiar.mo Prof. Ridella. Questo codice contiene una realizzazione di SMO che deriva direttamente da una semplificazione del codice originario di C.J.Lin di *LibSVM*. Il codice originario di Lin è in C/C++ mentre qpLinSemplificata è in *Fortran*: come detto rispetto alla versione di Lin c'è una semplficazione, questa consiste nella rimozione dello shrinking, questa è una tecnica per ridurre in modo significativo il numero di pattern (moltiplicatori) coinvolti nell'ottimizzazione. Il codice è scritto in *Fortran* molto semplice, senza l'utilizzo di particolari costrutti avanzati del linguaggio, quindi il porting non ha incontrato particolari difficoltà. L'unico costrutto che ha costretto ad un minimo di riflessione è stato il GOTO; questo costrutto è particolarmente da evitare in C perché fortemente sconsigliato da K&R e, tipicamente è fonte di confusione e bassa leggibilità. Nel codice originarie esiste la scelta fra kernel di tipo di lineare e kernel RBF. Nella versione portata in C è stato eliminato il supporto al kernel lineare ed è stato tutto impostato in funzione del kernel RBF. Questa scelta è stata fatta per due motivi: il kernel lineare ha limitate capacità di classificazione e inoltre non è possibile fare con questo tutta una serie di ottimizzazioni. Tutte le variabili in questo codice sono dichiarate `real*8` ovvero

reali a precisione estesa (1 byte * 8 = 64 bit). La nomenclatura delle variabili è identica a quella usata da Lin nel codice di *LibSVM*. Il codice ha la classica struttura di un algoritmo che realizza SMO : inizializzazione delle variabili, loop principale, e dentro a questo selezione del *Working Set* e ottimizzazione. Si rimanda al capitolo SMO per ulteriori chiarimenti sulla nomenclatura.

5.1.2 Porting in C

Come già accennato il porting non ha presentato particolari problemi di traduzione perché strutture e costrutti utilizzati nel codice *Fortran* erano del tutto standard e molto semplici: le strutture dati erano semplici matrici, vettori e variabili. I compilatori utilizzati sono stati il compilatore della piattaforma .NET 2003 e VC++ 6.0 di Microsoft per i test su macchina Windows; su macchina Linux invece si è utilizzato il compilatore GCC 3.3 su kernel 2.6 distribuzione Fedora Core III; per il codice *Fortran* è stato utilizzato il compilatore Intel Fortran integrato nell'ambiente .NET 2003. Il codice è stato scritto secondo le specifiche internazionali ANSI C ed è stato strutturato secondo lo schema classico sorgente.c sorgente.h: ovvero codice sorgente nel .c e definizioni di funzioni e definizioni globali per la compilazione condizionale nell'header. I nomi dei file sono SMO_simple.c e SMO_simple.h. Per omogeneità è stata mantenuta per intero la nomenclatura delle variabili. Ove è stato possibile, ogni variabile nota a compile time è stata trasformata in un `define` in modo da avere un overhead nullo per ciò che è noto a priori della compilazione. Il tipo di dato utilizzato è stato il `float` ovvero il floating-point a 32 bit secondo lo standatd IEEE 754: questa scelta si giustifica con il fatto che in *LibSVM* è stato utilizzato questo tipo di dato. Tutti i cicli di cui è nota l'estensione sono stati tradotti in cicli `for`, inoltre è stato previsto un numero massimo di iterazioni in modo tale da far scomparire strutture iterative di tipo `while` che all'inizio è stato necessario utilizzare per tradurre i GOTO. La normalizzazione è stata leggermente corretta: nel codice originale non viene effettuato alcun controllo per eventuali divisioni per 0, cosa che può accadere se i dati che si utilizzano hanno massimo e minimo coincidente proprio al valore di 0. Con questa modifica si ha la sicurezza che qualsiasi dataset venga correttamente normalizzato (il dataset ionosfera presenta

la peculiarità appena discussa per uno degli attributi). Si è fatto uso intensivo della compilazione condizionale per evitare al massimo l'overhead associato a degli if, anche e sopratutto in vista del porting per il DSP. Si è aggiunta (con la compilazione condizionale) la possibilità di non far fermare l'algoritmo se non entro il massimo numero di iterazioni: questo è stato utile per dare una misura attendibile dei tempi di esecuzione ponendo a $5 * 10^5$ il numero di iterazioni. Si ricorda brevemente che per compilazione condizionale si intendono i blocchi del tipo:

```
#ifdef ESPRESSIONE_COSTANTE
//Parte di codice da abilitare o meno
#endif
```

in cui se è definita la seguente riga di codice `#define ESPRESSIONE_COSTANTE` allora la parte di codice posta fra `#ifdef` e `#endif` è abilitata, diversamente è disattivata. Come già detto non si è realizzato il kernel lineare per i motivi già esposti. Sono state, inoltre, realizzate tutte le funzioni di conversione fra formato IEEE 754 e fixed-point Q15 e Q31 che sono i tipi di rappresentazione numerica usati per i calcoli con il DSP: esistono nell'header inoltre tutte le costanti utili alla conversione di cui si è appena detto. Per quanto rigurda le verifiche di consistenza queste sono state fatte lanciando i programmi con i seguenti dataset: Banana, Pima, Spam, Ionosfera, Heart, Philips. I dataset Pima, Spam, Ionosfera e Heart sono disponibili presso il database UCI su Internet; Banana è un dataset artificiale, Philips è un dataset basato sulla feature extraction di immagini per la valutazione della loro qualità. Questi stessi dataset (ristretti a 100 pattern 50/50 per classe a parte philips che erano 60) saranno utilizzati per i test sul codice del DSP; sono stati scelti questi dataset perché provengono da domini anche molto lontani. Pima, Heart vengono da rilevazioni mediche, Spam proviene viene da analisi testuali su e-mail, Ionosfera proviene da rilevazioni fisiche di nuvole elettroniche nella ionosfera e Banana è artificiale. Con questa scelta si è cercato di coprire domini di analisi anche molto diversi in modo da valutare i risultati il più oggettivamente possibile. In vista della successiva realizzazione su DSP il programma crea anche le hessiane associate al problema e le scrive su file: queste saranno utilizzate nel programma per DSP e poste in un header . Si è fatta questa scelta perché far leggere da file direttamente al DSP i dati, avrebbe comportato

dei tempi improponibili per lavorare degnamente.Purtroppo infatti non c'é un JTag emulato che avrebbe agevolato la realizzazione della tesi. Ad ogni modo il problema è facilmente aggirabile per le finalità di questo lavoro. Nell'effettuare la conversione dell'hessiana da IEEE 754 a Q15 e Q31 è stato calcolato l'errore relativo percentuale massimo commesso su i sigoli valori. Nel caso di conversione a 16 bit e 32 bit l'errore relativo massimo percentuale è stato dello 0.005396 %. Ritornando al codice tradotto, ancora, è stato aggiunto in fase di normalizzazione un ulteriore controllo di consistenza dei dati normalizzati.Anche nella conversione a 16 e 32 bit sono stati inseriti controlli di consistenza per assicurare che i valori siano nel dominio corretto [-1,1]. Per convalidare la correttezza del codice sono state effettuate prove comparative fra il codice originale di qpLinSemplficata.f90 e il programma portato in C con i dataset sopra citati, i seguenti parametri sono stati confrontati per assicurare la correttezza della traduzione:

- Numero e consistenza di True e Bounded Support Vector
- Costo primale,Costo duale,Rho (bias)
- Errore digitale,numero iterazioni,b_{up} e b_{down}

Tutte le prove effettuate hanno mostrato piena consistenza della traduzione del codice mostrando differenze nei risulati, per tutti i dataset e per tutti i parametri, sempre al di sotto dell'1 %. Questa differenza è imputabile direttamente al tipo di dato: nel codice originale il tipo di dato era a 64 bit, in quello C come in*LibSVM* è a 32 bit.

5.1.3 Porting su DSP

Strumenti utilizzati

Il compilatore utilizzato per effettuare il porting è stato quello integrato nell'ambiente di sviluppo VisualDSP++ 3.5.Questo ambiente di sviluppo è dotato di tutti gli strumenti standard tipici, con in più un notevole motore di ottimizzazione interno al compilatore, e la possibilità di effettuare il profiling del codice. I riferimenti tecnici per l'utilizzo del compilatore e delle librerie fornite sono stati, l'Help integrato nell'ambiente e le molte e utili *Engineer to Engineer Notes* messe

a disposizione sul sito di Analog Device: queste note consistono in chiare e dettagliate esposizioni di alcuni argomenti topici della programmazione del Blackfin e dell'utilizzo dell'ambiente. Alternativamente, e in base a possibilità ed esigenze, l'ambiente è stato utilizzato o direttamente con la scheda ADSP-BF533 EZ-Lite, o in modalità di emulazione: in ogni caso ogni dato riportato è sempre riferito a prove fatte direttamente con la scheda.

Rappresentazione numerica

La rappresentazione numerica che è stata utilizzata per questo lavoro è stata la rappresentazione fixed-point. Si è effettuata questa scelta perché sono già largamente noti i problemi realizzativi associati alla rappresentazione floating point, risulta dunque interessante esplorare la possibilità di eseguire un algoritmo di training su un sistema basato su fixed-point. Si ricorda ancora, che il Blackfin non ha supporto hardware per i floating-point il che implica, che utilizzare una rappresentazione floating-point è sempre possibile, al costo però di pagare una realizzazione dei calcoli interamente software. Questa situazione non garantisce in nessun modo prestazioni accettabili. Si ricorda che nello standard ANSI C non è definito con quanti bit debba essere rappresentato un determinato tipo: esistono tuttavia dei vincoli da rispettare quali per esempio `sizeof(char)`$\leq$`sizeof(int)`$\leq$`sizeof(long)`. Nella realizzazione del compilatore .NET per architettura x86 si ricorda che il tipo `float` è stato impostato a 32 bit; lo standard IEEE 754 impone che la rappresentazione del calcolatore sia la seguente:

- 1 bit per il segno;
- 8 bit per l'esponente;
- 23 bit per la mantissa.

Infatti il tipo complessivo si scrive:

$$x = (-1)^s * m * 2^{(e-127)} \tag{5.1}$$

dove s è il bit di segno, e è l'esponente e m la mantissa. La mantissa è sempre definita nell'intervallo [0,1). Così definito questo tipo, fa si che una sua istanza

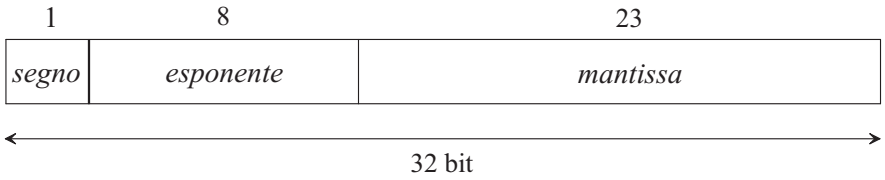

Figura 5.1 – *Rappresentazione IEEE 754 a 32 bit.*

possa assumere valori entro $[10^{-38}, 10^{38}]$ con 7 cifre significative. Generalmente si parla di precisione estesa quando si passa al tipo `double` a 64 bit.Esiste inoltre l'estensione dello standard anche a 128 e 256 bit. Esistono inoltre decodificati alcuni valori particolari come: `Infinity` e `NaN` , questi valori corrispondono ad alcune configurazioni particolari dei bit della variabile in uso, il primo significa valore infinito ed il secondo indica `Not A Number` ovvero un valore che non è stato possibile determinare. Esempio tipico di `NaN` che si verifica è quando si tenta una divisione per 0.

Questa rappresentazione come detto non è stata utilizzata per i motivi citati; ciò che di cui invece si è fatto uso è la rappresentazione fixed-point nella fattispecie quella chiamata Q15 e Q31. Mentre nel caso floating point tutte le operazioni necessarie per garantire la consistenza dei calcoli sono eseguite, o con una libreria, o in hardware, nel nostro caso sarà invece necessario gestire queste operazioni manualmente. In questo modo complicando leggermente la gestione dell'algoritmo si può usufruire di tutto il supporto in termini di intrinsic messo a disposizione dalle librerie del DSP.

Si ricorda ora brevemente la definizione del formato fixed-point. Questo formato non si appoggia a nessuna struttura hardware che non siano normali sommatori e moltiplicatori ad interi. Questo fa si che mentre la realizzazione software si complichi leggermente, l'hardware necessario per supportarlo sia molto semplice. In un fixed-point generico la virgola è implicitamente posta in un punto a scelta dei bit a disposizione, ripartendo fra parte frazionaria e parte intera. Nel caso dei Q15 e Q31 la virgola è posta fra il bit in posizione '1' ed il bit in posizione due '2', come nel floating point il bit '0' è riservato al segno. Così definiti i Q15 e Q31 risultano sempre nel dominio [-1,1). Essendo sempre in modulo minori di 1 i Q15 e i Q31 sono detti frazionari. Il tipo associato in C è lo `short` (16 bit) e `long` (32 bit) per fixed-point con segno, senza segno diventano `unsigned short` (16 bit

) e `unsigned long` (32 bit). Il range associato alla rappresentazione fixed-point

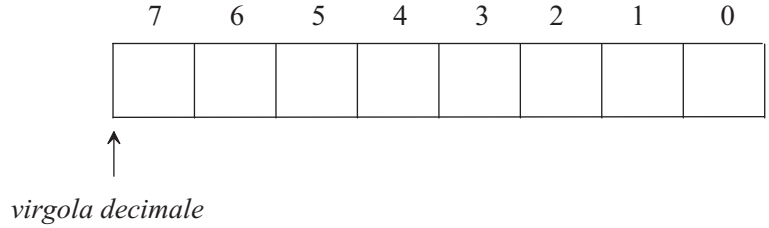

Figura 5.2 – *Esempio fixed-point.*

con segno a n bit è il seguente.

$$\left[-2^{(n-1)}, 2^{(n-1)} - 1\right] \tag{5.2}$$

Che nel caso di fixed-point frazionario diventa:

$$\left[\frac{-2^{(n-1)}}{2^b}, \frac{2^{(n-1)} - 1}{2^b}\right] \tag{5.3}$$

dove b indica il numero di bit della parte frazionaria. Scegliendo b=n-1 ovvero il caso Q(n-1) questo diventa:

$$\left[-1, \frac{2^{(n-1)} - 1}{2^{(n-1)}}\right] \tag{5.4}$$

I prototipi delle funzioni di conversione da Q15,Q31 a floating-point, e viceversa, sono i seguenti:

```
float convertFromFract16(fract16 input,int norma);
fract16 convert_to_fract16(float input,int norma);
float convertFromFract32(fract32 input,int norma);
fract32 convert_to_fract32(float input,int norma);
```

Listing 5.1 – *Prototipi delle primitive di conversione*

Come si vede si passa il numero da convertire e la norma desiderata, si noti poi che `fract16` e `fract32` sono dei `typedef` built-in di `short` e `long`. Le formule di conversione generiche fra floating-point e fixed-point e viceversa realizzate nelle funzioni sono le seguenti:

$$\hat{x} = \left\lfloor \frac{x(2^{(n-1)})}{norma}; \right\rfloor \quad \texttt{e} \quad x = \frac{(\hat{x})norma}{2^{(n-1)}}; \tag{5.5}$$

dove

n è il numero di bit impiegati

$norma$ è un fattore di normalizzazione

x è il numero da approssimare

$\hat{x}$ è il fixed-point frazionario. Ovviamente l'operazione di troncamento è effettuata con un cast.

Struttura e interfaccia

Il codice per DSP è stato struttrato nel seguente modo:

- I sorgenti in C: common.c,dma.c, main.c
- Gli header: hessian.h,SMO_simple.h, e i vari header associati alle hessiane dei problemi considerati.

In main.c risiede il programma principale, in dma.c ci sono le funzionalità associate all'accesso diretto alla memoria, in common.c ci sono alcune utili primitive. Esistono diverse versioni del codice più o meno ottimizzate: la struttura dei files è sempre la stessa, solo nel caso di utilizzo del doppio canale DMA invece del singolo file DMA.c compaiono i file dma_1d_1st_channel.c e dma_1d_2nd_channel.c. Per ogni step del programma esistono due versioni, quella a 32 e quella a 16 bit. I programmi rispetto alla versione per PC hanno delle opzioni in più, atte a modificare i parametri ed il comportamento del programma nel modo più flessibile possibile. Tutte queste opzioni sono state rese secondo il modello della compilazione condizionale in modo da avere un overhead nullo. Le opzioni aggiunte presenti in SMO_simple.h sono le seguenti:

- `NORMA` => consente di impostare il fattore di normalizzazione globale (valore di default = 4)
- `SHIFT_LUT1/2_FACTOR` => rappresenta il numero di shift associati ad un eventuale normalizzazione incrementale delle LUT utilizzate (valore di default = 0)

- `USE_DMA` => abilita,disabilita l'uso del DMA (valore di default = attivo)

- `USE_LUT_SATURATION` => abilita,disabilita l'uso della saturazione delle LUT (valore di default = attivo)

Ogni opzione verrà esposta diffusamente in seguito.

Scelte realizzative

Il codice portato su DSP si struttura in tre parti principali: nella prima ci sono le operazioni di inizializzazione dei parametri e delle LUT ed è off-line, nella seconda invece c'è il ciclo principale di SMO ed è on-line, nella terza in fine ci sono le varie verifiche associate al processo di training ed è ancora off-line.

Nella prima parte come detto hanno luogo le inizializzazioni dei parametri e delle LUT. I parametri vengono inizializzati attraverso l'opportuna primitiva di conversione fixed-point a 16 o 32 bit e con la norma selezionata. A titolo di esempio si consideri l'inizializzazione di alpha[i] il vettore dei moltiplicatori, i bias b[i] e il gradiente g[i].

Ecco una porzione di codice rappresentativa dell'inizializzazione dei parametri.

```
for (i=0;i<NP;i++)
{
        //Inizializzo alpha i moltiplicatori di Lagrange
        alpha[i]=0;

        //Inizializzo i b
        b[i]=convert_to_fract32(-1.0,NORMA);

        //Inizializzo il gradiente
        g[i]=b[i];
}
```

Listing 5.2 – *Esempio di inizializzazione dei parametri*

Per quanto riguarda l'inizializzazione delle LUT ora se ne spiega prima tutto i motivi di utilizzo. L'utilizzo di LUT è stato necessario perché ha eliminato la necessità nella fase on-line di fare una divisione, operazione notoriamente lenta

per qualsiasi rappresentazione numerica eccetto la recente e interessante *LNS* (*Logarithmic Numeric System*). Riprendendo ora le espressioni di aggiornamento degli alpha Eq(3.24) si vede che c'è una divisione. Sotto l'ipotesi di utilizzo di un kernel gaussiano si sa che l'hessiana associata $Q(i,j)$ ha sulla diagonale principale valore costante uguale 1. Questa osservazione permette di semplificare le espressioni in Eq(3.24) in questo modo:

$$\alpha_j^{new} = \begin{cases} \alpha_j + \frac{-G_i - G_j}{2(1+Q_{ij})} \; se \; y_i \neq y_j \\ \alpha_j + \frac{G_i - G_j}{2(1-Q_{ij})} \; se \; y_i = y_j \end{cases} \tag{5.6}$$

Si noti ancora che di volta in volta gli indici i e j che compaiono nella matrice Q sono quelli associati ai moltiplicatori del *Working Set* selezionati dall'euristica per l'ottimizzazione. Quindi, per quanto non sappia a priori quali siano i moltiplicatori da ottimizzare, possiamo comunque creare due LUT in questo modo:

$$\begin{cases} lut1_{ij} = \frac{1}{2(1+Q_{ij})} \\ lut2_{ij} = \frac{1}{2(1-Q_{ij})} \end{cases} \tag{5.7}$$

in tal modo creando off-line le LUT si può evitare la divisione facendola diventare una moltiplicazione:

$$\alpha_j^{new} = \begin{cases} \alpha_j + (-G_i - G_j) * lut1_{ij} \; se \; y_i \neq y_j \\ \alpha_j + (G_i - G_j) * lut2_{ij} \; se \; y_i = y_j \end{cases} \tag{5.8}$$

Tutti i calcoli delle LUT sono off-line in fase forward, quindi sono state fatte in floating-point visto che in prospettiva di una possibile integrazione del tipo: fase forward sul PC, ottimizzazione sul DSP, la fase di inizializzazione delle LUT sarebbe a carico del PC. Purtroppo non è stato possibile fare un sistema unico PC-DSP perché la scheda era esterna: con una scheda interna si sarebbe potuto integrare il sistema monoliticamente sfruttando la scheda DSP come una vera e propria scheda accelleratrice dei calcoli (cfr. 3Dfx Voodoo,Voodoo II). Vista l'estensione delle LUT e delle matrice Q esse sono state poste nella memoria SDRAM esterna.La massima occupazione in memoria corrisponde alla versione a 32 bit in cui si ha: la matrice Q di dimensione $np*np*sizeof(long) = 100*100*4 = 4*10^4 byte$; stessa dimensione hanno le LUT1 e LUT2. Quindi l'occupazione complessiva in memoria al massimo è $12*10^4$ byte ovvero circa 100 kb. Questo

è vero sotto l'ipotesi di utlizzo di 100 *pattern* cosa che succede per tutte le successive prove. Per ottenere migliori prestazioni fra le ottimizzazioni effettuate rispetto alla versione base, si troverà l'utilizzo del DMA che consente un più rapido trasporto dei dati dalla SDRAM alla memoria interna L1. Ancora per garantire migliori prestazioni tutte le matrici sono state vettorizzate in modo da ottenere migliori prestazioni sul DMA, e in generale. Infatti si è verificato che l'utilizzo di un DMA 2D garantisce sempre prestazioni inferiori rispetto al DMA monodimensionale.

Fatte allora queste considerazioni si procede con l'algoritmo vero e proprio. Come prima fase di porting si è semplicemente convertito tutto il codice da floating point a fixed point a 16 e 32 bit. Nel fare questo sono state subito utilizzate tutte le intrinsic messe a disposizione dalle librerie. Fra le librerie disponibili se ne trova una di particolare interesse: la così detta libreria ETSI (`libetsi.h`) la quale garantisce che tutti i calcoli fixed-point vengano eseguiti secondo l'omonimo standard. L'ETSI è l'ente europeo dei sistemi di telecomunicazione: usare queste librerie garantisce una piena compatibilità interoperativa fra i dispositivi che sono *ETSI Compliant*. Lo svantaggio di queste librerie è che sono particolarmente lente rispetto alle versioni built-in delle intrinsic, che al contrario, essendo scritte interamente in Assembly garantiscono prestazioni migliori. Dopo un confronto in termini di precisione e velocità si è optato per le intrinsic built-in, ritenendole un compromesso migliore, visto soprattutto che il dispositivo che si progetta non deve interagire con particolari periferiche. Nella versione a 16 bit sono state utilizzate le seguenti primitive di calcolo:

```
fract32 mult_fr1x32(fract16 op1,fract16 op2);
fract16 add_fr1x16(fract16 op1,fract16 op2);
fract16 sub_fr1x16(fract16 op1,fract16 op2);
```

Listing 5.3 – *Intrinsic a 16 bit*

come si nota la prima primitiva fornisce un risultato a 32 bit: si è usato questa intrinsic perchè usare la versione che da sola riconvertiva il risultato da 32 a 16 bit (`fract16 mult_fr1x16(fract16 op1,fract16 op2);`) garantiva tempi di calcolo più alti e precisione inferiore.

Nella versione a 32 bit sono state utilizzate le seguenti primitive di calcolo:

```
fract32 mult_fr1x32x32(fract32 op1,fract32 op2);
fract32 add_fr1x32(fract32 op1,fract32 op2);
fract32 sub_fr1x32(fract32 op1,fract32 op2);
```

Listing 5.4 – *Intrinsic a 32 bit*

Tutte le intrinsic lavorano con aritmetica saturata.

Come già accennato il programma per DSP gode di alcune ulteriori opzioni. Il parametro `NORMA` determina il valore di norma globale applicato per l'esecuzione dell'algoritmo.Si tratta di un valore cruciale che determina la rappresentazione corretta o meno dei valori in gioco. Va sottolineato che scegliere una norma grande significa diminuire la precisione della nostra rappresentazione per consentire tuttavia la corretta rappresentabilità di numeri maggiori
in modulo di 1.

Esempio:
Non si può fare 2.3+2.7 se non previa normalizzazione. Si ricorda che i Q15 o Q31 hanno valori nel dominio [-1,1). Allora ciò che si deve assicurare è che la somma rimanga nel dominio di definizione. Si fa dunque così: $2.3/norma + 2.7/norma$, dove *norma* è un opportuno valore che consenta un risultato corretto. Si ponga attenzione al fatto che non si può normalizzare a 5 infatti il risultato darebbe 1 che non è rappresentabile. Il valore più corretto è 6. In realtà non è strettamente necessario utilizzare 6 al posto di 5, perché la saturazione fa si che venga generato un risultato corretto approssimato, in altri termini: $Somma_{saturata}(2.3/5, 2.7/5) = 0.99999$ che è un risultato approssimato ma sostanzialmente corretto.

Si vedrà in seguito come è possibile dimostrare che una `NORMA=4` consente sempre la convergenza dell'algoritmo sotto opportune ipotesi. Nella versione a 16 bit il valore massimo di norma per non incorrere in risultati molto imprecisi è 64. Nella versione a 32 bit la precisione dei risultati è accettabile fino a al valore di norma 2048. Dato che spesso è richiesta una re-normalizzazione nel codice, la norma è stata posta sempre ad una potenza di 2 in modo da fare le normalizzazioni con degli shift.

Ora invece si analizzano i seguenti fondamentali parametri del codice:
`USE_LUT_SATURATION`
`SAFE_LUT_COEFFICIENT`

`LUT_SATURATION_VALUE = ((NORMA/(SAFE_LUT_COEFFICIENT))`

Il primo abilita o meno la saturazione delle LUT. Il secondo è un coefficiente di riscalo dei valori delle LUT. Il terzo è il valore che viene inserito nelle LUT in caso di saturazione della stessa. La logica di saturazione è la seguente: se il valore computato teorico della lut supera il dominio di rappresentabilità definito da `NORMA/SAFE_LUT_COEFFICIENT`, allora al posto del valore teorico, si pone un valore definito da `LUT_SATURATION_VALUE` ovvero il `NORMA/SAFE_LUT_COEFFICIENT`. In questo modo variando il `SAFE_LUT_COEFFICIENT` si fa in modo di avere maggiore o minore sicurezza che nei calcoli successivi non si esca dal dominio di definizione. Si parla di coefficiente di sicurezza delle lut perché più il coefficiente è grande più scende la probabilità di uscire fuori dal dominio dei fixed point nei calcoli successivi. Nelle prove di cui si parlerà dopo, come parametro standard, si è utilizzato un $C = 1$: in tal modo con la norma = 4 e un coefficiente di sicurezza anch'esso a 4 si ottiene sempre la convergenza dell'algoritmo con tutti i dataset provati. Quest'opzione è disabilitabile se si conosce con certezza il maggiornante di tutti i risultati dei calcoli, informazione che generalmente non è nota. In linea di massima è ragionevole per delle prove con dataset nuovi, utlizzare una norma a 16 e un coefficiente di sicurezza pari 6: se l'algoritmo converge si può provare ad abbassare la norma e successivamente se possibile anche il coefficiente di sicurezza; se non converge è consigliabile lasciare invariata la norma e provare ad aumentare il coefficiente di sicurezza. L'utilizzo del coefficiente di sicurezza e quindi la saturazione delle lut, è stato, ed è un fattore determinante per garantire la convergenza dell'algoritmo, quindi il suo uso è caldamente consigliato.

Le opzioni `SHIFT_LUT1_FACTOR` e `SHIFT_LUT2_FACTOR` sono gli shift incrementali di normalizzazione che si possono effettuare sulle lut. Si ricorda infatti che la norma, e gli errori di quantizzazione che ne derivano, si propagano indistintamente su tutti i calcoli. Queste opzioni fanno si che se per qualche motivo fosse noto il maggiorante delle lut si potrebbe aumentare la precisione globale facendo in modo di dedicare una norma maggiorata alle sole lut, potendo sperabilmente diminuire la norma globale. Non sempre questa possibilità c'é od è nota, dipende dalla conoscenza del problema. Di default sono poste a 0 che concide a supporre di non avere una conoscenza approfondita del problema.

Infine l'ultima opzione `USE_DMA` consente di abilitare o meno l'uso del DMA.

Si è data la possibilità di disabilitare il DMA per due motivi: il primo è la possibilità di effettuare velocemente comparative sulle prestazioni con e senza DMA, il secondo, molto più grezzo, è dovuto al fatto che in modalità emulazione il DMA viene emulato in modo scorretto costringendo l'utente a disabilitarlo.

5.2 Ottimizzazioni

Dopo una prima fase di porting atta solo a garantire il corretto funzionamento del programma, si è proseguito il lavoro con una fase di ottimizzazione. Ogni step di ottimizzazione ha seguito fedelmente le indicazioni fornite dalle varie *Engineer to Engineer notes* messe a disposizione dalla repository su Internet di Analog Device. Le ottimizzazioni eseguite consistono essenzialmente in: utilizzo del DMA a doppio canale, rimozione di if, allocazione congeniata delle variabili in memoria, allineamento in memoria,una nuova euristica (questa verrà analizzata singolarmente ed in modo puntuale nella sezione successiva) e le ottimizzazioni del compilatore.

DMA a doppio canale

L'elemento che maggiormente, insieme alla nuova euristica, ha contribuito all'aumento delle prestazioni è stato certamente l'uso del DMA. Utilizzando un singolo canale le prestazioni rispetto alla versione che accede alla SDRAM senza DMA, aumentano considerevolmente (mediamente 80%); a maggior ragione utilizzando due canali le prestazioni aumentano di un ulteriore 10-20%. Si è fatto uso del DMA in fase di inizializzazione e nella fase on-line. Nella fase di inizializzazione si è utilizzato il DMA per allocare le lut e riallocare l'hessiana associata al problema dopo la normalizzazione. La parte che più interessa è, tuttavia, la parte on-line, in cui avviene l'ottimizzazione e di cui vengono rilevate le prestazioni. In particolare nella fase on-line il DMA si occupa di rilevare le colonne dell'hassiana necessarie all'aggiornamento del gradiente. Quali colonne sia necessario prendere dalla memoria SDRAM è noto nel momento in cui si conclude l'euristica di scelta dei moltiplicatori: infatti non appena sono noti `igmax1_idx` e `igmax2_idx` (gli indici dei moltiplicatori da ottimizzare) il DMA si occupa di trasferire le colonne dell'hessiana a loro associate . Dato poi che il calcolo del gradiente avviene dopo tutta la fase di ottimizzazione questo dà un notevole vantaggio: nel caso di accesso classico, il tempo che ora il processore utilizza per l'ottimizzazione l'avrebbe impiegato in accessi in memoria. Con il DMA si libera il processore da quest'onere in modo tale da consentirgli di effettuare l'ottimizzazione invece di accedere lui direttamente in memoria. Si noti poi che `igmax1_idx` e `igmax2_idx`

sono sempre diversi per definizione quindi non esiste la possibiltà di conflitto di accessi in memoria nel caso di utilizzo di un doppio canale: proprio per questo motivo, come da specifiche, il trasferimento avviene alla velocità di clock di sistema SCLK. Detto questo le funzioni di accesso in memoria tramite DMA sono state così realizzate con due metodi: il primo si occupa di inizializzare il DMA in modalità registro, il secondo controlla la fine del trasferimento. Ecco i due metodi e la loro definizione:

```
void DMA_1D_transfer_1(fract16* dma_src_data,fract16* dma_dst_data,
int size,int offset,int destOffset)
{

//Ingressi (tutti a 16 bit):
// fract16* dma_src_data: puntatore al buffer sorgente
// fract16* dma_dst_data: puntatore al buffer sorgente
// size: numero di elementi da trasferire
// int offset: offset sul puntatore al buffer sorgente
// int destOffset:offset sul puntatore al buffer destinazione

     //Imposta il System Interrupt Controller affinché abiliti
     //l'interrupt del canale 0. Il bit 21 corrisponde proprio
     //al canale 0.
     //al canale 0.

    *pSIC_IMASK = 1 << 21;

   //DMA configurazione del flusso sorgente. MDMA_S0_xxx

    //Questo MMR imposta l'indirizzo di partenza del trasferimento
    //in questo caso l'indirizzo farà riferimento alla SDRAM
    //Come si vede è possibile specificare un offset di partenza
    *pMDMA_S0_START_ADDR = &dma_src_data[offset];

   //Questo MMR imposta il numero di elementi da trasferire
    *pMDMA_S0_X_COUNT    = size;

    //Indica l'estensione in byte del singolo elemento
    //16 bit in questo caso ovvero uno short
    *pMDMA_S0_X_MODIFY   = 2;
```

```
    //Questi parametri sono utili nel caso di trasferimento 2D
    *pMDMA_S0_Y_MODIFY    =  0;
    *pMDMA_S0_Y_COUNT     =  0;

    //Qui si configura la destinazione : deve essere tutto
    //identico al sorgente e differire solo nell'indirizzo
    //di destinazione

    *pMDMA_D0_START_ADDR =  &dma_dst_data[destOffset];
    *pMDMA_D0_X_COUNT     =  size;
    *pMDMA_D0_X_MODIFY    =  2;
    *pMDMA_D0_Y_MODIFY    =  0;
    *pMDMA_D0_Y_COUNT     =  0;

    //Ora si configura il registro di configurazione del
    //flusso sorgente. Si ricorda che va sempre configurato
    //prima il sorgente poi la destinazione

    *pMDMA_S0_CONFIG = 1 << 2;  // Abilita il DMA a 16 bit
    *pMDMA_S0_CONFIG |= 1 << 0;  // Abilita il DMA

    //Serve per sincornizzare il core con le modifiche ai registri
    ssync();

    //Ora si configura il registro di configurazione del
    //flusso destinazione. Si ricorda che va sempre configurato
    //dopo la destinazione e prima la sorgente.

    *pMDMA_D0_CONFIG  = 1 << 7; // Enable interrupts
    *pMDMA_D0_CONFIG |= 1 << 2;

    //Scrivendo il bit 1 si abilita la scrittua
    *pMDMA_D0_CONFIG |= 1 << 1;

    // Non appena su mette a 1 il bit 0 parte il trasferimento
    *pMDMA_D0_CONFIG |= 1 << 0; // Enable DMA
}
```

```
void test_and_transfer_1D_1()
{
        //Inizializzo questa variabile con l'indirizzo di
        //MDMA_D0_IRQ_STATUS. Quando il terzo bit
        // vale 1 (DMA_RUN = 1), significa che ho finito
        //il trasferimento dei dati e sono pronti per essere usati
        unsigned int addr = 0xFFC00E28;

        //creo un puntatore che indirizza la zona di memoria
        //in cui è mappato  il MMR (memory mapped register)
        // corrispondente a MDMA_D0_IRQ_STATUS
        volatile unsigned short int* reg_MDMA_D0_IRQ_STATUS
        = (unsigned short int volatile*)addr;

        //faccio un loop fino a che non ho finito di
        //fare il trasferimento
        while(*reg_MDMA_D0_IRQ_STATUS & 0x08);

}
```

Listing 5.5 – *Gestione del DMA*

Ecco il frammento di codice in cui viene utilizzato il DMA (fase on-line):

```
//Qui ho già trovato i due moltiplicatori da ottimizzare
//i loro indici sono igmax1_idx e igmax2_idx.

//Se il DMA è abilitato prendo le colonne con il DMA
#ifdef USE_DMA
DMA_1D_transfer_1(QfixVector,Q_selected_column_i,NP,igmax1_idx*NP,0);
DMA_1D_transfer_2(QfixVector,Q_selected_column_j,NP,igmax2_idx*NP,0);
#endif

//Altrimenti faccio una copia locale delle variabili
//con un classico accesso in memoria

#ifndef USE_DMA
for(k=0;k<NP;k++)
        {
                Q_selected_column_i[k]=QfixVector[igmax1_idx*NP+k];
```

```
                Q_selected_column_j[k]=QfixVector[igmax2_idx*NP+k];
        }
#endif
```

Listing 5.6 – *Il DMA nella fase on-line*

Come si vede a causa della vettorizzazione delle matrici, l'acceso richiede che gli indici vengano calcolati nel modo giusto. Si tratta tuttavia di una motliplicazione fra interi il cui costo, vista l'architettura del Blackfin, è di 1 ciclo di clock. Si vede inoltre che ci sono chiamate a due metodi diversi `DMA_1D_transfer_1` e `DMA_1D_transfer_2`. Come è facile immaginare si riferiscono al primo e secondo canale DMA. Il listato precedentemente proposto faceva riferimento al primo canale DMA; il metodo associato al secondo canale è identico con le uniche differenze che i registri di configurazione saranno quelli del tipo `NOME_REGISTRO_D/S1`. Il frammento invece associato al controllo dell'arrivo dei dati e il loro utilizzo nell'aggiornamento del gradiente è il seguente:

```
//Si attendono i dati sul canale 2
#ifdef USE_DMA
test_and_transfer_1D_2();
#endif

#pragma no_alias
#pragma vector_for
#pragma loop_count(NP,NP)
__builtin_aligned(g, 4);
__builtin_aligned(Q_selected_column_i, 4);
__builtin_aligned(Q_selected_column_j, 4);
for (k=0;k<NP;k++)
{
        temp1_G[k]=(mult_fr1x32(Q_selected_column_i[k],delta_alpha_i)
        >>(CONV_SHIFT-SHIFT_NORMA));
        temp2_G[k]=(mult_fr1x32(Q_selected_column_j[k],delta_alpha_j)
        >>(CONV_SHIFT-SHIFT_NORMA));
        temp3_G[k]=add_fr1x16(temp1_G[k],temp2_G[k]);
        g[k]=add_fr1x16(temp3_G[k],g_old[k]);
}
```

Listing 5.7 – *Aggiornamento del gradiente*

Dove la funzione test_and_transfer_1D_2() è quella che si occupa di attendere la fine del trasferimento che è effettuato in modalità registro (vedi 4.2.3 paragrafo Modalità Registro). Si vede dal listato appena mostrato che si attendono solo i dati sul canale due. Questo perché tramite le priorità dei flussi si ha la certezza che se finisce il canale 2, il canale 1 (che ha maggiore priorità) ha già terminato.

Le ottimizzazioni del compilatore

Il compilatore mette a disposizione dell'utente un efficace motore di ottimizzazione del codice. Questo motore è molto potente e flessibile, consente infatti finemente di decidere il trade-off fra estensione del codice e ottimizzazioni da effettuare. Nel nostro caso, per quel che riguarda le dimensioni del codice, si è dato un maggiorante imposto dalle condizioni di licenza: con la licenza in possesso non era possibile andare oltre i 20k di codice. In realtà non ci sono stati grossi problemi in quanto rimuovendo alcuni p̈rintf̈(che sono costosi in termini di estensione di codice) si è sempre rispettato questo maggiorante. Ad ogni modo si è sempre impostato il compilatore per avere il massimo delle prestazioni senza cercare di ottimizzare l'estensione del codice, che in ogni caso risulta esigua. Sempre attraverso il compilatore si è abilitato l'in-lining automatico di tutte le funzioni utilizzate. Un fattore fondamentale per ottenere adeguate ottimizzazioni dal compilatore è stato condizionare il codice, ovvero rendero più simile possibile alle strutture che il compilatore è in grado di ottimizzare. Per permettere le migliori ottimizzazioni, come consigliato, si è evitato di effettuare il loop unrolling, trasformazione fatta dal compilatore. I loop, nel possibile, si sono resi il più semplici possibile, in modo da mettere il compilatore nelle migliori condizioni di riconoscere strutture a lui note. Dove è stato possibile, si sono eliminate le deleterie loop carried dependencies, che avrebbero impedito ottimizzazioni aggressive. Nella versione Visual DSP 4.0 esistono inoltre dei costrutti che consentono una migliore branch prediction statica, purtroppo non sono stati utilizzati perché la versione in possesso del laboratorio è la 3.5: l'utilizzo di questi costrutti avrebbe consentito un ulteriore guadagno di velocità. Un fondamentale strumento di ottimizzazione è quella che viene chiamata IPA *Inter Procedural Optimization*. Si tratta di un set di ottimizzazioni aggressive e molto spinte che garantiscono un buon incremento di prestazioni. Analoghe considerazioni valgono sul condizionamento del codice

affinché l'IPA abbia successo. Esistono ancora due strumenti molto interessanti quali, il linear profiling e la Profile Guided Optimization. Anche questi strumenti sono stati utilizzati per ottenere migliori prestazioni e soprattutto per conoscere i colli di bottiglia del programma e lavorare su di essi.

Allocazione e allineamento della variabili

Per quanto concerne l'allineamento delle variabili i datasheet del Blackfin consigliano sempre di allinearle a 4 (32 bit) se le variabili in gioco sono a 16 e 32 bit. Questa indicazione è stata seguita utilizzando l'appropriata `#pragma align 4` prima delle variabili più importanti. L'allocazione delle variabili ha seguito la logica della massima concorrenza ottenibile negli accessi in memoria. Essendo le variabili coinvolte in diversi calcoli non si può garantire che in ogni calcolo ci sia un'accesso simultaneo alle due aree di memoria in banchi diversi che sono state appositamente definite.Per questo motivo il parallelismo degli accessi si è diretto verso il calcolo chiave del gradiente, e compatibilmente con esso poi si è cercato di massimazzare il parallelismo di tutti gli altri calcoli. Le aree di memoria si definiscono tramite il costrutto `section(NomeBuffer)`. Nel nostro caso i due Buffer sono chiamati BufferA e BufferB e sono allocati in distinti banchi di memoria interni del DSP. Per fare questo il linker si serve di un file di configurazione: modificandolo e specificando le aree di memoria, il linker è in grado effettuare in modo corretto le allocazioni richieste. Dato che si è fatto anche uso della memoria SDRAM esterna, si è dato un nome anche al buffer esterno con lo stesso metodo, esso è ExtMemory in cui, come già detto, sono allocate le lut e l'hessiana. Come si può notare dal listato sul calcolo del gradiente sono state usate delle funzioni __builtin per segnalare al compilatore che alcune variabili sono allineate in memoria e consentendogli così migliori ottimizzazioni.

Altre ottimizzazioni

Altre ottimizzazioni sono state effettuate, essendo diverse, in tal sede si dice soltanto che anche l'ottimizzazione di alcuni dettagli ha contribuito ad una miglioria delle prestazioni. Fra queste ricordiamo l'uso di:

- `#pragma vector_for` che segnala al compilatore che nel loop che segue non ci sono carried loop dependencies.

- `#pragma loop_count(NP,NP)` per indicare che il loop che segue itererà per NP volte.

Per la rimozione delgi if e la nuova euristica si rimanda al sezione immediatamente successiva alla presente.

5.3 La nuova euristica

5.3.1 Introduzione

La creazione di una nuova euristica è nata dell'esigenza di ottenere migliori prestazioni al costo di pagare la non completa soddisfazione delle KKT. Se da un punto di vista strettamente matematico (Vapnik), le KKT sono condizioni sufficienti ad una completa e valida soluzione del problema, sovente si verifica, nella realizzazione dei classificatori, che anche non soddisfare pienamente le KKT porti a risultati molto buoni, consentendo di identificare correttamente una buona approssimazione del minimo della funzione obbiettivo ed i Support Vector associati al processo di training. Si può dimostrare (con la nuova euristica) che la capacità di generalizzazione della macchina cambia (dipende dagli iperparametri), garantendo però buoni risultati in termini assoluti e relativi. Soddisfare completamente le KKT porta spesso l'algoritmo a perdere una grossa parte di tempo, e molte iterazioni, in un'attività che in realtà, ai fini della classificazione, risulta utile solo se si necessita di estrema precisione. Benché l'euristica classica sia matematicamente ineccepibile, computazionalmente è fonte di una notevole inefficienza specialmente nei casi in cui C è grande e ci sono molti pattern nel training. I problemi di datamining sono un esempio immediato e tangibile dei vantaggi portati da un incremento significativo delle prestazioni di training; in questi casi infatti si parla di *Knowledge discovery* su milioni di pattern.

Come già spiegato nella sezione relativa a SMO , l'algoritmo consta di due parti principali : la selezione dei moltiplicatori di Lagrange da ottimizzare e l'ottimizzazione vera e propria. Questa selezione consiste in un ciclo 'for' lungo NP in cui vi sono molti 'if' annidati per la migliore scelta dei moltiplicatori di Lagrange da ottimizzare. Qui di seguito si discuterà una possibile euristica alternativa a quella creata da J.Platt (nella fattispecie rielaborata da C.J.Lin in LibSVM) che consente di ottimizzare il processo di training mantenendo la validità dei risultati ottenuti . In alcuni casi si vedrà come i risultati con la nuova e vecchia euristica siano gli stessi: questo fatto tuttavia è comunque positivo in quanto le ultime versioni fatte della nuova euristica mostrano una velocità maggiore sulla singola iterazione rispetto a quella classica. Esistono diverse versioni del codice, in tal

sede verranno effettuati i confronti fra la versione con euristica di J.Platt e la versione a 16 e 32 bit con la nuova euristica ottimizzata.

5.3.2 Genesi dell'euristica

Le esigenze sopra citate hanno consentito di giungere ad una nuova eurisitica: qui di seguito verrà ripercorsa passo passo la genesi della nuova euristica dal codice orginario alla versione definitiva. L'euristica originale di cui si parla è stata elaborata da C.J.Lin che a sua volta l'ha derivata da *SVM light* (Joachims,1998). *SVM light* in generale sceglie gli n peggiori violatori delle KKT, in SMO in realtà ne servono solo 2 , quindi ponendo $n = 2$ nella formulazione di *SVM light* si ottengono le seguenti espressioni:

$$i = \arg\max([-\nabla f(\alpha_i)|y_i = +1, \alpha_i < C)], [\nabla f(\alpha_i)|y_i = -1, \alpha_i > 0]) \quad (5.9)$$

$$j = \arg\min([-\nabla f(\alpha_i)|y_i = -1, \alpha_i < C)], [\nabla f(\alpha_i)|y_i = +1, \alpha_i > 0]) \quad (5.10)$$

che si traducono nel seguente codice originario :

```
//Ciclo principale
for (i=0;i<NP;i++)
{
        if(y[i]==1) //Peggiori violatori della prima classe
        {       //Qui si scelgono in base al
                //gradiente i peggiori violatori delle KKT
                if(alpha[i]<CC)
                {
                        if (-g[i]>gmax1) {gmax1=-g[i]; igmax1_idx=i;}
                }
                if (alpha[i]>0)
                {
                        if (g[i] >gmax2) {gmax2=g[i]; igmax2_idx=i;}
                }
        }
        else //Peggiori violatori dell'altra classe
        {       //Qui si scelgono in base al
                //gradiente i peggiori violatori delle KKT
                if (alpha[i]<CC)
                {
```

```
                        if(-g[i]>gmax2) {gmax2=-g[i]; igmax2_idx=i;}
                }
                if (alpha[i]>0)
                {
                        if (g[i]>gmax1) {gmax1=g[i]; igmax1_idx=i;}
                }
        }
}
```

Listing 5.8 – *Euristica originale*

In prima istanza possiamo portare le condizioni su `y[i]` dalla parte più esterna agli if più annidati senza cambiare semantica. Inoltre fatto questo, è possibile fare un *jamming* (unione) degli if sulle condizioni su alpha.Ecco il risultato

```
for (i=0;i<NP;i++)
{
        if(alpha[i]<CC)
        {
                if (-g[i]>gmax1)
                {
                        if (y[i]==1) {  gmax1=-g[i]; igmax1_idx=i;}
                }
                if (-g[i]>gmax2)
                {
                        if  (y[i]==-1)  { gmax2=-g[i]; igmax2_idx=i;}
                }

        }
        if (alpha[i]>0)
        {
                if (g[i] >gmax2)
                {
                        if  (y[i]==1) { gmax2=g[i]; igmax2_idx=i;}
                }
                if (g[i]>gmax1)
                {
                        if (y[i]==-1) { gmax1=g[i]; igmax1_idx=i;}
                }
```

```
        }
}
```

Listing 5.9 – *Euristica prima modifica*

A questo punto abbiamo un codice meglio condizionato per le successive considerazioni e trasformazioni.Si ricorda che fin qui nessuna variazione semantica è avvenuta, si è solo effettuata un'operazione di condizionamento del codice. Per prima cosa notiamo dal codice così ottenuto che se `alpha[i]<CC`, il coefficiente moltiplicativo del vettore del gradiente è -1 ; d'altra parte si può notare che se `alpha[i]>0` il coefficiente del vettore del gradiente è +1. Questa osservazione suggerisce la possibilità di riconoscere il segno del gradiente guardando le condizioni su alfa: ci sarà utile per arrivare al nostro scopo di una nuova e più compatta euristica, infatti l'obbiettivo adesso è voler fare un *jamming* dei due if principali (cioé quelli riferiti agli alfa) in modo tale da unire i due if in uno unico, con quindi vantaggi in termini di compattezza e velocità del codice. Se si vogliono unire gli if, in un'unica struttura, bisogna fare in modo che come prima, anche adesso si entri nel corpo unico per le medesime condizioni. In termini logici prima le condizioni erano rispettivamente `alpha[i]<CC` e `alpha[i]>0`, adesso dato che è necessario entrare nel corpo unico, *anche se una sola delle condizioni è vera* bisognerà mappare questa nuova condizione con un `(alpha[i]<CC || alpha[i]>0)`. In questo modo essendo CC reale positivo e gli alfa limitati nel dominio `[0,CC]` la coppia di condizioni `(alpha[i]<CC || alpha[i]>0)` risulta sempre verificata e quindi è completamente rimovibile, ricordando però che in qualche modo (come si vedrà in seguito), sarà necessario recuperare le informazioni necessarie sugli `alpha[i]`. Inoltre si pone implicitamente di entrare nell'if a corpo unico anche con i casi estremi `(alpha[i]==CC || alpha[i]==0)`.

Ancora un'importante osservazione che si può fare riguarda il segno della classe con cui si confrontano di volta in volta gli `y[i]` : si può osservare che se il gradiente (indipendentemente dal suo segno) è confrontanto con `gmax1`, allora nel successivo if annidato, `y[i]` è confrontanto con *(+1)*sign* dove *sign* è sempre discorde rispetto al segno del gradiente. Se, al contrario, il gradiente (indipendentemente dal suo segno) è confrontanto con `gmax2` allora nel successivo if annidato, `y[i]` è confrontanto con *(+1)*sign* dove *sign* è sempre concorde rispetto al segno

del gradiente. Queste considerazioni fanno si che i due gruppi di if si possano unire in uno solo in cui :

- tramite un'opportuna maschera si può recuperare l'informazione sul segno di `alpha[i]-CC`.
- tramite quest'ultima operazione si può pervenire al coefficiente di moltiplicazione di `g[i]` (+-1) (si veda dopo) .
- e ancora conoscendo se il confronto del gradiente è effettuato con `gmax1` o `gmax2` si ottiene *sign*.

Fatte queste considerazioni costruiamo la tabella di verità associata al problema confrontando i segni di `alpha[i]-CC` e `alpha[i]`. Infatti abbiamo che se `(alpha[i]-CC)< 0` dobbiamo entrare indipendentemente dal segno di `alpha[i]` e ovviamente vale anche il viceversa ; parimenti se entrambe le condizioni sono vere entriamo, se no non entriamo. Ricordiamo come già detto che una di queste situazioni è impossibile visto il campo di definizione dei moltiplicatori di Lagrange. Segue la seguente tabella.

sign(alpha[i]-CC)	**sign(alpha[i])**	**Entrata**
+1	+1	SI
+1	-1	NO
-1	+1	SI
-1	-1	SI

Tabella 5.1 – *Tabella associata ai moltiplicatori.*

Osservando ora con attenzione il secondo listato e la tabella si vede che il segno del gradiente nel primo e nell'utlimo caso è univicamente determinato. Nel terzo caso si vede che entrambe le condizioni di entrata sono soddisfatte: questo implica che il segno del gradiente a priori non è noto.

La variante

Proprio questo è il punto chiave della nuova euristica : finora infatti si ricorda che non era stata effettuata nessuna modifica semantica cosa che invece accade

ora. Visto, come detto, che in questo caso non è possibile sapere il segno a priori la scelta è stata quella di assegnare del tutto arbitrariamente il segno del gradiente : in tal caso si è scelto di porre il segno negativo infatti con questa scelta la tabella definisce una semplice operazione di OR. Se si fosse scelto di porre il segno positivo si avrebbe avuto una scelta del tutto arbitraria anch'essa, ma che comportava una diversa tabella di verità, che a sua volta avrebbe comportato una diversa funzione logica, ottenendo in un certo senso un'euristica duale. Da questo si ottiene la tabella equivalente qui sotto in cui +1 indica lo zero logico, e -1 l'uno logico.

sign(alpha[i]-CC)	**sign(alpha[i])**	**Segno di g[i]**
+1	+1	+1
-1	+1	-1
-1	-1	-1

Tabella 5.2 – *Tabella di verità di un semplice OR.*

Come ultima considerazione si può vedere che gli `alpha[i]` mediamente sono nella stragrande maggioranza dei casi (quasi sempre) maggiori o uguali a 0 per cui la seconda maschera del listato dà sempre (o quasi) 0x0000, il che significa, riguardando la tabella di verità , che non serve neanche l'operazione di OR è che quindi basta valutare `sign(alpha[i]-CC)`, infatti l' OR si riduce a `(alpha[i]-CC & 0x8000) | 0x0` in cui l'operazione di AND serve a mascherare l'espressione e a trovare il segno.

Da tutte queste considerazioni ecco l'espressione ed il codice finale.

$$i = \arg\max([-\nabla f(\alpha_i)|y_i = +1, \alpha_i < C)], [\nabla f(\alpha_i)|y_i = -1, \alpha_i = C]) \quad (5.11)$$

$$j = \arg\min([-\nabla f(\alpha_i)|y_i = -1, \alpha_i < C)], [\nabla f(\alpha_i|y_i = +1, \alpha_i = C]) \quad (5.12)$$

Qui si vede come oltre al cambiamento di euristica si siano risparmiati 6 if rispetto alla versione originale in cui erano 10.

```
//Trovo il segno
sign_1=(alpha[i]-CC) && 0x8000;
//Se 0=<alpha[i]<CC sign_1 = -1, altrimenti +1
sign_1=(1-(sign_1<<1));
g_new=g[i]*(sign_1);
//Qui si sceglie l'indice "i" ,chiamato "igmax1_idx"
if (g_new>gmax1)
{
        if (y[i]==-sign_1)
        {
                gmax1=g_new; igmax1_idx=i;
        }
}

//Qui si sceglie l'indice "j" ,chiamato "igmax2_idx"
if (g_new>gmax2)
{
        if (y[i]==sign_1)
        {
                gmax2=g_new; igmax2_idx=i;
        }
}
```

Listing 5.10 – *Euristica nuova*

5.3.3 Analisi computazionale

Ora si propone un'analisi degli effetti sulla minimizzazione del costo confrontando l'euristica di Platt e quella nuova. Calcolando ad ogni iterazione il costo è possibile meglio evidenziare il comportamento della minimizzazione per evidenziarne la dinamica. Come già detto spesso l'algoritmo originale, utilizza una gran parte di tempo nel cercare pedissequamente di soddisfare le KKT per il problema. Questo porta l'algoritmo ad iterare molte volte senza però fare grandi progressi in termini della funzione costo. Si ricorda inoltre che sarebbe gradito che l'algoritmo iterasse il minor numero possibile di volte per almeno due ordini di motivi: il primo riguarda il tempo di training, il secondo riguarda il fatto che tipicamente meno iterazioni significa maggiore capacità di generalizzazione. Per meglio spiegare il comportamento si porta come esempio l'andamento del costo primale durante l'esecuzione dell'algoritmo, in comparativa fra l'euristica classica e quella proposta. Ecco per esempio cosa succede con il dataset Sonar, notoriamente ostico. In rosso è evidenziato il comportamento della nuova euristica, in

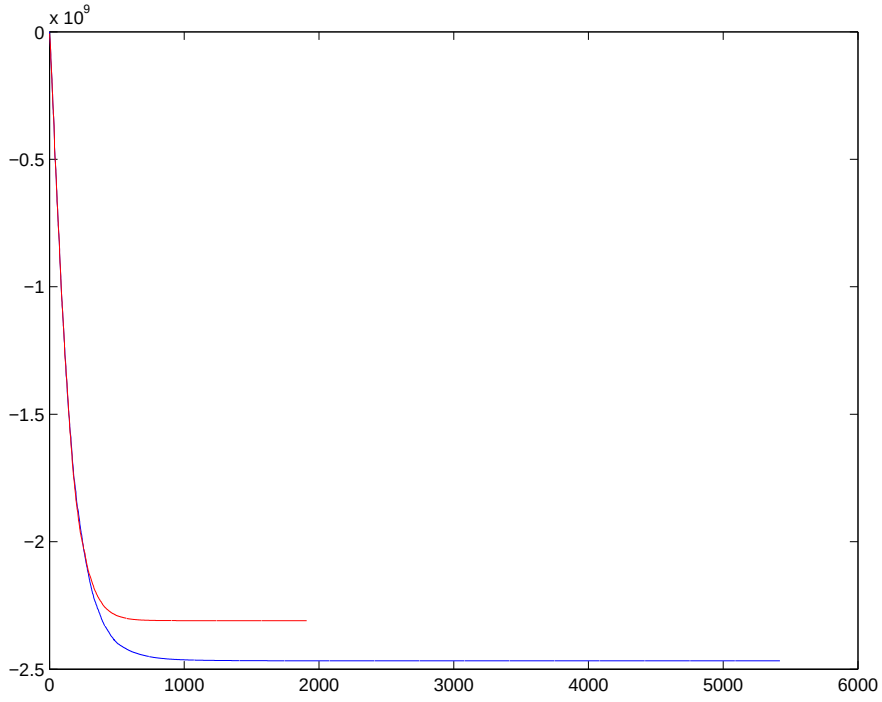

Figura 5.3 – *Sonar comparativa euristiche.*

blu il coportamento di quella classica; in ascissa ci sono le iterazioni, in ordinata

il costo. Come si può notare visivamente l'euristica classica prima di terminare l'algoritmo continua ad iterare senza fare particolari progressi in termini di minimo. L'euristica nuova paga un prezzo in termini del minimo (trova il 93% del valore trovato dall'euristica classica) che è intorno al 7% tuttavia consente un raddoppio delle prestazione. I parametri usati sono stati $C = 2000$ e $2\sigma^2 = 100$ ed il tutto è stato eseguito su PC, i pattern erano 208. Si ricorda che, in generale, associare a priori un minimo più basso a migliori capacità di generalizzazione è sbagliato, solo la model selection può dare una risposta corretta in tal senso. La model selection è tuttavia un procedimento lungo e dispendioso in termini di tempo e di risorse di calcolo, per cui per ora si rimanda questo approfondimento ad ulteriori sviluppi futuri. Ad ulteriore conferma del precedente grafico se ne mostra un altro: questo riguarda il dataset banana che consta di ben 4340 quindi un numero ben maggiore di Sonar. In questo caso lo speed-up è ancora maggiore e arriva circa 10. In tal caso la differenza sul minimo sale al 10%. Quindi sembra

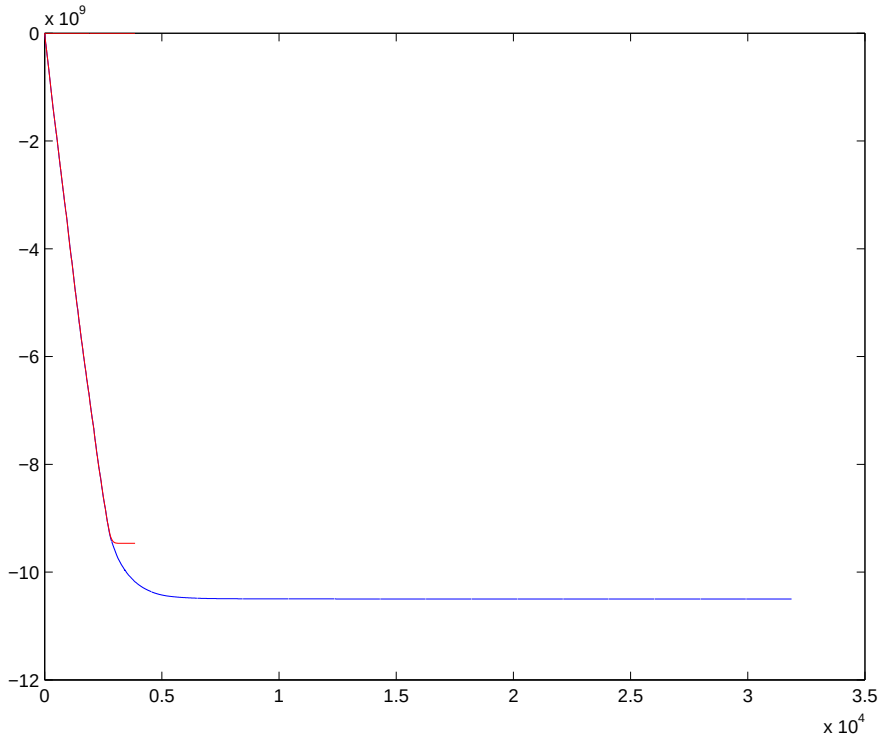

Figura 5.4 – *Banana comparativa euristiche.*

che se aumentano i pattern aumenta lo speed-up ed il gap sul minimo; con la

differenza che in questo caso mentre i pattern sono aumentati di un ordine di grandezza (208 => 4340) la perdita sul minimo passa dal 7% al 10%. Si vede dunque con queste comparative che aumentando il numero di pattern aumenta il gap per il minimo con una velocità che possiamo stimare con il 3% del minimo per ordine di grandezza di pattern, mentre lo speed-up subisce un aumento che è direttamente proporzionale al numero di pattern. Sarebbe molto interessante fare ulteriori prove con milioni di pattern, ma anche in questo caso si rimanda a sviluppi futuri. Il passo successivo da fare, di cui si rimandano al futuro i rusultati, è condurre ulteriori analisi computazionali e approfondite
model selection: queste ulteriori direttrici di indagine daranno ulteriori informazioni utili per capire, quale sia il migliore utilizzo, e con quali eventuali modifiche la nuova euristica possa costituire una valida alternativa all'euristica classica.

Capitolo 6

Risultati sperimentali e sviluppi

In questa sezione verranno sinteticamente presentati i risultati ottenuti a 16 e 32 bit. La loro qualità verrà valutata in termini di consistenza dei risultati del DSP con il PC e di autoconsistenza dei risultati (costo primale, costo duale). Quando in tal sede si farà riferimento a codice non ottimizzato si intenderà il primo porting del codice in cui si utilizzano le intrinsic e l'aritmetica fixed-point: si tenga presente che in questa versione sono `già abilitate al massimo` le ottimizzazioni del compilatore. Quando si parlerà di versioni ottimizzate le ottimizzazioni a cui ci si riferisce sono ottimizzazioni fatte a mano e non dal compilatore; anche in questo caso le ottimizzazioni del compilatore sono attivate al massimo. Quando si utilizzerà la nuova euristica sarà esplicitamente segnalato e ne verranno discussi i risultati. Per tutte le prove effettuate sono stati posti C e $2\sigma^2$ ai valori rispettivamente di 1 e 100, questo per fornire una base comune, inoltre la tolleranza per il criterio di stop è stata posta a 0.01.Fra i parametri di valutazione si è anche valutato l'errore digitale sul training. Si faccia tuttavia attenzione a non giudicare il valore assoluto dell'errore digitale come parametro assoluto di qualità: in parole povere un errore digitale uguale a 50 non è da valutare negativamente se le prove sul PC danno un medesimo errore. Infatti lo scopo di queste prove è mostrare che ci sia consistenza di risultati: il fatto che si abbia un certo errore è dovuto alla scelta di C e di $2\sigma^2$ che sono stati scelti uguali per tutte le prove. Dunque l'ottica di valutazione dei risultati è semplicemente lo scostamento dei valori del DSP dai valori forniti dal PC. In più si ricordi di valutare la metrica di autoconsistenza fra costo duale e costo primale. Verrà omessa l'analisi

pedissequa dei support vector: essendoci 100 pattern per ogni dataset, questo avrebbe obbligato a confrontare 100 * 6 dataset * 2 versioni = 1200 support vector e si ritiene opportuno non riportare questi confronti; va comunque detto che per molte prove il sottoscritto ha analizzato singolarmente tutti i support vector constatandone la correttezza nel valore assunto. Si ricorda che in un training corretto e autoconsistente il costo primale deve uguagliare il costo duale e che il vincolo lineare deve essere nullo (vedi cap 2 sezione 2.2.1).

6.1 Versione a 16 bit

Per completezza di informazione, viene riportato lo scostamento percentuale fra valori trovati con il DSP e con il PC. Tutte le prove sono state effettuate con valore di norma = 4, con coefficiente di sicurezza delle lut = 4 e senza post normalizzazione delle lut.

6.1.1 Dataset Spam

Questo dataset è sicuramente fra i più utilizzati nel *Machine Learning*. Si trova ed è scaricabile gratuitamente nel database UCI. Proviene dall'analisi di e-mail per costruire un sistema anti-spam. Consta di 100 pattern completi, metà di una classe metà dell'altra; ha 57 attributi tutti numerici. Per primo si analizza un confronto di training fra PC e DSP con l'euristica classica. Come si vede

Parametro	**Dsp**	**PC**	**Scostamento**
Costo primale	-32,593056	-31,911636	-2,13%
Costo duale	-31,978428	-31,911636	+0,21%
True SV	16	17	-6%
Bounded SV	49	49	0%
Vincolo	0,000002	-0,000000	0%
Iterazioni	172	55	+68%
Bias	0,270577	0,252786	-5,6%
Errore digitale	1	1	0%

Tabella 6.1 – *Comparativa Dsp,PC a 16 bit dataset spam.*

gli scostamenti sono contenuti entro un 6% e questo è da considerarsi un buon risultato visto che si è passati da floating-point a fixed-point e da 32 a 16 bit. Come si può notare il sistema fa il triplo delle iterazioni: questo è dovuto alla minore accuratezza dei calcoli, il che provoca un maggiore numero di iterazioni per soddisfare le KKT, cosa che non accade sul PC grazie ai floating-point.

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	Non ottimizzata	Speed-up
1212515	3194066	2,63

Tabella 6.2 – *Comparativa Dsp,spam,16 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	Dsp eu1	Dsp eu2	Scostamento
Costo primale	-32,593056	-32,196284	+1,21%
Costo duale	-31,978428	-31,894266	+1%
True SV	16	17	-6%
Bounded SV	49	46	+6.1%
Vincolo	0,000002	-0,000001	0%
Iterazioni	172	86	2 (speed-up)
Bias	0,270577	0,330727	-22%
Cicli di clock	3194066	598010	5,34 (speed-up)
Errore digitale	1	1	0%

Tabella 6.3 – *Comparativa Dsp,spam,16 bit,euristica.*

Si tenga presente che usare un'altra euristica significa cambiare la soluzione del problema, quindi scostamenti anche notevoli sono giustificati (si ricorda il rilassamento delle KKT voluto con la nuova euristica). Il dato sintetico importante è l'errore digitale che non varia, questo è il dato più significativo ai fini della classificazione. Quindi nuova euristica e ottimizzazioni hanno complessivamente contribuito con uno speed-up di 5,34.

6.1.2 Dataset Banana

Si tratta di un dataset artificiale. Il nome deriva dalla particolare e pittoresca disposizione dei dati a banana nello spazio bidimensionale, infatti la dimensione è 2.

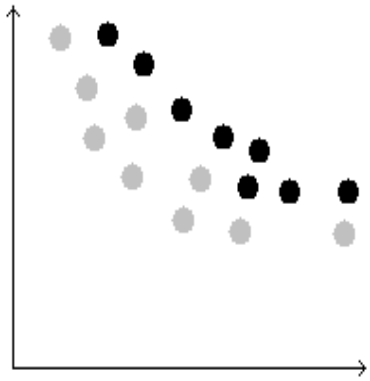

Figura 6.1 – *Dataset Banana*

Anche in questo caso 50 pattern per classe; ecco la prima comparativa

Parametro	**Dsp**	**PC**	**Scostamento**
Costo primale	-97,436363	-96,649155	-0,8%
Costo duale	-96,863669	-96,649155	+0,22%
True SV	0	0	0%
Bounded SV	98	98	0%
Vincolo	0,000000	-0,000000	0%
Iterazioni	97	51	+90%
Bias	0,866787	0,850701	-1,98%
Errore digitale	51	51	0%

Tabella 6.4 – *Comparativa Dsp,PC a 16 bit dataset banana.*

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	Non ottimizzata	Speed-up
653805	1799341	2,75

Tabella 6.5 – *Comparativa Dsp,banana,16 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	Dsp eu1	Dsp eu2	Scostamento
Costo primale	-97,436363	-97,073506	+1,21%
Costo duale	-96,863669	-96,772569	+1%
True SV	0	4	N.D.%
Bounded SV	98	95	-3,06%
Vincolo	0.000000	-0.000000	0%
Iterazioni	97	52	1,87 (speed-up)
Bias	0.866787	-0,866909	-0,1%*
Cicli di clock	1799341	357755	5,03 (speed-up)
Errore digitale	51	49	-3,92%

Tabella 6.6 – *Comparativa Dsp,banana,16 bit,euristica.*

Dove si trova * si voleva segnalare che è stato considerato il modulo; si considera il modulo perché quando ci sono support vector rispetto al caso che non ce ne siano il segno del bias cambia. Si noti l'efficacia della nuova euristica: le iterazioni sul dsp con la nuova euristica diventano in numero uguale (52,51) a quello del pc; questo fatto non è assolutamente casuale, significa che la nuova euristica ha rimosso efficacemente quella situazione di stallo dell'algoritmo che fra l'altro era aggravata dalla disponibilità di soli 16 bit. Anche in questo caso gli scostamenti, rispetto alla versione originale, sono molto piccoli mentre lo speed-up complessivo è di 3.

6.1.3 Dataset Heart

Si tratta di un dataset UCI. Deriva da rilevazioni di parametri su malattie cardiache. N_i è 13. Anche in questo caso 50 pattern per classe; ecco la prima comparativa

Parametro	**Dsp**	**PC**	**Scostamento**
Costo primale	-62,901738	-62,447689	-0,72%
Costo duale	-62,576456	-62,447689	-0,20%
True SV	5	4	-20%
Bounded SV	74	74	0%
Vincolo	0,000002	0,000000	0%
Iterazioni	71	47	-57%
Bias	0,059377	0,061310	-3,25%
Errore digitale	20	20	0%

Tabella 6.7 – *Comparativa Dsp,PC a 16 bit dataset heart.*

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	**Non ottimizzata**	**Speed-up**
485955	1326286	2,73

Tabella 6.8 – *Comparativa Dsp,heart,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**	**Scostamento**
Costo primale	-62.901738	-62.770781	-2,1%
Costo duale	-62.576456	-62.507363	-0.11%
True SV	5	7	+40%
Bounded SV	74	73	-1.35%
Vincolo	0.000002	-0.000000	0%
Iterazioni	97	58	1,67 (speed-up)
Bias	0.059377	0.043999	-25,9%
Cicli di clock	1799341	400985	4,48 (speed-up)
Errore digitale	20	20	0%

Tabella 6.9 – *Comparativa Dsp,heart,16 bit,euristica.*

Anche in questo caso buona precisione, errore digitale costante e speed-up 4,48.

6.1.4 Dataset Philips

Si tratta di un dataset proveniente da immagini fornite dalla philips per la valutazione della qualità delle immagini. In particolare i dati sono il risultato di una feature extraction dalle immagini originali. Questo è l'unico caso in cui il numero di pattern disponibili era inferiore a 100, infatti si tratta di 60 pattern non equamente distribuiti nelle classi di dimesione 12. Ecco la prima comparativa Si noti come, benché il risultato sia consistente con il PC, si paghi una quantità

Parametro	**Dsp**	**PC**	**Scostamento**
Costo primale	-22,106689	-21,947485	-0,72%
Costo duale	-21,958986	-21,947485	-0,20%
True SV	5	8	+60%
Bounded SV	20	19	-5%
Vincolo	0.000000	0.000000	0%
Iterazioni	109	24	-77%
Bias	-1,028645	-1,006795	-2,12%
Errore digitale	11	11	0%

Tabella 6.10 – *Comparativa Dsp,PC a 16 bit dataset philips.*

enorme di iterazioni. Ancora una volta (si vedrà dopo) l'apporto della nuova euristica a rimuovere lo stallo è stato fondamentale.

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	**Non ottimizzata**	**Speed-up**
495485	1254342	2,53

Tabella 6.11 – *Comparativa Dsp,philips,16 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**	**Scostamento**
Costo primale	-22,106689	-21,893711	-0,96%
Costo duale	-21,958986	-21,874894	-0,38%
True SV	5	7	+40%
Bounded SV	20	17	-8,5%
Vincolo	0.000000	-0.000000	0%
Iterazioni	109	16	6,81 (speed-up)
Bias	-1,028645	-1,081646	-5,15%
Cicli di clock	1254342	74420	16,86 (speed-up)
Errore digitale	11	11	0%

Tabella 6.12 – *Comparativa Dsp,philips,16 bit,euristica.*

Anche in questo caso buona precisione, errore digitale costante e un importante speed-up di 16,86 .

6.1.5 Dataset Pima

Si tratta di un dataset proveniente dal database UCI. Consiste in dati estratti da malati di diabete e quindi appartiene ad una classe di problemi di tipo medico: N_i vale 8. Ecco la prima comparativa In questo caso i risultati sono pressoché

Parametro	**Dsp**	**PC**	**Scostamento**
Costo primale	-92,327006	-91,828072	-0,54%
Costo duale	-92,043306	-91,828072	-0,20%
True SV	0	0	0%
Bounded SV	98	98	0%
Vincolo	0,000000	0,000000	0%
Iterazioni	49	49	0%
Bias	-0,520585	-0,510372	-1,96%
Errore digitale	51	50	0%

Tabella 6.13 – *Comparativa Dsp,PC a 16 bit dataset pima.*

identici.

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	**Non ottimizzata**	**Speed-up**
338875	920686	2,72

Tabella 6.14 – *Comparativa Dsp,pima,16 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**	**Scostamento**
Costo primale	-92,327006	-92,327006	0%
Costo duale	-92,043306	-92,043306	0%
True SV	0	0	0%
Bounded SV	98	98	0%
Vincolo	0,000000	0,000000	0%
Iterazioni	49	49	0%
Bias	-0,520585	-0,520585	0%
Cicli di clock	1254342	334770	3,74 (speed-up)
Errore digitale	51	51	0%

Tabella 6.15 – *Comparativa Dsp,pima,16 bit,euristica.*

In questo caso particolare utilizzare o meno la nuova euristica ha il solo effetto di dimuire leggermente i cicli di clock necessari all'esecuzione dell'algoritmo, perché come già detto, anche se i risultati sono identici, la nuove euristica sulla singola iterazione è più veloce.

6.1.6 Dataset Ionosfera

Si tratta di un dataset proveniente dal database UCI. Consiste in dati estratti da rilevazioni tramite un radar di nuvole elettroniche nell'atmosfera. $N_i = 34$. Ecco la prima comparativa In questo caso i risultati sono pressoché identici e

Parametro	**Dsp**	**PC**	**Scostamento**
Costo primale	-49,984459	-49,606995	-0,75%
Costo duale	-49,707806	-49,606995	-0,20%
True SV	9	9	0%
Bounded SV	63	62	2%
Vincolo	0,000000	0,000000	0%
Iterazioni	69	47	-32%
Bias	-1,891917	-1,882265	-0,51%
Errore digitale	12	12	0%

Tabella 6.16 – *Comparativa Dsp,PC a 16 bit dataset ionosfera.*

come al solito si pagano più iterazioni.

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	**Non ottimizzata**	**Speed-up**
481570	1301331	2,70

Tabella 6.17 – *Comparativa Dsp,iono,16 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**	**Scostamento**
Costo primale	-49,984459	-49,726453	-0,75%
Costo duale	-49,707806	-49,514206	-0,20%
True SV	9	10	0%
Bounded SV	63	63	0%
Vincolo	0,000000	0,000000	0%
Iterazioni	69	54	-32%
Bias	-1,891917	-1,895224	-0,51%
Cicli di clock	1301331	378150	3,44 (speed-up)
Errore digitale	12	12	0%

Tabella 6.18 – *Comparativa Dsp,iono,16 bit,euristica.*

La nuova euristica di nuovo, ha sensibilmente consentito di avvicinare la prestazione del PC in termini di iterazioni.

6.2 Versione a 32 bit

Lo scostamento percentuale non verrà riportato essendo palesemente trascurabile. Tutte le prove sono state effettuate con valore di norma = 64 con coefficiente di sicurezza delle lut (SLC) = 4 abilitato solo in alcuni casi (vedi 5.1.3), e senza post normalizzazione delle lut.

6.2.1 Dataset Spam

Per primo si analizza un confronto di training fra PC e DSP con l'euristica classica. SLC disabilitato. I risultati sono abbastanza chiari.

Parametro	**Dsp**	**PC**
Costo primale	-31,914663	-31,911636
Costo duale	-31,911628	-31,911636
True SV	17	17
Bounded SV	49	49
Vincolo	0,000000	-0,000000
Iterazioni	55	55
Bias	0,252867	0,252786
Errore digitale	1	1

Tabella 6.19 – *Comparativa Dsp,PC a 32 bit dataset spam.*

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	Non ottimizzata	Speed-up
585389	1566707	2,68

Tabella 6.20 – *Comparativa Dsp,spam,32 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	Dsp eu1	Dsp eu2
Costo primale	-31,914663	-31,833650
Costo duale	-31,911628	-31,831353
True SV	17	17
Bounded SV	49	46
Vincolo	0,000000	-0,000000
Iterazioni	55	42
Bias	0,252867	0,252786
Cicli di clock	1566707	425704
Errore digitale	1	1

Tabella 6.21 – *Comparativa Dsp,spam,32 bit,euristica.*

Con la nuova euristica si riesce con un guizzo di 13 iterazioni a scendere sotto le iterazioni del PC con un risultato analogo in termini di training. Speed-up = 3,68.

6.2.2 Dataset Banana

Ecco la prima comparativa. SLC è disabilitato per queste prove.

Parametro	Dsp	PC
Costo primale	-96,654006	-96,649155
Costo duale	-96,649038	-96,649155
True SV	0	0
Bounded SV	98	98
Vincolo	0,000000	-0,000000
Iterazioni	52	51
Bias	0,850801	0,850701
Errore digitale	51	51

Tabella 6.22 – *Comparativa Dsp,PC a 32 bit dataset banana.*

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	Non ottimizzata	Speed-up
533839	1476392	2,77

Tabella 6.23 – *Comparativa Dsp,banana,32 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**
Costo primale	-96,654006	-96,649413
Costo duale	-96,649038	-96,644494
True SV	0	2
Bounded SV	98	97
Vincolo	0,000000	-0,000000
Iterazioni	52	52
Bias	0,850801	-0,850801
Cicli di clock	1476392	519809
Errore digitale	51	49

Tabella 6.24 – *Comparativa Dsp,banana,32 bit,euristica.*

Anche in questo caso gli scostamenti, rispetto alla versione originale, sono molto piccoli mentre lo speed-up complessivo è di 2,8.

6.2.3 Dataset Heart

Ecco la prima comparativa. SLC abilitato

Parametro	Dsp	PC
Costo primale	-62,451144	-62,447689
Costo duale	-62,447800	-62,447689
True SV	4	4
Bounded SV	74	74
Vincolo	0,000000	0,000000
Iterazioni	47	47
Bias	0,061316	0,061310
Errore digitale	20	20

Tabella 6.25 – *Comparativa Dsp,PC a 32 bit dataset heart.*

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	Non ottimizzata	Speed-up
486789	1341632	2,76

Tabella 6.26 – *Comparativa Dsp,heart,32 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**
Costo primale	-62,451144	-62,422169
Costo duale	-62,447800	-62,419050
True SV	4	6
Bounded SV	74	74
Vincolo	0,000000	0,000000
Iterazioni	47	42
Bias	0,061316	0,061679
Cicli di clock	1341632	423084
Errore digitale	20	20

Tabella 6.27 – *Comparativa Dsp,heart,32 bit,euristica.*

Anche in questa situazione stessi risultati ed euristica nuova più veloce. Lo speed-up è 3,17.

6.2.4 Dataset Philips

Ecco la prima comparativa. SLC disabilitato Questa volta la maggiore risoluzione

Parametro	Dsp	PC
Costo primale	-21,949341	-21,947485
Costo duale	-21,948784	-21,947485
True SV	8	8
Bounded SV	19	19
Vincolo	0,000000	0,000000
Iterazioni	26	24
Bias	-1,005246	-1,006795
Errore digitale	11	11

Tabella 6.28 – *Comparativa Dsp,PC a 32 bit dataset philips.*

di 32 bit non causa un aumento notevole delle iterazioni rispetto al PC come nel caso a 16 bit. Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	Non ottimizzata	Speed-up
172987	429042	2,48

Tabella 6.29 – *Comparativa Dsp,philips,32 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**
Costo primale	-21,949341	-21,936927
Costo duale	-21,948784	-21,936544
True SV	5	12
Bounded SV	20	15
Vincolo	0,000000	-0,000000
Iterazioni	26	18
Bias	-1,005246	-1,031529
Cicli di clock	429042	116002
Errore digitale	11	11

Tabella 6.30 – *Comparativa Dsp,philips,32 bit,euristica.*

Tutto corretto e speed-up = 3,7

6.2.5 Dataset Pima

Ecco la prima comparativa. SLC abilitato. In questo caso i risultati sono

Parametro	Dsp	PC
Costo primale	-91,832681	-91,828072
Costo duale	-91,828050	-91,828072
True SV	0	0
Bounded SV	98	98
Vincolo	0,000000	0,000000
Iterazioni	49	49
Bias	-0,510466	-0,510372
Errore digitale	11	11

Tabella 6.31 – *Comparativa Dsp,PC a 32 bit dataset pima.*

pressoché identici.

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	Non ottimizzata	Speed-up
504119	1397987	2,77

Tabella 6.32 – *Comparativa Dsp,pima,32 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e la versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**
Costo primale	-91,832681	-91,832681
Costo duale	-91,828050	-91,828050
True SV	0	0
Bounded SV	98	98
Vincolo	0,000000	0,000000
Iterazioni	49	49
Bias	-0,510466	-0,510466
Cicli di clock	1397987	490924
Errore digitale	51	50

Tabella 6.33 – *Comparativa Dsp,pima,32 bit,euristica.*

In questo caso particolare utilizzare o meno la nuova euristica ha il solo effetto di dimuire leggermente i cicli di clock necessari all'esecuzione dell'algoritmo, perché come già detto, anche se i risultati sono identici, la nuove euristica sulla singola iterazione è più veloce.Lo speed-up complessivo è 1,83.

6.2.6 Dataset Ionosfera

Ecco la prima comparativa. SLC abilitato. In questo caso i risultati sono

Parametro	Dsp	PC
Costo primale	-49,610219	-49,606995
Costo duale	-49,607225	-49,606995
True SV	9	9
Bounded SV	63	63
Vincolo	0,000002	0,000000
Iterazioni	47	47
Bias	-1,883310	-1,882265
Errore digitale	12	12

Tabella 6.34 – *Comparativa Dsp,PC a 32 bit dataset ionosfera.*

pressoché identici .

Passando alla versione ottimizzata ecco il confronto in termini di cicli di clock.

Ottimizzata	Non ottimizzata	Speed-up
491124	1352767	2,75

Tabella 6.35 – *Comparativa Dsp,iono,32 bit,versioni ottimizzate e non.*

Ora invece si consideri una comparativa fra versione ottimizzata con la nuova euristica (eu2) e versione base (eu1) .

Parametro	**Dsp eu1**	**Dsp eu2**
Costo primale	-49,610219	-49,469066
Costo duale	-49,607225	-49,466369
True SV	9	11
Bounded SV	63	62
Vincolo	0,000002	0,000000
Iterazioni	47	42
Bias	-1,883310	-1,895521
Cicli di clock	1352767	426469
Errore digitale	12	12

Tabella 6.36 – *Comparativa Dsp,iono,32 bit,euristica.*

Di nuovo un'ottima prestazione

6.3 Tempisitca

Si ricorda che lo scopo di questo lavoro non era fare un sistema che avesse eccezionali prestazioni, ma che potesse invece, sotto opprtune ipotesi ottenere buoni tempi di calcolo e soprattutto con buona precisione. In questa ottica i dati sintetici ora presentati vanno interpretati. Si ricorda, ancora, che tutte le prove su DSP sono state effettuate a 270Mhz, tuttavia il chip ha le potenzialità per arrivare a 756Mhz. Le prove dei tempi sul PC sono state effettuate su un PC così configurato: Pentium III 550 Mhz (bus sulla memoria a 100 Mhz), 256 Mb di RAM, sistema operativo Linux Fedora Core III Kernel 2.6, compilatore GCC 3.3, nessun demone (processo), se non quelli di sistema, attivo durante la presa dei tempi. Ciò che qui verrà analizzato sono i tempi su iterazione medi e i tempi di esecuzione complessivi. Per fare un confronto equo con il PC a 550 Mhz si sono presi i tempi equivalenti del DSP: andando a 270Mhz non si è fatto altro che rilevare i tempi e dimezzarli. Le versioni utilizzate dei programmi per DSP sono quelle ottimizzate al massimo e con e senza la nuova euristica. Ecco allora i tempi sulle singole iterazioni (a 16 bit), con `ne` si intende nuova euristica

DSP	**DSP ne**	**PC**
$13.1\mu s$	$12.5\mu s$	$11.7\mu s$

Tabella 6.37 – *Comparativa Dsp-PC,tempi su iterazione 16 bit.*

Ecco poi i tempi sulle singole iterazioni (a 32 bit), con ne si intende nuova euristica

DSP	**DSP ne**	**PC**
$18.8\mu s$	$18.2\mu s$	$11.7\mu s$

Tabella 6.38 – *Comparativa Dsp-PC,tempi su iterazione 32 bit.*

Si vede chiaramente che la prestazione a 16 bit è molto vicina a quella del PC, mentre a 32 bit si paga l'emulazione dell'aritmetica. Va infatti sottolineato che le strutture hardware di calcolo del Blackfin sono tutte a 16 bit (fanno eccezione i registri) quindi per ottenere aritmetica a 32 bit si ricorre all'emulazione, che rallenta notevolmente i tempi di calcolo.

Questa seconda analisi riguarda invece i tempi complessivi per il ciclo principale di SMO per effettuare il training. Questa è la versione a 16 bit.

	DSP	**DSP ne**	**PC**
Spam	642 μs	575 μs	573 μs
Banana	1271 μs	650 μs	596 μs
Heart	930 μs	725 μs	550 μs
Philips	1428 μs	200 μs	281 μs
Pima	642 μs	612 μs	573 μs
Ionosfera	904 μs	675 μs	550 μs

Tabella 6.39 – *Comparativa Dsp-PC,tempi complessivi 16 bit.*

Le prestazioni raggiunte sono, con la nuova euristica, vicine, in un caso migliori a quelle del PC.

Questa è la versione a 32 bit.

	DSP	**DSP ne**	**PC**
Spam	921 μs	837 μs	573 μs
Banana	978 μs	946 μs	596 μs
Heart	883 μs	837 μs	550 μs
Philips	489 μs	325 μs	281 μs
Pima	921 μs	892 μs	573 μs
Ionosfera	884 μs	764 μs	550 μs

Tabella 6.40 – *Comparativa Dsp-PC,tempi complessivi 32 bit.*

Come già fatto notare, qui si paga l'emulazione del calcolo a 32 bit che non è nativo per questa piattaforma.

6.4 Conclusioni

Come si vede da i numerosi dati esposti in dettaglio, i sistemi realizzati sono in grado di effettuare training corretti in tempi che nel caso della versione a 16 bit con la nuova euristica abilitata avvicinano i tempi del PC e in un caso, grazie alla nuova euristica, lo superano. Questo risultato suggerisce dunque che, sotto le ipotesi iniziali relativamente a dimensione del problema e valori di C, è possibile ottenere training corretti in tempi molto buoni su un dispositivo embedded basato su un chip entry-level come il Blackfin, garantendo quindi bassi costi delle singole unità all'utente finale. Si consideri che questo chip è abbastanza prestante per funzionare per esempio su un telefonino di ultima generazione. Su una periferica di questo tipo potrebbe essere utile per costruire, per esempio, un sistema locale anti-spam personalizzato. Questo è solo un semplice esempio di che cosa si potrebbe fare con il training su una periferica che tipicamente è: a basso costo, facile da usare e portatile. Infatti, quello che capita nelle rare implementazioni di SVM su sistemi embedded, è che il processo di training venga effettuto su PC e poi se ne utilizzino i risultati nella fase di test sulla periferica: adesso invece non è più strettamente necessario ricorrere al PC ma è possibile in loco, magari con un palmare o un telefonino, effettuare il training necessario: questo consente flessibilità, semplicità, e velocità nell'utilizzo di questa potente tecnologia. Un altro modo di utilizzare il sistema è quello, in prospettiva, di utilizzare il sistema come un vero e proprio coprocessore per accelerare il processo di training. Per quanto concerne la versione a 32 bit, benché non nativa per la piattaforma vista la rappresentazione numerica, ha dimostrato comunque tempi accetabili in termini assoluti e notevole precisione. Infatti in ulteriori prove è stato dimostrato che quest'ultima versione è in grado di effettuare agilmente training di problemi completi, dove per problemi completi si intende un numero minore uguale a 500 pattern, situazione che spesso si presenta nei database UCI, che sono il riferimento universale, per questo tipo di problemi. Infine ricordiamo il contributo significativo della nuova euristica, che in alcuni casi è stato decisivo.

6.5 Sviluppi

Gli sviluppi di questo lavoro ed in generale della tematica SVM-DSP si ramificano in tante e diverse direzioni.

La direzione realizzativa riguarda, per esempio, la sperimentazione di SMO su altri sistemi DSP. Oppure utilizzare la stessa piattaforma tecnologica però su scheda interna in modo da consentire una completa integrazione con il PC e sfruttarne tutte le possibilità di interazione: questa configurazione vede implicitamente il DSP (qualunque esso sia) come una vera e propria scheda acceleratrice di calcoli. In questo caso per ottenere un sistema prestante bisognerà utlizzare una contropartita degna di tale compito come per esempio il potente TigerSharc di Analog Devieces stessa. Ancora si potrà realizzare un sistema embedded con il Blackfin stesso e con un applicativo completo per la soluzione di qualche problema di training particolare, partendo da questo lavoro.

La direttrice invece che va verso SVM è assai vasta. Lo sviluppo di questo lavoro ha messo in luce diversi aspetti fondamentali di SVM ed ha consentito di scendere in profondità nell'effettivo comportamento della macchina quando impara. Risulta evidente che qui quando si parla di comportamento implicitamente si fa riferimento a prestazioni di classficazione e la dinamica dei numeri. La nuova euristica è nata proprio a fronte di una comprensione approfondita del comportamento di SMO e della già citata dinamica dei numeri: per dinamica dei numeri si intende come essi variano all'interno dell'algoritmo passo passo. Uno degli argomenti maggiormente da sviluppare e approfondire è proprio questa nuova euristica che in questa analisi preliminare ha dimostrato notevoli vantaggi in termini computazionali. Ulteriori e approfondite analisi sono necessarie per convalidare questa nuova euristica in tutti gli aspetti teorici e pratici di SVM, anche se i risultati fin qui ottenuti sono promettenti e incoraggianti. In particolare, viste le prestazioni, la possibilità di risolvere problemi di data mining di milioni di pattern con speed-up pari ad un ordine di grandezza è un notevole guadagno.

Bibliografia

[1] Nello Cristianini, John Shawe–Taylor, '*An introduction to Support Vector Machines*', Cambridge University Press 2000.

[2] Christopher J.C. Burges, '*A tutorial on Support Vector Machines for Pattern Recognition*',In Knowledge Discovery and Data Mining, 2(2), 1998.

[3] Chih–Jen Lin, '*On the convergence of the Decomposition Method for Support Vector Machines*'.

[4] T. Joachims, '*Making large-Scale SVM Learning Practical*'. In Advances in Kernel Methods – Support Vector Learning, B. Schölkopf and C. Burges and A. Smola (ed.), MIT–Press, 1999, software disponibile all'indirizzo `http://svmlight.joachims.org/`.

[5] J. C. Platt, '*Fast training of Support Vector Machines using Sequential Minimal Optimization*', In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, Advances in Kernel Methods – Support Vector Learning, pagg. 185–208, Cambridge, MA, MIT Press, 1999.

[6] Orlando Mocci, '*Sviluppo ed ottimizzazione di algoritmi per Support Vector Machines in sistemi elettronici embedded*'

[7] S. Keerthi, S. Shevade, C. Bhattacharyya e K. Murthy, '*Improvements to Platt's SMO algorithm for SVM classifier design*'. In Neural Computation, vol.13, pagg. 637–649, Marzo 2001.

[8] Chih–Chung Chang, Chih–Jen Lin, '*LIBSVM: a Library for Support Vector Machines*', Novembre 2004, software disponibile all'indirizzo `http://www.csie.ntu.edu.tw/ cjlin/libsvm`.

[9] Hans Ulrich Simon, '*On the Complexity of Working Set Selection*', Maggio 2004.

[10] Changchun Shi, '*Statistical Quantization Effects and Floating-point to Fixed-point Conversion*' . Department of Electrical Engineering and Computer Science University of California, Berkeley.

[11] Engineer To Engineer Note 149, Analog Devices, '*Tuning C Source Code for the Blackfin® Processor Compiler*'

[12] Analog Devices, '*Visual DSP 3.5 Linker and Utilities Manual for 16-Bit Processors Revision 1.0, October 2003*'

[13] Analog Devices, '*ADSP-BF533 EZ-KIT Lite® Evaluation System Manual*'

[14] O. L. Mangasarian, David R. Musicant,'*Active Support Vector Machine Classiffication*'

[15] Davide Anguita, Sandro Ridella, Fabio Rivieccio, Rodolfo Zunino Silvia Amerio, Ignazio Lazzizzera '*Model Selection in Top Quark Tagging with a Support Vector Classifier*'