Relazione progetto - Gruppo 2

"REGRESSIONE POLINOMIALE"

Introduzione

Durante la prima fase di lavoro sul progetto, è stato necessario uno di studio della richiesta, cercando di individuare, logicamente, il funzionamento totale dell'algoritmo di regressione polinomiale, individuando la struttura e la forma da dare a tutti i parametri in discussione. Ovviamente una prima messa in opera del programma è stata effettuata interamente in linguaggio C, e poi ovviamente, ove possibile, sono stati replicate e ottimizzate le varie funzioni in linguaggio Assembly sfruttando il repertorio sse.

Nella fase preliminare abbiamo cercato di individuare la migliore soluzione per ottenere, dati il numero di parametri del problema (variabile "d") e il grado del polinomio richiesto (variabile "degree"), i coefficienti che vengono utilizzati per l'approssimazione del vettore "theta" all'interno dell'algoritmo di regressione. Quindi come ottenere, considerando come esempio d=4 e degree=2, la seguente combinazione:

[1 , x1 , x2 , x3 , x4 , x1*x1 , x1*x2 , x1*x3 , x1*x4 , x2*x2 , x2*x3 , x2*x4 , x3*x3 , x3*x4, x4*x4]

Siamo arrivati alla conclusione che si trattassero di generazione di combinazioni con ripetizione e adoperato un algoritmo che generasse tale insieme di valori, già ordinato e senza duplicati in modo da ottenere i risultati finali già ordinati dalla generazione e senza dover ricorrere ad un meccanismo per riordinare i dati in seguito alla conclusione dell'algoritmo. La struttura dati utilizzata nel nostro algoritmo avrà la seguente forma:

[0]
[1]
[2]
[3]
[4]
[1 1]
[1 2]
[1 3]
[1 4]
[2 2]
[2 3]
[2 4]
[3 3]
[3 4]
[4 4]

Come possiamo osservare avremo una matrice sparsa, a blocchi di "h" elementi, dove h è il numero di elementi di cui quella specifica combinazione sarà composta. Quindi vengono calcolate combinazioni di 1 solo elemento, di 2 elementi……, fino al degree scelto con cui eseguire l'algoritmo. Questo primo metodo di utilità calcola dato un h fissato e il numero di elementi d quanti elementi ci saranno per quella specifica combinazione:

```
int dimNow(int n, int k) {
	int x=0;
	int numeratore=fattoriale(n+k-1);
	int denominatore=fattoriale(k)*fattoriale(n-1);
	x+=numeratore/denominatore;
	return x;
}
```

Come si può osservare è un classico algoritmo per il calcolo di combinazioni con ripetizione che rispecchia la formula matematica, dove a variare di

$$C'_{n,k} = C_{n+k-1,k} = \binom{n+k-1}{k} = \frac{(n+k-1)!}{(n-1)!k!}$$

Mentre con la successiva otteniamo la lunghezza totale, quindi la lunghezza finale del vettore "theta".

```
int dim(int n, int k) {
	int x=0;
	int i;
	for(i=1;i<=k;i++) {
		int numeratore=fattoriale(n+i-1);
		int denominatore=fattoriale(i)*fattoriale(n-1);
		x+=numeratore/denominatore;
	}
	return x;
}
```

Per la generazione delle combinazioni è stato necessario l'utilizzo di un algoritmo ricorsivo in C, che in pochissimi passi ottiene tutte le combinazioni necessarie. La matrice risultato, vettorizzata e salvata in memoria "per righe", viene ottenuta tramite i due seguenti metodi:

```
int* genera_xStar(int d, int degree){
	//riempie l'intera matrice allocata con gli indici delle combinazioni per x*
	int* coordinate=(int*)get_block(sizeof(int), d);
	int lunghezza=dim(degree,d)+1;
	int h;
	for(h=1;h<=d;h++){
		coordinate[h-1]=h; // genera [1, 2, ...., n] che corrispondono a x1, x2, ..., xn
	}
	int pos[degree]; // pari al massimo valore che assumerà  quindi pari a degree
	int* m=(int*)get_block(sizeof(int),stampaLunghezza(degree,d));
	m[0]=0;
	//richiamo x_star per ogni blocco di dimNow elementi da riempire

	for(h=1;h <= degree; h++){
		xi_star(coordinate,pos, 0, h, 0, d,m);
	}
	return m;
}
```

Conversione (32/64)

Il metodo "**genera_xStar**" ottiene quindi la matrice che abbiamo già visto precedentemente sparsa, dove il metodo "**stampaLunghezza**" individua quanti elementi saranno all'interno della matrice essendo essa sparsa.

```
int stampaLunghezza(int deg, int d){
	int lunghezza=1;
	int i,tmp;
	int j=1, dimR=1;
	while(dimR<=deg){
		tmp=dimNow(d,dimR);
		lunghezza+=tmp*dimR;
		dimR++;
	}
	return lunghezza;
}
```

Il metodo "**genera_xStar**" genera prima un array di dimensione "d", che conterrà valori del tipo [1,2,3,4] nel caso in cui d fosse 4 rappresenterebbe gli indici di x1, x2, x3 e x4, su cui lanciare poi la ricorsione per individuare i valori da inserire di volta in volta.

```
int indice=1;
void xi_star(int * x, int * pos, int num_corr, int h, int at, int d, int* mat){
	//popolo matrice
	int i;
	long count = 0;

	if (num_corr == h) { //condizione di uscita della ricorsione
		if (!pos) return;

		for (i = 0; i < h; i++){
			mat[indice+i]=x[pos[i]];
		}
	indice+=h;
	return;
	}

	//at è l'indice che ricorda l'ultimo valore da cui iniziare per la prossima combinazione
	//num_corr indica l'iterazione attuale

	for (i = at; i < d; i++) {
		if (pos) // MECCANISMO PER NON ANDARE IN CORE DUMP
		pos[num_corr] = i;
		xi_star(x,pos, num_corr + 1, h, i, d,mat);
	}
}
```

Il successivo metodo che viene richiamato è "**calcolaValoriXStar**". Questo metodo lavora sulla matrice (vettorizzata) degli indici, calcolata nel metodo precedente, e sulla matrice X che viene letta come parametro. Esso crea una matrice (sempre vettorizzata) con lo stesso numero di righe di X, quindi n, ma dove ogni riga non contiene [x1,x2,x3…], ma conterrà la riga calcolata in precedenza grazie alle combinazioni, quindi sarà [1, x1, x2…, x1*x1, x1*x2…], dove alla singola **xi** andrà a sostituire il valore di quella variabile per la specifica

riga della matrice **X** (dato del problema). Questa sarà la matrice **xast,** che verrà poi utilizzata per gli algoritmi **SDG_Batch** ed **SDG_Adagrad**.

```
float* calcolaValoriXStar(int* indici, float*x, int n, int d, int degree){
	int lenght = dim(d,degree)+1;
	int lunghezza=dimPadding(lenght);
	float* xStarRes=(float*)get_block(sizeof(float),n*lunghezza);
	int i;
	/*
	Abbiamo memorizzato tutte le combinazioni di coefficienti ( x* ) dentro
	la matrice xStarRes. Ora xStarRes[i] "linearizza" la matrice per poi compiere
	una sostituzione per ognuna delle entry del dataset
	Ad esempio se xStar[i]= [1, 1] (coefficente x1*x1), con x1=5 l'osservazione 1 della riga
	i-esima. Calcolo e inserisco nella nuova matrice 5*5
	*/
	 /*Ogni iterazione del for calcola xi* ovvero sostituisce ad x* i valori
	 dell'array x corrente */
	#pragma omp parallel for
	for(i=0;i<n;i+=4){
		//meccanismo di LOOP Unrolling
		calcolaRigaXStar(indici, x, i, degree, d, xStarRes, lunghezza);
		calcolaRigaXStar(indici, x, i+1, degree, d, xStarRes, lunghezza);
		calcolaRigaXStar(indici, x, i+2, degree, d, xStarRes, lunghezza);
		calcolaRigaXStar(indici, x, i+3, degree, d, xStarRes, lunghezza);
	}

	i=i-4;

	//spingo la macchina ad eseguire operazioni contigue in parallelo
	//dato che non ci sono dipendenze tra i valori

	for(;i<n;i++){
		calcolaRigaXStar(indici, x, i, degree, d, xStarRes, lunghezza);
	}
	return xStarRes;
}
```

Come possiamo osservare, il metodo, lavorando solo in lettura sulla matrice degli indici, può essere richiamato a blocchi, sfruttando il principio di ottimizzazione del **LOOP Unrolling**, dato che la scrittura avverrà su righe, e quindi posizioni, differenti, non generando alcun tipo di conflitto, sia che si lanci il metodo normalmente, sia che si sfrutti il paradigma **fopenmp.**

Osserviamo ora il metodo "**calcolaRigaXStar**", che va a riempire i singoli elementi di una riga lavorando sull'indice i di riga, matrice degli indici e matrice dei dati X. Osserveremo che dovendo andare a calcolare prodotti del tipo x1*x3*x7 ad esempio, i dati a cui accedere per calcolare tale prodotto non sono contigui ergo tale procedura non può essere ottimizzata tramite assembly, ne verrà mostrata una piccola dimostrazione, e nella versione finale rimarrà puramente in C con l'utilizzo del meccanismo del **LOOP Unrolling.**

```
void calcolaRigaXStar(int* mat, float* arr, int posizione, int degree, int d, float* xStarRes, int lenght){
    /*
    Utilizzo la matrice degli indici ricevuta per sotuire i valori
    di una specifica osservazione
    Ad esempio data la riga delle osservazioni:              x1              x2
                                                    [     2.000000 2.000000 ]

                        0         x1         x2        x1*x1     x1*x2      x2*x2
    Ottengo la riga: [  1.000000   2.000000   2.000000   4.000000   4.000000   4.000000   ]
    */
    int dimensioneCorrente=1;
    int j=1,indice=1;                                          //Aggiunta indice per vettorizzazione
    int tmp,i;
    float prod;

    int pos=posizione*lenght;
    int pos2=posizione*d-1;
    xStarRes[pos]=1.0;
    for (dimensioneCorrente; dimensioneCorrente<=degree; dimensioneCorrente++) {
    /*Calcola quanti elementi ha la dimensione corrente
    Ovvero quante righe scorrere nella matrice degli indici x*
    */
        tmp=dimNow(d,dimensioneCorrente);
        while(tmp>0){
            prod=1.0;
            /*Il ciclo for cicla gli elementi di una singola riga della matrice degli indici
            Ovvero una singola conbinazioni degli indici*/
            for(i=0;i<dimensioneCorrente;i++){
                prod=prod*arr[pos2+mat[indice+i]];
            }
            xStarRes[pos+j]=prod;
            j++;
            indice+=dimensioneCorrente;
            tmp--;
        }
    }
    while(j<lenght){
        xStarRes[pos+j]=0.0;
        j++;
    }
}
```

Il triplo for serve a navigare la matrice degli indici, che come abbiamo già detto è sparsa, quindi richiede questo meccanismo di iterazione, dove va a calcolare dimensione per dimensione quanti elementi ci saranno in quel blocco.

Se osserviamo il codice vediamo l'esecuzione del metodo **"dimPadding()"**, questo metodo va a calcolare una nuova dimensione multipla di 16, questo perché nel codice assembly si lavora a blocchi di 16 elementi, sia che si lavori con registri XMM quindi 4 float, sia che con registri YMM quindi con 4 double, lavorando a gruppi di 4 elementi per volta.

```
int dimPadding(int dimensioneReale){
    int d=dimensioneReale/16;
    int r=dimensioneReale%16;
    if(r==0) return dimensioneReale;
    else return (d+1)*16;
}
```

Se avessi quindi un vettore di 14 elementi, si farebbe padding a 16, ergo nella costruzione delle righe vado ad aggiungere alla fine gli zeri di padding, in modo da non sforare sulla dimensione su assembly.

Ovviamente i due metodi **genera_xStar** e **generaValoriXStar** hanno la medesima impostazione sia che si lavori a 32 che a 64 bit, essendo l'implementazione solo lato C, ovviamente si lavora con double invece che con float, quindi le allocazioni saranno della dimensione opportuna.

Nelle prime fasi di lavoro i metodi **genera_xStar** e **generaValoriXStar** sono stati ottenuti lavorando direttamente con matrici (non vettorizzate), per una prima comprensione dell'algoritmo, ecco qui le versioni a matrice:

```
int indice=1;
long xi_star(int * x, int * pos, int num_corr, int h, int at, int d, int**mat){
	//popolo matrice allocata con il metodo "alloca" solo per il blocco
	//di dimNow righe (ovvero dimNow elementi)
	int i;
	long count = 0;

	if (num_corr == h) { //condizione di uscita della ricorsione
		if (!pos) return 1;

		for (i = 0; i < h; i++){
			mat[indice][i]=x[pos[i]];
		}
	indice++;
	return 1;
 }

	for (i = at; i < d; i++) {
		if (pos) // MECCANISMO PER NON ANDARE IN CORE DUMP
		pos[num_corr] = i;
		count += xi_star(x,pos, num_corr + 1, h, i, d,mat);
	}
	return count;
}

int** genera_xStar(int d, int degree){
	//riempie l'intera matrice allocata con gli indici delle combinazioni per x*
	int* coordinate=(int*)get_block(sizeof(int), d);
	int h;
	for(h=1;h<=d;h++){
		coordinate[h-1]=h; // genera [1, 2, ...., n] che corrispondono a x1, x2, ..., xn
	}
	int pos[degree]; // pari al massimo valore che assumerà  quindi pari a degree
	int** m=allocaSpazioXStar(degree,d);
	m[0][0]=0;
	//richiamo x_star per ogni blocco di dimNow elementi da riempire
	for(h=1;h <= degree; h++){
		xi_star(coordinate,pos, 0, h, 0, d,m);
	}
	return m;
}
```

Questo il metodo che genera gli indici delle combinazioni che originariamente, solo per test, restituiva anche il numero "count" di combinazioni. Successivamente eliminato.

```
float* calcolaRigaXStar(int** mat, float* arr, int posizione, int degree, int d){
 /*
      Utilizzo la matrice degli indici ricevuta per sotuire i valori
      di una specifica osservazione
      Ad esempio data la riga delle osservazioni:                x1                  x2
                                                          [      2.000000 2.000000 ]

                                0          x1          x2          x1*x1       x1*x2       x2*x2
      Otterï¿½ la riga: [  1.000000   2.000000   2.000000   4.000000   4.000000   4.000000   ]
 */
      int dimensioneCorrente=1;
      int j=1;
      int tmp,i;
      float prod;
      int x=dim(d,degree) +1 ; //Calcolo la lunghezza della riga, numero combinazioni x*
      x = (((x/16)+1)*16);
      int inizioRestanti;
      float* res=(float*)get_block(sizeof(float), x);
      res[0]=1.0;
      for (dimensioneCorrente; dimensioneCorrente<=degree; dimensioneCorrente++) {
   /*Calcola quanti elementi ha la dimensione corrente Ovvero quante righe scorrere nella matrice degli indici x* */
            tmp=dimNow(d,dimensioneCorrente);
            while(tmp>0){
                  prod=1.0;
    /*Il ciclo for cicla gli elementi di una singola riga della matrice degli indici
      Ovvero una singola conbinazioni degli indici*/
                  for(i=0;i<dimensioneCorrente;i++){
                        prod=prod*arr[posizione*d+mat[j][i]-1];
                  }
                  res[j]=prod;
                  j++;
                  inizioRestanti = j;
                  tmp--;
            }
      }

      for(inizioRestanti; inizioRestanti < x; inizioRestanti++){
            res[inizioRestanti] = 0.0;
      }
      return res;
}

float** calcolaValoriXStar(int** indici, float*x, int n, int d, int degree){
      float** xStarRes=(float**)get_block(sizeof(float*),n);
      int i;
 /*
      Abbiamo memorizzato tutte le combinazioni di coefficienti ( x* ) dentro
      la matrice xStarRes. Ora xStarRes[i] "linearizza" la matrice per poi compiere
      una sostituzione per ognuna delle entry del dataset
      Ad esempio se xStar[i]= [1, 1] (coefficente x1*x1), con x1=5 l'osservazione 1 della riga
      i-esima. Calcolo e inserisco nella nuova matrice 5*5
 */
       /*Ogni iterazione del for calcola xi* ovvero sostituisce ad x* i valori
       dell'array x corrente */

      for(i=0;i<n;i++){
            xStarRes[i]=calcolaRigaXStar(indici, x, i, degree, d);
      }
      return xStarRes;
}
```

Qui il metodo che dati gli indici va a calcolare la matrice risultante andando a generare le singole righe, come vediamo allochiamo sia la memoria non contigua, ergo più tempo per le operazioni di lettura e scrittura e non veniva neanche utilizzato il paradigma openMP o il meccanismo del LOOP Unrolling.

```
float* convert_data(params* input){
      // ---------------------------------------------
      // Codificare qui l'algoritmo di conversione
      // ---------------------------------------------
      int** m = genera_xStar(input->d, input->degree);
      float** ciao = calcolaValoriXStar(m, input->x, input->n, input->d, input->degree);
      return ciao;
}
```

Effettivamente, come possiamo osservare da i test temporale eseguiti di seguito (d=4, degree=4), otteniamo una miglioria temporale se usiamo una versione vettorizzata, rispetto ad una versione matriciale, sia da un punto di vista di allocazione della memoria, sia da un punto di vista di utilizzo e accesso alla struttura dati.

```
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 20
Degree: 4
Eta: 0.010000
Adagrad disabled
Conversion time = 0.001885 secs
```

Test versione matriciale senza ottimizzazioni

```
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 20
Degree: 4
Eta: 0.010000
Adagrad disabled
Conversion time = 0.001570 secs
```

Test versione vettorizzata con ottimizzazioni (senza OpenMp)

Inoltre, possiamo anche apprezzare l'efficienza ulteriore che aggiunge l'utilizzo del paradigma **openmp:**

```
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 20
Degree: 4
Eta: 0.010000
Adagrad disabled
Conversion time = 0.000503 secs
```

Test versione vettorizzata con ottimizzazioni (con OpenMp)

Dimostrazione inefficienza utilizzo repertorio sse per la conversione

Come abbiamo già apprezzato, per quello che è stata la nostra idea progettuale e l'algoritmo che è stato ottenuto, nella fase di costruzione della matrice delle osservazioni quando bisogna andare a prelevare l'elemento **i** e quindi sostituirlo con il valore **xij** (i, indice di riga per il gruppo di x di una osservazione e j l'indice della variabile specifica), questi valori non sono contigui in memoria quindi non si presta prettamente all'utilizzo del repertorio **sse**.

Per mostrare ciò è stata sviluppata una versione differente del codice in C in cui sono stati modificati sia il metodo **genera_xStar** che **generaValoriXStar.** L'obiettivo del metodo genera_xStar sarà quello di ottenere una matrice di questo tipo (con d=4 e degree=2, scelto degree=2 per motivi di stampa):

0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
1	2	0	0	0	0	0	0
1	3	0	0	0	0	0	0
1	4	0	0	0	0	0	0
2	2	0	0	0	0	0	0
2	3	0	0	0	0	0	0
2	4	0	0	0	0	0	0
3	3	0	0	0	0	0	0
3	4	0	0	0	0	0	0
4	4	0	0	0	0	0	0

Qui come possiamo vedere generata una matrice uguale alla prima solo che non sarà più sparsa, ma bensì avrà padding fino ad 8 elementi, questo in modo tale da ottimizzare la produttoria in assembly degli elementi di una riga. Quindi il metodo che genera gli indici genera una matrice non più sparsa, ma avrà dimensione delle righe dato dal metodo **dimPaddingXStar** che è analogo al metodo **dimPadding**, solo con dimensione del padding pari ad 8 anziché 16.

Qui possiamo osservare il metodo modificato, va specificato che questa dimostrazione sfrutta le ottimizzazioni di vettorizzazione e di LOOP Unrolling ottenute nella versione standard del problema (senza usare OpenMp):

```
int indice;
//int lunghezza;
void xi_star(int * x, int * pos, int num_corr, int h, int at, int d, int* mat,int paddingConvertion){
	//popolo matrice allocata con il metodo "alloca" solo per il blocco
	//di dimNow righe (ovvero dimNow elementi)
	int i;

	if (num_corr == h) { //condizione di uscita della ricorsione
		if (!pos) return;
		int i=0;
		for (; i < h; i++){
			mat[indice+i]=x[pos[i]];
		}

		for(;i<paddingConvertion;i++)
			mat[indice+i]=0;
	indice+=paddingConvertion;
	return 1;
	}

	for (i = at; i < d; i++) {
		if (pos) // MECCANISMO PER NON ANDARE IN CORE DUMP
		pos[num_corr] = i;
		xi_star(x,pos, num_corr + 1, h, i, d,mat,paddingConvertion);
	}
}

int* genera_xStar(int d, int degree){
	//riempie l'intera matrice allocata con gli indici delle combinazioni per x*
	int* coordinate=(int*)get_block(sizeof(int), d);
	int lunghezza=dim(degree,d)+1;
	int h;
	for(h=1;h<=d;h++){
		coordinate[h-1]=h; // genera [1, 2, ...., n] che corrispondono a x1, x2, ..., xn
	}
	int pos[degree]; // pari al massimo valore che assumerà  quindi pari a degree
	int paddingConvertion=dimPaddingXStar(degree);
	indice=paddingConvertion;
	int* m=(int*)get_block(sizeof(int),lunghezza*paddingConvertion);
	for(int t=0;t<paddingConvertion;t++)
		m[t]=0;
	//richiamo x_star per ogni blocco di dimNow elementi da riempire
	for(h=1;h <= degree; h++){
		xi_star(coordinate,pos, 0, h, 0, d,m,paddingConvertion);

	}
	return m;
}
```

Il metodo **calcolaValoriXStar** ora genererà una matrice in cui in ogni riga viene replicata l'intera matrice degli indici con sostituzione agli 1, 2, 3… di quest'ultima con x1, x2, x3…, mentre gli 0 di padding sostituiti con l'elemento neutro della moltiplicazione, ossia 1.

Qui possiamo osservare il metodo **convert_data** e il risultato che dovremmo ottenere, sempre considerando anche le righe di padding a 0 per i metodi di SGD, prima di mandare in esecuzione i metodi si SDG:

```
extern void convert(float* tmp, float* res, int n, int pR, int pC, int realC);

float* convert_data(params* input){
	// ------------------------------------------------
	// Codificare qui l'algoritmo di conversione
	// ------------------------------------------------

	int* m = genera_xStar(input->d, input->degree);
	int pC=dimPaddingXStar(input->degree);
	int pFAKE=dim(input->d,input->degree)+1;
	int pR=dimPadding(pFAKE);

	float* x=input->x;
	int n=input->n;
	int d=input->d;
	int degree=input->degree;
	float* tmp=(float*)get_block(sizeof(float),n*pR*pC);

	for(int i=0;i<n;i++){
		int j=0;
		for(;j<pFAKE;j++){
			for(int k=0;k<pC;k++){
				int now=m[j*pC+k];
				if(now==0)
					tmp[i*pR*pC+j*pC+k]=1.0;
				else
					tmp[i*pR*pC+j*pC+k]=x[i*d+now-1];
			}
		}
		for(;j<pR;j++)
			for(int k=0;k<pC;k++)
				tmp[i*pR*pC+j*pC+k]=0.0;
	}
	float* res=(float*)get_block(sizeof(float),n*pR);

	convert(tmp, res, n, pR, pC, pC/8);

	return res;
}
```

Questo metodo, inizialmente tramite la chiamata a **genera_xStar** la matrice degli indici con il padding di 8 elementi che abbiamo visto precedentemente. Successivamente genera una matrice in cui per ognuna delle ennesime righe, costruisce una riga con l'intera matrice degli indici vista precedentemente dove andiamo a sostituire ad ogni posizione **i,** il valore **xi** corrispondente, in modo tale da eseguire la produttoria degli elementi della stessa riga in assembly.

Osserviamo che se la matrice degli indici conteneva 0.0, questo verrà sostituito dall'elemento neutro della moltiplicazione cioè 1.0, in modo da non dare problemi nella produttoria, nel fare padding si metterà effettivamente il valore 0.0, dato che su quelle righe il valore dovrà effettivamente essere 0.0 alla fine.

Qui un breve esempio con la riga x=[0.32, 0.53, 0.12, 0.08], d=4 e degree=2:

1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.32	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.53	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.12	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.08	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.32	0.32	1.00	1.00	1.00	1.00	1.00	1.00
0.32	0.53	1.00	1.00	1.00	1.00	1.00	1.00
0.32	0.12	1.00	1.00	1.00	1.00	1.00	1.00
0.32	0.08	1.00	1.00	1.00	1.00	1.00	1.00
0.53	0.53	1.00	1.00	1.00	1.00	1.00	1.00
0.53	0.12	1.00	1.00	1.00	1.00	1.00	1.00
0.53	0.08	1.00	1.00	1.00	1.00	1.00	1.00
0.12	0.12	1.00	1.00	1.00	1.00	1.00	1.00
0.12	0.08	1.00	1.00	1.00	1.00	1.00	1.00
0.08	0.08	1.00	1.00	1.00	1.00	1.00	1.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Possiamo osservare l'ultima riga vuota dato che essendo per d=4 e degree=2 un totale di 15 righe, facendo padding ad 16 vengono inserite le righe 0.

In questo modo in assembly viene calcolata la matrice finale ottenuta andando a moltiplicare fra loro tutti gli elementi di una riga a blocchi di 8 usando 2 registri XMM in modo da sfruttare le potenzialità del repertorio.

```
convert(tmp, res, n, pR, pC, pC/8);
```

Osserviamo che alla chiamata del metodo in C metodo viene passato anche pc/8, dove va ad indicare quanti cicli dovrà ripetere a blocchi di 8 per completare una riga.

```
%include "sseutils32.nasm"

section .data                   ; Sezione contenente dati inizializzati
      dim equ 4
      align 16
      uno  dd    1.0,1.0,1.0,1.0

section .bss                    ; Sezione contenente dati non inizializzati
      alignb 16
      n           resd 1
      alignb 16
      pR          resd 1
      alignb 16
      pC          resd 1
      alignb 16
      RC          resd 1
      alignb 16
      c8          resd 1

      alignb 16
      indice resd 1

      alignb      16
      temp        resd 4

section .text                   ; Sezione contenente il codice macchina

      indici            equ  8
      res               equ  12
      num               equ  16
      paddR             equ  20
      paddC             equ  24
      realC             equ  28

global convert

convert:
                        ;
                        ;sequenza Ingresso
                        ;
                        start
                        ;
                        ;lettura dei parametri
                        ;
                        mov       eax, [ebp+indici]         ;EAX<- puntatore agli indici

                        mov       ebx, [ebp+res]                  ;EBX<- puntatore a res

                        mov       ecx, [ebp+num]
                        mov       [n], ecx                  ;ECX<- n

                        mov       ecx, [ebp+paddR]
                        mov       [pR], ecx                 ;pR<- paddingRighe

                        mov       ecx, [ebp+paddC]
                        mov       [pC], ecx                 ;pC<- paddingColonne

                        mov       ecx, [pC]
                        imul            ecx, [pR]
                        mov       [RC], ecx                 ;RC<- paddRighe*paddColonne
                        ;
                        mov       ecx, [ebp+realC]
                        mov       [c8], ecx                 ;pC<- paddingColonne

                        ;
```

Qui osserviamo la lettura dei parametri e il salvataggio del loro valore/puntatore in aree di memoria locali al metodo **convert**.

```
;
        MOV     ESI, 0              ;i=0
fori:
        MOV     EDI, 0              ;j=0
forj:
        MOV         ECX, 0              ;k=0
        MOVAPS  XMM0, [uno]
fork:
        MOV     EDX, ESI
        IMUL    EDX, [RC]           ;in EDX ho i*pR*pC
        MOV         [indice], EDX
        MOV     EDX, EDI
        IMUL    EDX, [pC]           ;in EDX ho j*pC
        ADD     EDX, [indice]
        ADD     EDX, ECX
        IMUL    EDX, dim            ; in EDX ora ho (i*pR*pC+j*pC+k)*dim
        CVTSI2SS XMM7,ESI
        MOVAPS  [temp], XMM7
        ;printss    temp
        CVTSI2SS XMM7,EDI
        MOVAPS  [temp], XMM7
        ;printss    temp
        ;
        MOVAPS  XMM1, [EAX+EDX]
        movaps  [temp], XMM1
        ;printps    temp,1

        MOVAPS  XMM2, [EAX+EDX+16]
        movaps  [temp], XMM2
        ;printps    temp,1

        MULPS   XMM1, XMM2
        movaps  [temp], XMM1
        ;printps    temp,1

        MOVAPS  XMM2,XMM1
        SHUFPS  XMM2,XMM2, 01001110b
        movaps  [temp], XMM2
        ;printps    temp,1

        MULPS   XMM1, XMM2
        movaps  [temp], XMM1
        ;printps    temp,1

        SHUFPS  XMM2, XMM1, 10010011B
        movaps  [temp], XMM2
        ;printps    temp,1

        MULPS   XMM1, XMM2
        movaps  [temp], XMM1
        ;printss    temp

        MULSS   XMM0, XMM1      ;nella prima posizione di XMM0 ho il porodotto degli 0 elementi

        INC     ECX
        CMP     ECX, [c8]
        JL      fork
        ; fine ciclo k

        MOV         ECX, ESI
        IMUL    ECX, [pR]
        ADD     ECX, EDI
        IMUL    ECX, dim                ;ora in ECX ho (i*pR+j)*dim
        MOVSS   [EBX+ECX], XMM0

        INC     EDI
        CMP         EDI, [pR]
        JL    forj
        ;fine ciclo j



        INC     ESI
        CMP         ESI, [n]
        JL      fori
        ;fine ciclo i
        ;
        ;ripristino stack
        ;
        stop
```

Il codice assembly appena visto è la traduzione del seguente:

```
float* convert1(float* indici, float* res, int n, int paddR, int paddC, int otto){
    for(int i=0;i<n;i++){
        for(int j=0;j<paddR;j++){
            float tmp=1.0;
            for(int k=0;k<otto;k+=4){
                tmp=tmp*indici[i*paddR*paddC+j*paddC+k];
                tmp=tmp*indici[i*paddR*paddC+j*paddC+k+1];
                tmp=tmp*indici[i*paddR*paddC+j*paddC+k+2];
                tmp=tmp*indici[i*paddR*paddC+j*paddC+k+3];
            }
            res[i*paddR+j]=tmp;
        }
    }

    return res;
}
```

Ora, andando a controllare le differenze temporali tra i due meccanismi, possiamo osservare come in questo caso non risulti essere efficiente l'utilizzo del repertorio sse data la sparsità degli indici da utilizzare per la costruzione della matrice da utilizzare in sse:

```
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 20
Degree: 4
Eta: 0.010000
Adagrad disabled
Conversion time = 0.005629 secs
```

Test versione per la dimostrazione

```
Data set size [n]: 2000
Number of dimensions [d]: 4
Batch dimension: 20
Degree: 4
Eta: 0.010000
Adagrad disabled
Conversion time = 0.001533 secs
```

Test versione normale (senza OpenMp)

Per apprezzare ancora di più le differenze temporali è stato utilizzato un degree=4, perché con degree più bassi la differenza è molto sottile, ovviamente all'aumentare dello spazio occupato aumenta il tempo di costruzione.

SGD Batch (32)

La realizzazione del codice è partita dallo studio del problema e un primo approccio allo stesso in codice C. E' proposta la seguente soluzione.

```
float* sgdBatch(params* input,float* osservazioni, int lenght){

    //start sgd
    float* theta=(float*)get_block(sizeof(float), lenght);
    int it=0;
    int i;
    float* y=input->y;
    int iter=input->iter;
    int n=input->n;
    float rate=input->eta;
    int k=input->k;

    for(int it = 0; it<iter; it++){
        float tmp[lenght];
        for(i=0; i<n; i+=k){

            for(int l=0;l<=lenght;l++)
                tmp[l]=0.0;

            int j;
            float prodScal;
            int v;
            if(n-i>k)
                v=k;
            else
                v=n-i;

            for(int p=i; p<i+v;p++){
                prodScal=0;
                //prodotto scalare
                for(j=0;j<lenght;j++){
                    prodScal+=theta[j]*osservazioni[p*lenght+j];
                }

                //sottraggo yi
                prodScal=prodScal-y[p];


                //moltiplico per xi*
                for(j=0;j<lenght;j++){
                    tmp[j]+=osservazioni[p*lenght+j]*prodScal;
                }

            }

            //sottraggo a theta
            for(j=0;j<lenght;j++){
                theta[j]=theta[j]-tmp[j]*rate/v;
            }

        }
    }
    stampaEQM(input, theta, osservazioni, input->d, input->degree);
    return theta;
}
```

Ci riferiremo a questo for su j come forj1

Questo for verrà identificato come forj2

Questo for verrà identificato come forj3

A seguito dell'inizializzazione delle strutture dati necessarie, viene avviato il for per eseguire le iterazioni necessarie. A ogni iterazione viene azzerato tmp (array che conterrà la sommatoria evidenziata nel calcolo di theta), mentre il theta calcolato all'iterazione precedente viene mantenuto in memoria, per far si che ogni iterazione vada a raffinarlo e renderlo più preciso, garantendo risultati convergenti verso la soluzione.

$$\Theta := \Theta - \eta \frac{1}{\nu} \sum_{j=i}^{i+\nu-1} (\langle \Theta, \mathbf{x}_j^* \rangle - y_j) \cdot \mathbf{x}_j^*$$

Il calcolo della variabile v rappresenta la determinazione delle osserazioni che verranno attualmente analizzate (nel corrente passo di batch), difatti il for successivo (con indice p) itera sulle stesse, partendo da i (indice della prima osservazione del passo di batch corrente) e arrivando a i+v.

Nella matrice osservazioni, linearizzata per righe, saranno contenuti i valori delle varie osservazioni del DB fornito, già moltiplicate per x*, basterà quindi compiere il prodotto per il theta fornito (da inserire in prodscal) e sottrarvi la y corrispondente (della osservazione associata).

Fatto cioò moltiplichiamo il valore ottenuto per l'osservazione corrente e ne cumuliamo i risultati in tmp (in cui alla fine delle iterazioni sul passo di batch troveremo il risultato della sommatoria.

Segue l'aggiornamento di theta sulla base del calcolo della sommatoria compiuta , che deve essere moltiplicata per i parametri eta e 1/v.

Ottimizzazioni in Assembly

Una prima soluzione testata è stata il trasferimento dell'intero codice in assembly, scrivendone una versione macchina.

Nella versione descritta sono state sfruttate le tecniche di **Loop Unrolling** e **Loop Vectorization.**

Loop Unrolling sono state replicate all'interno della trasposizione in assembly di ogni ciclo le istruzioni, introducendo un fattore di unrolling pari a 4. Questa scelta è applicata in combinazione all'utilizzo del repertorio SSE, per sfruttare parallelismo SIMD (quindi Loop Vectorization), permette quindi di leggere 4 elementi alla volta (grazie a registri SSE) e nel complesso gestire sedici elementi in un singolo passo del ciclo (ripetiamo 4 volte ogni operazione, utilizzando ciascuna delle 4 volte registri SSE).

Risulta evidente quindi la motivazione del padding compiuto su dimensione 16. Leggendo 16 elementi di riga (matrice linearizzata per righe) alla volta, c'è bisogno di assicurarsi che il numero di elementi di riga sia un multiplo esatto di 16

```
int dimPadding(int dimensioneReale){
	int d=dimensioneReale/16;
	int r=dimensioneReale%16;
	if(r==0) return dimensioneReale;
	else return (d+1)*16;
}
```

Di seguito è riportata la versione in C appena descritta

```
extern void batch32(params* input, float* osservazioni, float* theta, int lenght, float* tmp);

float* sgdBatch(params* input, int lenght){

    //start sgd
    float* osservazioni=input->xast;
    lenght=dimPadding(lenght);
    float* theta=(float*)get_block(sizeof(float), lenght);
    float* tmp=(float*)get_block(sizeof(float), lenght);

    int iter=input->iter;

    for(int i = 0; i < lenght; i++){
        theta[i]=0.0;
        tmp[i]=0.0;
    }

    //#pragma omp parallel for
    for(int it = 0; it<iter; it++){
        batch32(input, osservazioni, theta, lenght, tmp);
    }
    return theta;
}
```

Osserviamo il codice nasm contenente la funzione batch32

```
batch32:
        ; ------------------------------
        ; Sequenza di ingresso nella funz
        ; ------------------------------
        push    ebp
        mov     ebp, esp
        push    ebx
        push    esi
        push    edi


        MOV EAX, [EBP + startlenght]
        MOV [lengthB], EAX

        MOV EAX, [EBP+input]
        MOV EBX, [EAX + 12]
        MOV [nB], EBX
        MOV EBX, [EAX + 20]
        MOV [kB], EBX
        MOV EBX, [EAX + 28]
        MOV [rateB], EBX

        MOV     EAX, [EBP+starttmpB]

        MOV     EBX, 0                  ;
foriB:
        MOV [iB], EBX
        MOV     ESI, 0                  ;
        MOV     EDI, 0                  ;
forlB:
        XORPS   XMM0, XMM0              ;
        MOVAPS  [ EAX + ESI ], XMM0
        MOVAPS  [ EAX + ESI  + 16], XMM0
        MOVAPS  [ EAX + ESI  + 32], XMM0
        MOVAPS  [ EAX + ESI  + 48], XMM0
        ADD     ESI, elementiunrollingB
        ADD     EDI, 16
        CMP     EDI, [lengthB]
        JL      forlB
        ;---------------------
        ;CALCOLO DI V
        ;---------------------
        MOV     ECX, [nB]
        SUB     ECX, EBX
        CMP ECX, [kB]
        JLE         saltoB

        MOV ECX, [kB]
```

A seguito della lettura dei parametri passati viene avviato il for che itera sui diversi passi di batch selezionate (etichetta *foriB*)

Il valore del corrente indice di osservazione (contenuto in EBX), viene salvato in [iB], vengono di seguito azzerati ESI (indice di colonna per scorrere una singola osservazione o theta o tmp, aventi stesso numero di colonne) e EDI (indice per contare il numero di colonne lette e fermarci una volta arrivati a fine colonna)

Viene spostato in EAX il puntatore alla struttura dati tmp.

All'interno di *forlB* vengono sfruttati gli indici descritti per navigare tmp. XMM0 conterrà gli zeri per sovrascrivere tmp, che viene quindi azzerato, ricordiamo che il vettore è passato al metodo da C

Viene successivamente calcolato v, il numero di osservazioni del corrente passo di batch (corrente iterazione *foriB*). Calcoliamo prima n – i, nel caso in cui questo valore dovesse essere minore di k (passo di batch effettivo), bisogna usarlo (saltando riga MOV ECX, [kB]), vorrà dire che le osservazioni rimaste da analizzare, sono in numero minori di quelle da scorrere a ogni passo.

ESI rappresenta l'indice j (di colonna), viene incrementato di 64 (elementiUnrolling) perché ad ogni ciclo si leggono 16 elementi, ciascuno occupa 4 byte (essendo float) e di conseguenza il prossimo indirizzo di partenza sarà dato da

indirizzoTmp di Base + j*dim ovvero indirizzoTmp precedente + 64

Oltre a questo incremento si considera lo sfasamento di 16 posizioni che si ha dopo aver compiuto ogni lettura SSE, quindi si accede a partire da indice j corrente (ESI) a j+16 (saltando i primi 4 elementi letti in precedenza), poi j+32 e così via

Riferimento al codice C

```
for(int it = 0; it<iter; it++){
    float tmp[lenght];
    for(i=0; i<n; i+=k){

        for(int l=0;l<=lenght;l++)
            tmp[l]=0.0;

        int j;
        float prodScal;
        int v;
        if(n-i>k)
            v=k;
        else
            v=n-i;
```

```
        MOV [vB], ECX
        MOV     EDX, EBX
        MOV ESI, EBX
        ADD     ESI, ECX
        ;edx ho p e in esi ho i+v

        MOV [limiteB], ESI
        |
forpB:
        XORPS   XMM1, XMM1
        MOV     EDI, 0
        MOV     ESI, 0
        MOV     ECX, EDX
        IMUL    ECX, [lengthB]
        IMUL    ECX, ECX, dimB
```

Memorizziamo il valore di v appena calcolato, nell'area di memoria [vB]. Memorizziamo in EDX il corrente indice di riga (osservazione corrente) e in ESI l'indice di fine del corrente passo di batch (i+v) (osservazione a cui il corrente passo di batch deve fermarsi). Memorizziamo i+v in [limiteB].

Avviamo a questo punto *forpB*, di effettivo scorrimento delle osservazioni del corrente passo di batch. XMM1 conterrà prodScal, che deve essere azzerato e riguarda l'osservazione corrente.

A causa della linearizzazione delle strutture dati passate al metodo, a partire dall'indice di riga corrente, vinee calcolato l'indirizzo del puntatore che si riferisce a una specifica osservazione, come p*length*dim, ovvero indice di riga*numero elementi di riga* numero di byte per elemento. Vediamo il codice C di riferimento.

```
for(int p=i; p<i+v;p++){
    prodScal=0;
```

Il successivo codice si riferisce al forj1 del codice C, per calcolare prodScal di una singola osservazione

```
        MOV     [pB], EDX
        MOV EBX,[EBP + theta]
        MOV EDX,[EBP + osserv]

forj1B:
        MOVAPS  XMM2, [EBX + EDI]
        MULPS   XMM2, [EDX + ECX]
        ADDPS   XMM1, XMM2
        MOVAPS  XMM3, [EBX + EDI + 16]
        MULPS   XMM3, [EDX + ECX + 16]
        ADDPS   XMM1, XMM3
        MOVAPS  XMM4, [EBX + EDI + 32]
        MULPS   XMM4, [EDX + ECX + 32]
        ADDPS   XMM1, XMM4
        MOVAPS  XMM5, [EBX + EDI + 48]
        MULPS   XMM5, [EDX + ECX + 48]
        ADDPS   XMM1, XMM5

        ;questo chiude il forj1
        ADD     ESI, 16
        ADD     EDI, elementiunrollingB
        ADD     ECX, elementiunrollingB
        CMP ESI, [lengthB]
        JL      forj1B

        HADDPS  XMM1, XMM1
        HADDPS  XMM1, XMM1
```

Vengono spostati i puntatori alle strutture dati passate nei registri m32. In EBX avremo theta in EDX osservazioni. Nel for che segue si scorre per ogni osservazione (una stessa riga), un insieme di 16 elementi (indici di colonna) alla volta.

Ogni MOVAPS legge in contemporanea 4 elementi float di theta e li sposta in un registro XMM, il contenuto dello stesso registro XMM viene poi moltiplicato per le 4 colonne corrispondenti di theta (ogni coordinata di osservazioni vinee moltiplicata per la corrispondente coordinata di theta)

Infine si cumulano in XMM1 con somma in parallelo, i valori appena calcolati e quelli calcolati in precedenza.

XMM1 conterrà prodScal, di conseguenza sarà calcolato dal contributo (sommatoria) di tutte le coordinate della corrente osservazione, moltiplicate per la corrispondente coordinata in theta. E' importante quindi mantenere la distinzione fra colonne al momento della moltiplicazione, ma in prodScal si possono sommare indistintamente i contributi di tutti, alla fine del for (che similmente a prima) controlla quando è terminata l'iterazione sulle colonne, si compie una doppia HADDPS, che ha il compito di sommare le 4 somme parziali formatesi (avendo lavorato in SSE). XMM1, infatti, conterrà all'indice 0 la sommatoria di tutti gli elementi iniziale di ogni quadrupla letta, all'indice 1 avremo tutti i contributi degli elementi che si trovavano in posizione 1 negli XMM che sono stati sommati a XMM1 e così via).

L’equivalente in C del codice appena visto

```
for(j=0;j<lenght;j++){
    prodScal+=theta[j]*osservazioni[p*lenght+j];
}
```

```
MOV EDX, [EBP + input]
MOV EBX, [EDX+4]

;ripristino l'indice p giusto
MOV     EDX, [pB]

IMUL    EDI, EDX, dimB
SUBSS   XMM1, [EBX + EDI]
SHUFPS  XMM1, XMM1, 00000000
```

Dopo aver ricavato prodScal viene sottratto allo stesso il valore della Y corrispondente alla osservazione da cui è stato calcolato.

Spostiamo l’indice della struttura dati y nel registro EBX, successivamente calcoliamo la cella di y che si riferisce alla osservazione corrente, come Base Indirizzo Y + nrCellePer float + indice osservazione (ovvero y + p*dim).

Immaginando la matrice osservazioni (linearizzata) nella sua forma originale, ogni osservazione corrisponde a una riga, ed è identificata dall’indice di riga corrente, p. il vettore y ha numero di celle pari al numero di righe di osservazioni, perché avremo un valore di y per ciascuna osservazione. Di conseguenza data l’osservazione all’indice p, ci spostiamo all’indice p di y per leggere la y corrispondente.

Salviamo il valore della sottrazione in tutte le celle del registro XMM1, così da replicare prodScal in 4 celle e poi replicarlo in altri 3 registri, quando dovremo compiere operazioni su 16 elementi, al prossimo ciclo di for

Equivalente in C:

```
//sottraggo yi
prodScal=prodScal-y[p];
```

```
        MOV     EDI, 0
        IMUL    EDX, [lengthB]
        IMUL    EDX, EDX, dimB


        MOV EBX, [EBP + osserv]
        MOV ECX, 0
forj2B:
        MOVAPS XMM2, XMM1
        MOVAPS XMM4, XMM1
        MOVAPS XMM6, XMM1

        MULPS   XMM1, [EBX + EDX]

        MULPS   XMM2, [EBX + EDX + 16]

        MULPS   XMM4, [EBX + EDX + 32]

        MULPS   XMM6, [EBX + EDX + 48]

        ADDPS   XMM1, [EAX + ECX]
        ADDPS   XMM2, [EAX + ECX + 16]
        ADDPS   XMM4, [EAX + ECX + 32]
        ADDPS   XMM6, [EAX + ECX + 48]


        MOVAPS  [EAX + ECX], XMM1
        MOVAPS  [EAX + ECX + 16], XMM2
        MOVAPS  [EAX + ECX + 32], XMM4
        MOVAPS  [EAX + ECX + 48], XMM6

        ;questo chiude il forj2
        ADD     EDI, 16
        ADD     ECX, elementiunrollingB
        ADD     EDX, elementiunrollingB
        CMP EDI, [lengthB]
        JL      forj2B
```

Il *forj2B*, rappresenta l'esecuzione della sommatoria, incrementa tmp in funzione del contributo della osservazione corrente e del prodScal che genera.

In ogni cella di XMM1 è stato salvato il valore di prodScal, viene copiato nei registri XMM2,4,6 (MOVAPS)

A questo punto si moltiplica prodScal per ogni coordinata di osservazioni e si cumulano i 4 risultati. MULPS XMM, [EBX+EDX]

È importante in questo caso mantenere separati i risultati derivanti da ciascuna colonna, in quanto ognuna corrisponderà a una specifica coordinata dell'osservazione originale, che dovrà cumularsi con le coordinate analoghe delle altre osservazioni, in una specifica colonna di tmp.

Dopo aver moltiplicato (separatamente) i 16 valori (parte delle colonne della corrente osservazione) per prodScal (uguale per tutti) si somma il valore (aggiornato al passo precedente) di tmp, nelle colonne corrispondenti e si aggiorna tmp ADDPS XMM, [EAX + ECX] e poi aggiornamento tmp con MOVAPS [EAX + ECX], XMM

È necessario mantenere due puntatori alla "colonna" (j) differenti, in quanto EDX (puntatore alla colonna di osservazioni) tiene conto anche degli indirizzi da saltare in quanto di osservazioni precedenti (parte da p*length*dim e ad ogni ciclo punta ai 16 float successivi a quelli del ciclo precedente, incrementando di 64 indirizzi).

EAX deve essere un semplice puntatore che scorre tmp, array (e non matrice linearizzata) quindi incrementa sempre di 64, in quanto da un ciclo all'altro si scorrono 16 float, ma non tiene conto di alcun indice di riga (parte da 0)

Equivalente in C

```c
//moltiplico per xi*
for(j=0;j<lenght;j++){
    tmp[j]+=osservazioni[p*lenght+j]*prodScal;
}
```

```
    MOVSS   XMM0, [rateB]
    CVTSI2SS    XMM1, [vB]
    DIVSS   XMM0, XMM1
    SHUFPS  XMM0, XMM0, 00000000

    MOVAPS  [rapportoB], XMM0


    MOV     EDI, 0
    MOV     ECX, 0
    MOV EBX, [EBP + theta]
```

A questo punto vengono letti i valori forniti in input all'esecuzione dell'algoritmo, rate e v, vengono convertiti in float per evitare che usandoli come interi siano interpretati come divisione per 0 (dividendo float per un intero e poi considerandolo float)

Viene calcolato rate/v e memorizzato nelle 4 celle di XMM0

```
forj3B:
    MOVAPS  XMM2, [EAX + EDI]
    MOVAPS  XMM3, [EAX + EDI + 16]
    MOVAPS  XMM4, [EAX + EDI + 32]
    MOVAPS  XMM5, [EAX + EDI + 48]

    MULPS   XMM2, [rapportoB]
    MULPS   XMM3, [rapportoB]
    MULPS   XMM4, [rapportoB]
    MULPS   XMM5, [rapportoB]

    MOVAPS  XMM0, [EBX + EDI]
    MOVAPS  XMM1, [EBX + EDI + 16]
    MOVAPS  XMM6, [EBX + EDI + 32]
    MOVAPS  XMM7, [EBX + EDI + 48]

    SUBPS   XMM0, XMM2
    SUBPS   XMM1, XMM3
    SUBPS   XMM6, XMM4
    SUBPS   XMM7, XMM5

    MOVAPS  [EBX + EDI], XMM0
    MOVAPS  [EBX + EDI + 16], XMM1
    MOVAPS  [EBX + EDI + 32], XMM6
    MOVAPS  [EBX + EDI + 48], XMM7

    ;questo chiude il forj3
    ADD     ECX, 16
    ADD     EDI, elementiunrollingB
    CMP ECX, [lengthB]
    JL      forj3B
```

Nel *forj3b* si legge nuovamente tmp, se ne moltiplica ogni elemento per la quantità calcolata (rate/v)

In altri 4 registri si legge theta, facendo attenzione a compiere la giusta associazione fra coordinate di theta e coordinate di tmp.

Ogni coordinata di theta viene sottratta alla corrispondente coordinata in tmp e successivamente i risultati ottenuti aggiornano theta stesso.

NB

Questa frazione del codice ha determinato il grado di unrolling applicabile, essendo necessaria la lettura di alcuni valori di tmp, ma anche dei corrispettivi valori theta, è stato necessario utilizzare tutti i registri disponibili, da XMM0 a XMM7.

Non sarebbe stato possibile usarne di più, in quanto i valori di tmp devono essere sottratti a theta, ma DOPO la moltiplicazione per rat/v, quindi non sarebbe possibile attuare una soluzione in cui si usano tutti i registri (mettendo theta in XMM) e si compie un operazione del tipo

SUBPS XMM, [EAX + EDI] * [rapporto]

Perché bisogna trasferire necessariamente tmp in un registro prima di moltiplicarlo per rate. Avremmo potuto anche forzare l'uso di tutti i registri in un solo ciclo, replicando il codice due volte, adattando il padding compiuto sui dati, ma avrebbe appesantito troppo il codice e probabilmente garantito risultati simili o di poco migliori

Equivalente in C

```c
//sottraggo a theta
for(j=0;j<lenght;j++){
    theta[j]=theta[j]-tmp[j]*rate/v;
}
```

PERCHE' NON FARE CACHING ?

Secondo la nostra analisi del problema e l'interpretazione dello stesso compiuta, per fare caching avremmo potuto sfruttare il fatto che in un singolo passo di batch, il theta letto rimane invariato, scorrendo le varie osservazioni.

Per quanto riguarda le osservazioni stesse, di esse non viene "riutilizzata" alcuna componente, ogni colonna di osservazione viene usata una volta, successivamente bisonga calcolare prodScal (che necessita lo scorrimento di TUTTE le coordinate dell'osservazione per essere corretta) e poi vinee riutillizzata, quindi non si può fare caching sulle colonne di osservazioni.

Fare Caching sulle righe di osservazioni è altrettanto inutile, perché ogni osservazione differisce dalle altre, si potrebbe però pensare a fare caching su theta, in quanto ogni osservazione (di uno stesso passo di batch) accederà allo stesso theta.
Per fare ciò dovremmo selezionare un frammento delle colonne di osservazioni (per cui theta rimane fissato) e scorrere tutte le righe di osservazioni (non per intero, ma solo nell'intervallo del frammento selezionato).

Questa soluzione non è applicabile, in quanto il primo calcolo che bisogna fare per una determinata osservazione è quello di prodScal, che per essere calcolato (prima di sottrarre y[p]) deve aver cumulato il contributo di **tutte** le colonne della osservazione corrente, non possiamo quindi utilizzare in maniera utile "frammenti" di riga, per favorire caching.

Versione OMP Parallel

Nella versione OpenMP del codice viene sfruttato parallelismo, individuando l'operazione più "critica" eseguita nel codice e migliorabile attraverso questa tecnica: l'iterazione sulle osservazioni del corrente passo di batch.

Dovendo parallelizzare frammenti di codice che potessero interfogliarsi senza causare errori nel codice, è stato necessario individuare quali componenti non necessitassero di eseguire in ordine.

Non è pensabile parallelizzare iterazioni differenti, in quanto ogni iterazione deve avvenire in funzione dei dati (theta) come calcolati nella versione precedente, allo stesso modo non si possono parallelizzare i passi di batch, perché ognuno deve avvenire in ordine, al fnie di incrementare correttamente theta, su cui poggerà il passo di batch successivo.

La componente di codice parallelizzabile è l'esecuzione delle iterazioni di forp, il ciclo che scorre le osservazioni del corrente passo di batch. Ogni osservazione di fatti contribuisce al calcolo di prodScal, ma esso deve essere semplicemente incrementato (in modo cumulativo, somma è ovviamente commutativa) e soprattutto non è necessario (come operando) in alcuna operazione che l'osservazione successiva deve compiere.

Con ispirazione al meccanismo di gestione dei risultati delle singole osservazioni che avviene in Adagrad, viene perciò realizzato tmp come matrice, con numero di righe (pari al numero di osservazioni che vi scriveranno), dato dal passo di batch.
In questo modo ci si tutela dall'unico errore che parallelismo potrebbe comportare (esclusa la possibilità di conflitti logici, date operazioni indipendenti), ovvero, l'accesso contemporaneo di più thread a uno stesso indirizzo (che avverrebbe se tmp fosse semplice array).

Ogni thread si occupa di un dato numero di osservazioni, del corrente passo di batch, ne calcola i risultati parziali e successivamente memorizza il risultato di ogni osservazione nella propria riga di tmp; successivamente si sommano i contributi delle osservazioni, sommando tutte le righe di tmp.

```
float* sgdBatch(params* input, int lenght){

    //start sgd
    float* osservazioni=input->xast;
    lenght=dimPadding(lenght);
    int k=input->k;
    int n=input->n;
    float*y=input->y;
    float rate=input->eta;
    float* theta=(float*)get_block(sizeof(float), lenght);
    float* tmp=(float*)get_block(sizeof(float), k*lenght);
    int iter=input->iter;

    for(int i = 0; i < lenght; i++){
        theta[i]=0.0;
        tmp[i]=0.0;
    }

    for(int it=0;it<iter;it++){

        for(int i=0; i<n; i+=k){ //FOR dei passi di batch

            for(int l=0;l<k*lenght;l++)
                tmp[l]=0.0;

            int v;
            if(n-i>k)
                v=k;
            else
                v=n-i;

            indice=i+v-1;

            Bmetodo2(i, v, theta, osservazioni, lenght, tmp, y);
            Bmetodo1(indice, i, v, theta, osservazioni, lenght, tmp, y);

            for(int j=0;j<lenght;j++){
                for(int i=1;i<v;i++){
                    tmp[j]+=tmp[i*lenght+j];
                }
            }

            float rapporto = rate/v;

            //sottraggo a theta
            for(int j=0;j<lenght;j++){
                theta[j]=theta[j]-tmp[j]*rapporto;
            }

        }
    }
    return theta;
}
```

Il metodo *Bmetodo2*, avvia la computazione parallela delle osservazioni del corrente passo di batch.

In funzione del grado di parallelismo introdotto (numero di osservazioni gestite da ogni thread), si determinano le osservazioni restanti (non gestite da thread) nel caso in cui la dimensione di batch non fosse multiplo esatto delle osservazioni analizzate dai singoli elementi

Si cumula sulla prima riga di tmp la sommatoria dei contributi di tutte le altre righe, infine si sottrae tmp*rapporto a theta, come nella versione non parallela

NB: Come si può notare dai codici che seguono, è stata scelto (in base alle prove condotte) un numero di osservazioni per thread pari a 2. Questo parametro può essere aumentato, facendo in modo che un singolo thread si dedichi a più osservazioni, senza perdita di generalità nel ragionamento e senza intaccare il funzionamento del codice.
Ovviamente decrementando il numero di osservazioni per thread si aumenta l'efficienza, idealmente si arriva a far si che ogni riga del passo di batch sia eseguita in parallelo, questa scelta non è nella pratica applicabile, non avendo un numero infinito di thread lanciabili in parallelo e avendo passi di batch con molte osservazioni

I metodi che seguono sono necessari per applicare correttamente la direttiva di parallelismo pragma, in quanto c'è bisogno di compiere una separazione fra le variabili del codice C e quelle proprie dei thread da avviare e inoltre, come visto a lezione, bisogna fare in modo che la direttiva sia inserita in cima a un for privo di for innestati, per evitare che agisca su altri for e parallelizzi in profondità.

```
extern void batch32Prod(int p, int pfin, float* theta, float* osservazioni, int lenght, float* tmp, float* y, int i);

void BmetodoOPM2(int p, float* theta, float* osservazioni, int lenght, float* tmp, float* y, int i){
    batch32Prod(p,p+2,theta, osservazioni, lenght, tmp,y,i);
}

void BmetodoOPM1(int p, float* theta, float* osservazioni, int lenght, float* tmp, float* y, int i){
    batch32Prod(p,p+1,theta, osservazioni, lenght, tmp,y,i);
}

void Bmetodo2(int i, int v, float* theta, float*osservazioni, int lenght, float*tmp, float*y){
    #pragma omp paraller for
    for(int p=i;p<=i+v-2;p+=2){
        BmetodoOPM2(p, theta, osservazioni, lenght, tmp, y, i);
    }
}

void Bmetodo1(int indice, int i, int v, float* theta, float*osservazioni, int lenght, float*tmp, float*y){
    if(indice%2==1)
        BmetodoOPM1(indice, theta, osservazioni, lenght, tmp, y, i);
}
```

Bmetodo2 avvia parallelismo sulle iterazioni del corrente passo di batch, di conseguenza, incrementa gli indici di riga di 2 elementi alla volta, avendo deciso di far si che ogni thread si riferisca a due osservazioni del corrente passo di batch, al suo interno richiama il metodo batch32, appositamente modificato per interessarsi delle osservazioni che vanno dall'indice p all'inidice pfin (escluso).

Bmetodo1 non sfrutta parallelismo, ma gestisce eventuale osservazione rimasta, nel caso in cui il passo di batch dovesse essere dispari (quindi non divisibile esattamente per 2)

È di seguito riportato il metodo batch32Prod, contenuto nel file assembly regression32OMP, richiamato dal codice C appena analizzato, saltiamo la fase di lettura dei parametri, tenendo conto che **p** rappresenta l'indice della osservazione da cui deve partire corrente analisi (da cui p-i sarà la riga della matrice tmp su cui bisognerà scrivere, rispettivamente); **p+2** è in generale l'indice pfin che definisce "fino a che indice" arrivare, quindi l'ultima osservazione da analizzare (scorrendo le osservazioni contigue da p a pfin-1); **theta** è l'array dei risultat;, **Osservazioni** è la matrice linearizzata delle osservazioni (moltiplicate per x*); **lenght** è la lunghezza dell'array theta (e quindi numero di colonne di tmp e osservazioni), **tmp** è la matrice descritta precedentemente; **y** è vettore passato in input, **i** è indice del corrente passo di batch.

I è necessario perchè matrice tmp ha numero di righe pari alla dimensione di batch, se vi accedessimo in base a p, cercheremmo una riga non presente (nel DB di prova ad esempio p arriva a 2000, ma con batch 20 ho tmp di venti righe, quindi solo prendendo righe date da p-i ottengo sempre un valore valido fra 0 e 19)

```
forpB:
        MOV EDX, [p]

        XORPS   XMM1, XMM1
        XOR     EDI, EDI
        XOR     ESI, ESI
        MOV     ECX, EDX
        IMUL    ECX, [lenght]
        IMUL    ECX, ECX, dim

        MOV EBX,[theta]
        MOV EDX,[osservazioni]
```

forpB scorre le osservazioni previste, azzera per ciascune di esse XMM! (prodScal) e compie operazioni del tutto analoghe alla versione di batch32 priva di parallelizzazione.

```
        MOV     EDI, 0
        MOV     ESI, EDX
        SUB     ESI, [i]
        IMUL    ESI, [lenght]
        IMUL    ESI, ESI, dim

        IMUL    EDX, [lenght]
        IMUL    EDX, EDX, dim

        MOV     EAX, [tmp]
        MOV EBX, [osservazioni]
forj2B:
        MOVAPS  XMM0, XMM1
        MOVAPS  XMM2, XMM1
        MOVAPS  XMM4, XMM1
        MOVAPS  XMM6, XMM1

        MULPS   XMM0, [EBX + EDX]
        MULPS   XMM2, [EBX + EDX + 16]
        MULPS   XMM4, [EBX + EDX + 32]
        MULPS   XMM6, [EBX + EDX + 48]

        MOVAPS  [EAX + ESI], XMM0
        MOVAPS  [EAX + ESI + 16], XMM2
        MOVAPS  [EAX + ESI + 32], XMM4
        MOVAPS  [EAX + ESI + 48], XMM6

        ;questo chiude il forj2
        ADD     EDI, 16
        ADD     ESI, elementiunrolling
        ADD     EDX, elementiunrolling
        CMP EDI, [lenght]
        JL      forj2B

        ; questo chiude il forp
        MOV EDX, [p]
        INC         EDX
        MOV [p], EDX
        MOV     ESI, [pfin]
        CMP     EDX, ESI
        JL      forpB
```

Evidenziamo come questa volta tmp sia gestito come una matrice linearizzata di elementi, quindi vi si accede calcolando (p-i), indice di riga corrente, e moltiplicando il valore calcolato per lenght e poi per dim.

Questo valore rappresenterà lo scostamento relativo dall'indirizzo di base di tmp, quindi la locazione iniziale della "riga" su cui scrivere

Il resto delle operazioni sfrutta le stesse meccaniche descritte nella versione non parallelizzata, con la sostanziale differenza che il nuovo estremo superiore del ciclo *forpB* è adesso ovviamente pFin

SGD Adagrad (32) – David Azzato

La realizzazione del codice è partita dallo studio del problema e un primo approccio allo stesso in codice C. E' proposta la seguente soluzione.

```
extern void adagrad32(params* input, float* osservazioni, float* theta,  int lenght, float* Gj, float* gj, float* sommatoria);
//extern void prova32();
float* sgdAdagrad(params* input, float* osservazioni, int lenght){

    //start sgd
    float* theta=(float*)get_block(sizeof(float), lenght);
    int it=0;
    int i;
    float* y=input->y;
    int iter=input->iter;
    int n=input->n;
    float rate=input->eta;
    int k=input->k;
    float eps=1E-8;

    //prova32();
    float Gj[k * lenght];
    float gj[k * lenght];
    float sommatoria[lenght];

    for(int p1=0;p1<k;p1++){
        for(int p2=0;p2<lenght;p2++){
            Gj[p1 * lenght + p2]=0.0;
            gj[p1 * lenght + p2]=0.0;
        }
    }
```

Nella fase iniziale del codice vengono create e inizializzate le strutture dati necessarie a implementare l'algoritmo risolutivo, nelle righe successive viene descritto il calcolo effettuato, per ogni singola iterazione

Similmente a quanto accade nella versione batch, si ha il for di scorrimento delle diverse iterazioni, al suo interno il for che scorre i diversi passi di batch e infine, per ogni passo di batch, il for che ne scorre le osservazioni.

Per ogni osservazione, vinee eseguito il calcolo di prodScal, in maniera del tutto analoga abatch, ma cambia l'aggiornamento delle strutture dati.

Sono presenti due matrici (linearizzate): **gj** e **Gj**, con il compito di rappresentare in ogni loro riga i risultati di una osservazione del corrente passo di batch. **gj** è l'esatta trasposizione di tmp (nel codice batch), cumula i risultati di una singola osservazione (del corrente passo di batch) e poi viene azzerata al successivo passo.

Gj è invece cumulativa, ogni sua riga contiene il contributo di tutte le osservazioni (di passi di batch precedenti, ma anche iterazioni precedenti) che corrispondevano alla riga stessa. Quanto descritto è evidente dalla descrizione dell'algoritmo fornita alla consegna del problema.

$$g_j := (\langle \Theta, \mathbf{x}_\mathbf{j}^* \rangle - y_j) \cdot \mathbf{x}_\mathbf{j}^*$$

$$G_j := \sum_{t=1}^{it} g_j^2$$

Dopo aver terminato la costruzione di Gj nel corrente passo di batch, si procede al calcolo di theta, secondo la seguente formula:

$$\Theta := \Theta - \frac{1}{\nu} \sum_{j=i}^{i+\nu-1} \frac{\eta}{\sqrt{G_j + \epsilon}} g_j$$

Si noti come Gj non sia cumulativo e comune per ogni osservazione, ma bisogna mantenere la corrispondenza fra: posizione (nel corrente passo di batch e quindi in gj) dell'osservazione di gj che si sta moltiplicando e posizione di Gj ad essa corrispondente (che a sua volta sarà stata "arricchita" da tutte le altre osservazioni che si sono trovate allo stesso indice di riga, in iterazioni o passi di batch precedenti)

Vediamo quindi la parte restante del codice C realizzato

```
    while(it<iter){

        for(i=0; i<n; i=i+k){

            float prodScal;
            int v;
            if(n-i>k)
                v=k;
            else
                v=n-i;

            for(int p=i; p<=i+v-1;p++){
                prodScal=0;

                //prodotto scalare
                for(int j=0;j<lenght;j++){
                    prodScal+=theta[j]*osservazioni[p*lenght+j];
                }

                //sottraggo yi
                prodScal=prodScal-y[p];

                //moltiplico per xi*
                for(int j=0;j<lenght;j++){
                    indice = ((p-i) * lenght) +j;
                    gj[indice]=osservazioni[p*lenght + j]*prodScal;
                    Gj[indice]=Gj[indice]+gj[indice]*gj[indice];
                }
            }

            for(int j=0;j<lenght;j++)
                sommatoria[j]=0.0;

            for(int p=i; p<=i+v-1;p++)
                for(int j=0;j<lenght;j++){
                    indice = ((p-i) * lenght) +j;
                    sommatoria[j]+=(rate/sqrt(Gj[indice]+eps))*gj[indice];
                }
                //sottraggo a theta
                for(int j=0;j<lenght;j++){
                    theta[j]=theta[j]-sommatoria[j]/v;
                }
            }
            it++;

        }
        stampaEQM(input, theta, osservazioni, input->d, input->degree);
        return theta;
}
```

$$g_j := (\langle \Theta, \mathbf{x}_\mathbf{j}^* \rangle - y_j) \cdot \mathbf{x}_\mathbf{j}^*$$

$$G_j := \sum_{t=1}^{it} g_j^2$$

$$\Theta := \Theta - \frac{1}{\nu} \sum_{j=i}^{i+\nu-1} \frac{\eta}{\sqrt{G_j + \epsilon}} g_j$$

Ottimizzazioni in Assembly

Le ottimizzazioni compiute, la scelta delle strutture dati e delle tecniche di velocizzazione sono del tutto analoghe a quelle compiute per batch, sono altrettanto condivise le motivazioni del mancato caching.

Osserviamo quindi il codice assembly, concentrandoci sulle sole differenze sostanziali con batch

```
        MOV     EAX, [gj]
        MOV EBX, [osservazioni]
        MOV ECX, [Gj]

forj2:
        MOVAPS  XMM0, XMM1
        MOVAPS  XMM2, XMM1
        MOVAPS  XMM4, XMM1
        MOVAPS  XMM6, XMM1

        MULPS   XMM0, [EBX + EDX]
        MULPS   XMM2, [EBX + EDX + 16]
        MULPS   XMM4, [EBX + EDX + 32]
        MULPS   XMM6, [EBX + EDX + 48]

        MOVAPS  [EAX + ESI], XMM0
        MOVAPS  [EAX + ESI + 16], XMM2
        MOVAPS  [EAX + ESI + 32], XMM4
        MOVAPS  [EAX + ESI + 48], XMM6

        MULPS   XMM0, XMM0
        MULPS   XMM2, XMM2
        MULPS   XMM4, XMM4
        MULPS   XMM6, XMM6

        ADDPS   XMM0, [ECX + ESI]
        ADDPS   XMM2, [ECX + ESI + 16]
        ADDPS   XMM4, [ECX + ESI + 32]
        ADDPS   XMM6, [ECX + ESI + 48]

        MOVAPS  [ECX + ESI], XMM0
        MOVAPS  [ECX + ESI + 16], XMM2
        MOVAPS  [ECX + ESI + 32], XMM4
        MOVAPS  [ECX + ESI + 48], XMM6

        ;questo chiude il forj2
        ADD     EDI, 16
        ADD     ESI, elementiunrolling
        ADD     EDX, elementiunrolling
        CMP EDI, [lenght]
        JL      forj2

        ; questo chiude il forp
        MOV EDX, [p]
        INC         EDX
        MOV [p], EDX
        MOV     ESI, [pfin]
        CMP     EDX, ESI
        JL      forp
```

In seguito al calcolo di prodScal per ogni osservazione, vengono aggiornate le strutture dati Gj e gj, secondo le formule descritte in precedenza.

Il calcolo di gj è lo stesso di tmp (in batch), mentre Gj ha in ogni sua riga il quadrato della corrispondente riga in gj, sommato al valore precedente di Gj stesso

Si noti come l'accesso alle matrici non debba dipendere dall'indice di riga "assoluto" della osservazione corrente (che va da 0 a n) ma dal suo indice relativo nel corrente passo di batch (che va da 0 a passo di batch, escluso), maniera simile a quanto avviene nella versione OMP di batch

Equivalente in C

```
//moltiplico per xi*
for(int j=0;j<lenght;j++){
    indice = ((p-i) * lenght) +j;
    gj[indice]=osservazioni[p*lenght + j]*prodScal;
    Gj[indice]=Gj[indice]+gj[indice]*gj[indice];
}
```

```
        MOV     ESI, 0                  ;
        MOV     EDI, 0                  ;
        MOV EAX, [EBP + sommatoria]
forl:

        XORPS   XMM0, XMM0              ;
        MOVAPS  [ EAX + ESI ], XMM0
        MOVAPS  [ EAX + ESI  + 16], XMM0
        MOVAPS  [ EAX + ESI  + 32], XMM0
        MOVAPS  [ EAX + ESI  + 48], XMM0
        ADD     ESI, elementiunrolling
        ADD     EDI, 16
        CMP     EDI, [lenght]
        JL      forl

        MOV ESI, [i]
        MOV EDI, [limite]
        MOV [p], ESI

forp2:
        MOV     EAX, [p]
        SUB     EAX, [i]
        IMUL    EAX, [lenght]
        IMUL    EAX, EAX, dim
        MOV [indice], EAX
        MOV EBX, 0
        MOV ECX, [EBP + gj]
```

forl esegue azzeramento del vettore sommatoria, successivamente in *forp2* si scorrono le iterazioni del corrente passo di batch, per formare sommatoria stessa

Calcoliamo p*lenght*dim, da usare come offset per accedere alla giusta locazione (giusta riga) di gj

Equivalente in c

```
for(int j=0;j<lenght;j++)
    sommatoria[j]=0.0;
```

```
forj3:
    MOV [indice], EAX
    MOV EDX, [EBP + Gj]
    IMUL    EBX, EBX, dim

    MOVAPS  XMM4, [EDX + EAX]
    MOVAPS  XMM5, [EDX + EAX + 16]
    MOVAPS  XMM6, [EDX + EAX + 32]
    MOVAPS  XMM7, [EDX + EAX + 48]

    ADDPS   XMM4, [epsilon]
    ADDPS   XMM5, [epsilon]
    ADDPS   XMM6, [epsilon]
    ADDPS   XMM7, [epsilon]

    SQRTPS  XMM4, XMM4
    SQRTPS  XMM5, XMM5
    SQRTPS  XMM6, XMM6
    SQRTPS  XMM7, XMM7

    MOVSS   XMM0, [rate]
    SHUFPS  XMM0, XMM0, 00000000

    MOVAPS  XMM1, XMM0
    MOVAPS  XMM2, XMM0
    MOVAPS  XMM3, XMM0

    DIVPS   XMM0, XMM4
    DIVPS   XMM1, XMM5
    DIVPS   XMM2, XMM6
    DIVPS   XMM3, XMM7

    MOVAPS  XMM4, [ECX + EAX]
    MOVAPS  XMM5, [ECX + EAX + 16]
    MOVAPS  XMM6, [ECX + EAX + 32]
    MOVAPS  XMM7, [ECX + EAX + 48]


    MULPS   XMM0, XMM4
    MULPS   XMM1, XMM5
    MULPS   XMM2, XMM6
    MULPS   XMM3, XMM7

    MOV EDX, [EBP + sommatoria]

    ADDPS   XMM0, [EDX + EBX]
    ADDPS   XMM1, [EDX + EBX + 16]
    ADDPS   XMM2, [EDX + EBX + 32]
    ADDPS   XMM3, [EDX + EBX + 48]

    MOVAPS  [EDX + EBX], XMM0
    MOVAPS  [EDX + EBX + 16], XMM1
    MOVAPS  [EDX + EBX + 32], XMM2
    MOVAPS  [EDX + EBX + 48], XMM3

    ;questo chiude forj3
    MOV EDX, 0
    MOV EAX, EBX
    MOV EBX, dim
    DIV     EBX
    MOV EBX, EAX

    ADD     EBX, 16
    MOV EAX, [indice]
    ADD     EAX, elementiunrolling
    CMP EBX, [lenght]
    JL      forj3
```

Forj3 scorre le coordinate (colonne) della corrente osservazione (riga), salva il contenuto di EAX ((p-i)*lenght*dim) in memoria di supporto: [indice].

Successivamente si sfrutta EDX per puntare a prima cella di Gj, e si moltiplica indice j (corrente colonna) per dim

NB
Questa procedura serve a risolvere la mancanza di registri, EBX verrà riutilizzato (dopo averlo diviso per dim, per contare il numero di colonne già scorse, poi verrà incrementato di 16 (per saltare gli indirizzi dei 16 elementi già letti) e moltiplicato nuovamente per dim all'inizio del ciclo

Spostiamo in XMM4,5,6,7 i 16 elementi di Gj del corrente passo di batch, a cui sommiamo indice EAX per accedere alla riga corretta

Aggiungiamo in parallelo epsilon ai valori di ogni colonna di Gj, successivamente ne calcoliamo la radice quadrata

Spostiamo [rate] (letto da input) nelle quattro posizioni di XMM0, e da XMM0 facciamo copia in parallelo su XMM1,2,3.

Dividiamo i 16 rate salvati per i 16 valori corrispondenti di Gj, successivamente leggiamo le 16 colonne corrispondenti di gj e le moltiplichiamo

A questo punto EBX viene riusato per puntare alla locazione di base di sommatoria, e implementare l'operazione sommatoria+= prodotto appena calcolato, sempre mantendo la corrispondenza fra colonne

Qui avviene la gestione di EBX descritta in precedenza

Equivalente in C

```c
for(int p=i; p<=i+v-1;p++)
    for(int j=0;j<lenght;j++){
        indice = ((p-i) * lenght) +j;
        sommatoria[j]+=(rate/sqrt(Gj[indice]+eps))*gj[indice];
}
```

```
forj4:
        MOVAPS  XMM2, [EAX + EDI]
        MOVAPS  XMM3, [EAX + EDI + 16]
        MOVAPS  XMM4, [EAX + EDI + 32]
        MOVAPS  XMM5, [EAX + EDI + 48]

        DIVPS   XMM2, [v]
        DIVPS   XMM3, [v]
        DIVPS   XMM4, [v]
        DIVPS   XMM5, [v]

        MOVAPS  XMM0, [EBX + EDI]
        MOVAPS  XMM1, [EBX + EDI + 16]
        MOVAPS  XMM6, [EBX + EDI + 32]
        MOVAPS  XMM7, [EBX + EDI + 48]

        SUBPS   XMM0, XMM2
        SUBPS   XMM1, XMM3
        SUBPS   XMM6, XMM4
        SUBPS   XMM7, XMM5

        MOVAPS  [EBX + EDI], XMM0
        MOVAPS  [EBX + EDI + 16], XMM1
        MOVAPS  [EBX + EDI + 32], XMM6
        MOVAPS  [EBX + EDI + 48], XMM7

        ;questo chiude il forj3
        ADD     ECX, 16
        ADD     EDI, elementiunrolling
        CMP ECX, [lenght]
        JL      forj4
```

Equivalente in C

```c
//sottraggo a theta
for(int j=0;j<lenght;j++){
    theta[j]=theta[j]-sommatoria[j]/v;
}
```

Versione OMP Parallel

Valgono le considerazioni compiute per la versione di batch parallelizzata, analizziamo di seguito il codice prodotto in merito.

```
void metodoOPM2(int p, float* theta, float* osservazioni, int lenght, float* gj, float* Gj, float* y, int i){
    adagrad32Prod(p,p+2,theta, osservazioni, lenght, gj, Gj,y,i);
}

void metodoOPM1(int p, float* theta, float* osservazioni, int lenght, float* gj, float* Gj, float* y, int i){
    adagrad32Prod(p,p+1,theta, osservazioni, lenght, gj, Gj,y,i);
}

void metodo2(int i, int v, float* theta, float*osservazioni, int lenght, float*gj, float* Gj, float*y){
    #pragma omp paraller for
    for(int p=i;p<=i+v-2;p+=2){
        metodoOPM2(p, theta, osservazioni, lenght, gj, Gj, y, i);
    }
}

void metodo1(int indice, int i, float* theta, float*osservazioni, int lenght, float*gj, float* Gj, float*y){
    if(indice%2==1)
        metodoOPM1(indice, theta, osservazioni, lenght, gj, Gj, y, i);
}
```

I metodi iniziali servono a gestire correttamente il parallelismo e evitare vengano condivise fra thread delle variabili in maniera impropria.

Il metodo AdagradOmp alloca le strutture dati allo stesso modo della propria versione non OMP, e successivamente presenta il seguente corpo

```
for(int it=0;it<iter;it++){
    for(i=0; i<n; i=i+k){

        int j;
        int v;
        if(n-i>k)
            v=k;
        else
            v=n-i;

        indice=i+v-1;

        metodo2(i, v, theta, osservazioni, lenght, gj, Gj, y);
        metodo1(indice, i, theta, osservazioni, lenght, gj, Gj, y);

        for(int j=0;j<lenght;j++)
            sommatoria[j]=0.0;

        int p=i;
        int pfin=i+v-1;
        aggiornamentoG(p, pfin, lenght, sommatoria, rate, Gj, gj);

        //sottraggo a theta
        for(j=0;j<lenght;j++){
            theta[j]=theta[j]-sommatoria[j]/v;
        }

    }
}
return theta;
```

Come avviene in batchOmp, vi è un meotod per analizzare con parallelismo tutte le osservazioni del corrente passo di batch (metodo2) e successivamente vi è un metodo per eseguire i calcoli necessari sulle osservazioni rimanenti;

difatti nel codice è specificato “quante righe” della matrice osservazioni gestisca ogni thread lanciato in parallelo, e bisogna gestire il caso in cui questo numero non sia sottomultiplo delle osservazioni per passo di batch, quindi rimangano osservazioni in più

Analizziamo il codice nasm di **adagrad32Prod**, questo metodo riceve: p indice di osservazione di partenza, pfin (nel nostro caso p+2) indice di osservazione finale (analizzerà osservazioni da p a pfin), l'indice i del corrente passo di batch e tutte le strutture dati definite già in precedenza.

```
forp:
        MOV     EDX, [p]

        XORPS   XMM1, XMM1
        XOR     ESI, ESI
        XOR     EDI, EDI
        MOV     ECX, EDX
        IMUL    ECX, [lenght]
        IMUL    ECX, ECX, dim

        MOV EBX, [theta]
        MOV EDX, [osservazioni]
```

forp scorre le osservazioni di interesse per il metodo, inizializzando per ciascuna di esse prodScal

```
forj1:
        MOVAPS  XMM2, [EBX + EDI]
        MULPS   XMM2, [EDX + ECX]
        ADDPS   XMM1, XMM2
        MOVAPS  XMM3, [EBX + EDI + 16]
        MULPS   XMM3, [EDX + ECX + 16]
        ADDPS   XMM1, XMM3
        MOVAPS  XMM4, [EBX + EDI + 32]
        MULPS   XMM4, [EDX + ECX + 32]
        ADDPS   XMM1, XMM4
        MOVAPS  XMM5, [EBX + EDI + 48]
        MULPS   XMM5, [EDX + ECX + 48]
        ADDPS   XMM1, XMM5

        ;questo chiude il forj1
        ADD     ESI, 16
        ADD     EDI, elementiunrolling
        ADD     ECX, elementiunrolling
        CMP ESI, [lenght]
        JL      forj1

        HADDPS  XMM1, XMM1
        HADDPS  XMM1, XMM1
```

Forj1 si occupa del calcolo di prodScal, come visto in precedenza

```
    MOV EBX, [y]
    MOV EDX, [p]

    XOR     EDI, EDI
    IMUL    EDI, EDX, dim
    SUBSS   XMM1, [EBX + EDI]
    SHUFPS  XMM1, XMM1, 00000000

    MOV     EDI, 0
    MOV     ESI, EDX
    SUB     ESI, [i]
    IMUL    ESI, [lenght]
    IMUL    ESI, ESI, dim

    IMUL    EDX, [lenght]
    IMUL    EDX, EDX, dim

    MOV     EAX, [gj]
    MOV EBX, [osservazioni]
    MOV ECX, [Gj]
```

Dopo aver calcolato prodScal si esegue la differenza con la coordinata di y corrispondente all'osservazione

```
forj2:
    MOVAPS  XMM0, XMM1
    MOVAPS  XMM2, XMM1
    MOVAPS  XMM4, XMM1
    MOVAPS  XMM6, XMM1

    MULPS   XMM0, [EBX + EDX]
    MULPS   XMM2, [EBX + EDX + 16]
    MULPS   XMM4, [EBX + EDX + 32]
    MULPS   XMM6, [EBX + EDX + 48]

    MOVAPS  [EAX + ESI], XMM0
    MOVAPS  [EAX + ESI + 16], XMM2
    MOVAPS  [EAX + ESI + 32], XMM4
    MOVAPS  [EAX + ESI + 48], XMM6

    MULPS   XMM0, XMM0
    MULPS   XMM2, XMM2
    MULPS   XMM4, XMM4
    MULPS   XMM6, XMM6

    ADDPS   XMM0, [ECX + ESI]
    ADDPS   XMM2, [ECX + ESI + 16]
    ADDPS   XMM4, [ECX + ESI + 32]
    ADDPS   XMM6, [ECX + ESI + 48]

    MOVAPS  [ECX + ESI], XMM0
    MOVAPS  [ECX + ESI + 16], XMM2
    MOVAPS  [ECX + ESI + 32], XMM4
    MOVAPS  [ECX + ESI + 48], XMM6
```

Forj2 aggiorna la singola riga di Gj, associata alla osservazione corrente

La parte mancante del codice C di riferimento è quella di aggiornamento di sommatoria, in funzione dei risultati prodotti.

Questa operazione avviene alla fine di ogni passo di batch, di conseguenza non si presta ad essere parallelizzata secondo la tecnica prescelta. Il codice nasm che compie l'aggiornamento di sommatoria è **aggiornamentoG** (segue codice)

```
forp2:
    MOV     EAX, [p]
    SUB     EAX, [i]
    IMUL    EAX, [lenght]
    IMUL    EAX, EAX, dim
    MOV [indice], EAX

    XOR     EBX, EBX
    MOV ECX, [gj]
```

Forp2 calcola l'indice di riga corrente, come offset da applicare all'indirizzo di partenza di Gj

```
forj3:
    MOV [indice], EAX
    MOV EDX, [Gj]
    IMUL    EBX, EBX, dim

    MOVAPS  XMM4, [EDX + EAX]
    MOVAPS  XMM5, [EDX + EAX + 16]
    MOVAPS  XMM6, [EDX + EAX + 32]
    MOVAPS  XMM7, [EDX + EAX + 48]

    ADDPS   XMM4, [epsilon]
    ADDPS   XMM5, [epsilon]
    ADDPS   XMM6, [epsilon]
    ADDPS   XMM7, [epsilon]

    SQRTPS  XMM4, XMM4
    SQRTPS  XMM5, XMM5
    SQRTPS  XMM6, XMM6
    SQRTPS  XMM7, XMM7

    MOVSS   XMM0, [rate]
    SHUFPS  XMM0, XMM0, 00000000

    MOVAPS  XMM1, XMM0
    MOVAPS  XMM2, XMM0
    MOVAPS  XMM3, XMM0

    DIVPS   XMM0, XMM4
    DIVPS   XMM1, XMM5
    DIVPS   XMM2, XMM6
    DIVPS   XMM3, XMM7

    MOVAPS  XMM4, [ECX + EAX]
    MOVAPS  XMM5, [ECX + EAX + 16]
    MOVAPS  XMM6, [ECX + EAX + 32]
    MOVAPS  XMM7, [ECX + EAX + 48]

    MULPS   XMM0, XMM4
    MULPS   XMM1, XMM5
    MULPS   XMM2, XMM6
    MULPS   XMM3, XMM7

    MOV EDX, [sommatoria]

    ADDPS   XMM0, [EDX + EBX]
    ADDPS   XMM1, [EDX + EBX + 16]
    ADDPS   XMM2, [EDX + EBX + 32]
    ADDPS   XMM3, [EDX + EBX + 48]

    MOVAPS  [EDX + EBX], XMM0
    MOVAPS  [EDX + EBX + 16], XMM1
    MOVAPS  [EDX + EBX + 32], XMM2
    MOVAPS  [EDX + EBX + 48], XMM3

;questo chiude forj3
MOV EDX, 0
MOV EAX, EBX
MOV EBX, dim
DIV     EBX
MOV EBX, EAX

ADD     EBX, 16
MOV EAX, [indice]
ADD     EAX, elementiunrolling
CMP EBX, [lenght]
JL      forj3

;questo chiude forp2
INC     ESI
MOV [p],ESI
CMP ESI, EDI
JL      forp2
```

SGD Batch (64)

L'implementazione dell'algoritmo scaturisce dallo studio approfondito della richiesta e da una prima realizzazione in codice C.

```
double* sgdBatch(params* input,double* osservazioni, int lenght){

    //start sgd
    double* theta=(double*)get_block(sizeof(double), lenght);
    int it=0;
    int i;
    double* y=input->y;
    int iter=input->iter;
    int n=input->n;
    double rate=input->eta;
    int k=input->k;

    for(int i=0;i<lenght;i++)
        theta[i]=0.0;

    for(int it = 0; it<iter; it++){
        double tmp[lenght];
        for(i=0; i<n; i+=k){

            for(int l=0;l<=lenght;l++)
                tmp[l]=0.0;

            int j;
            double prodScal;
            int v;
            if(n-i>k)
                v=k;
            else
                v=n-i;

            for(int p=i; p<i+v;p++){
                prodScal=0;

                //prodotto scalare
                for(j=0;j<lenght;j++)
                    prodScal+=theta[j]*osservazioni[p*lenght+j];


                //sottraggo yi
                prodScal=prodScal-y[p];

                //moltiplico per xi*
                for(j=0;j<lenght;j++){
                    tmp[j]+=osservazioni[p*lenght+j]*prodScal;
                }

            }

            //sottraggo a theta
            for(j=0;j<lenght;j++)
                theta[j]=theta[j]-tmp[j]*rate/v;

        }
    }
    return theta;
}
```

Come approfonditamente spiegato in precedenza, ritroviamo qui l'implementazione dell'algoritmo nella versione che considera i dati come double. In seguito l'inizializzazione delle strutture dati e delle variabili, un for che esegue il numero di iterazioni richieste. Il codice sfrutta un array, azzerato, ad ogni iterazione che conserva il valore calcolato dalla sommatoria nella formula matematica.

Il *forP* è necessario per calcolare il prodotto scalare tra theta e le corrispettive osservazioni nel corrente passo di batch al fine di far convergere la soluzione. Questo proviene dal fatto che ad ogni iterazione aggiorniamo il vettore theta con il valore che aveva all'iterazione precedente in modo da minimizzare l'errore quadratico medio.

L'aggiornamento sfrutta i valori η e $\frac{1}{v}$ moltiplicati al calcolo della sommatoria.

Ottimizzazioni in Assembly

La prima ipotesi di implementazione era quella di trasferire tutto il codice dal linguaggio C a quello assembly sfruttando le tecniche di loop unrolling e loop vectorization. Abbiamo introdotto infatti un fattore di unrolling pari a 4 che, in sinergia con il parallelismo SIMD, ci ha permesso di leggere 4 elementi alla volta e quindi 16 elementi per ogni iterazione replicando 4 volte ogni operazione.

Dal fattore 4 di unrolling nasce l'esigenza di dover effettuare padding su dimensione 16. Al fine di non sforare la dimensione effettiva del vettore andiamo a riempire le celle mancanti della struttura dati fino a raggiungere il multiplo di 16 più vicino: così facendo riusciremo a leggere sempre correttamente tutti gli elementi di riga senza incorrere in celle poco significative o occupate da altri valori.

```
int dimPadding(int dimensioneReale){
        int d=dimensioneReale/16;
        int r=dimensioneReale%16;
        if(r==0) return dimensioneReale;
        else return (d+1)*16;
}
```

Riportiamo quindi la versione C dell'algoritmo appena descritto per poi andare ad analizzare l'implementazione in assembly.

```
extern void batch64(params* input, double* osservazioni, double* theta, int* lenght);

double* sgdBatch(params* input, int lenght){

        //start sgd
        lenght=dimPadding(lenght);
        double* osservazioni=input->xast;
        double* theta=(double*)get_block(sizeof(double), lenght);
        int iter=input->iter;

        for(int i =0; i< lenght; i++){
                theta[i]=0.0;
        }

        int* len = (int*)get_block(sizeof(int), 1);

        len[0] = lenght;
        for(int it = 0; it<iter; it++){
                batch64(input, osservazioni, theta, len);
        }
        return theta;
}
```

Andiamo quindi ad analizzare più in dettaglio il codice del file nasm relativo al metodo batch64 con repertorio AVX.

```
batch64:
        ; ------------------------------------------------------------
        ; Sequenza di ingresso nella funzione
        ; ------------------------------------------------------------
        push    RBP                     ; salva il Base Pointer
        mov     RBP, RSP                ; il Base Pointer punta al Record di Attivazione corrente
        pushaq                          ; salva i registri generali

        ;PASSAGGIO DI LENGHT
        MOVSD           XMM0, [RCX]
        VEXTRACTPS      EAX, XMM0,0
        MOV             [lenghtB], EAX


        ;PASSAGGIO DI INPUT

        MOV     RAX, [RDI + 24]
        MOV     [nB], RAX

        MOV     RAX, [RDI + 32]
        MOV     [kB], RAX

        MOV     RAX, [RDI+8]
        MOV     [yB], RAX

        VMOVSD  XMM7, [RDI+40]
        VMOVSD   [rateB], XMM7

        ;PASSAGGIO DI THETA
        MOV     [thetaB], RDX

        MOV     [osservB], RSI

        XOR     RAX, RAX

        getmem  dim, [lenghtB]          ;RAX non lo utilizziamo perche sara utilizzato da getmem

        MOV     [indirizzotempB], RAX

        XOR     RBX, RBX                        ;RBX indice i del for dei passi di batch
foriB:

        MOV     [iB], EBX
        XOR     RSI, RSI                ;RSI indirizzo di partenza dei 4 float di interesse
        XOR     RDI, RDI                ;RDI numero di valori che scorriamo in un ciclo, (indice l del for di azzeramento di tmp)
        MOV     RAX, [indirizzotempB]
forlB:

        VXORPD  YMM0, YMM0              ;YMM0 ho tmp
        VMOVAPD [ RAX ], YMM0
        VMOVAPD [ RAX + 32], YMM0
        VMOVAPD [ RAX + 64], YMM0
        VMOVAPD [ RAX + 96], YMM0
        ADD     RAX, elementiunrolling
        ADD     EDI, 16
        CMP     EDI, [lenghtB]
        JL      forlB
```

Nel repertorio AVX la lettura dei parametri risulta essere differente rispetto alla versione 32 la quale faceva utilizzo del repertorio SSE. In questo caso infatti i primi sei parametri interi (scorrendo l'elenco dei parametri da sinistra verso destra) vengono passati, rispettivamente, nei registri RDI, RSI, RDX, RCX, R8 ed R9. Ulteriori parametri interi vengono passati sullo stack. Andiamo in questo caso ad allocare tramite funzione **getmem** il vettore tmp il cui puntatore sarà restituito in RAX. Iniziamo quindi il *foriB* nel quale utilizziamo l'indice i in RBX (che alla prima iterazione sarà pari a zero e salvato in memoria tramite variabile "iB") e, tramite il ciclo *forlB* andiamo ad azzerare l'intera struttura dati al fine di poter calcolare correttamente il nuovo valore della sommatoria relativo all'iterazione corrente. Andiamo ora a calcolare il valore v, upper bound della sommatoria che, come da formula matematica, va da "i" ad "i+v". una volta fatto ciò convertiamo il valore in floating point (in modo da eseguire correttamente il rapporto finale) e lo salviamo in apposita variabile "vB". Calcolo quindi la somma "i+v" e lo salvo in una variabile "limiteB" in modo da poter utilizzare quest'ultimo per i confronti finali e capire quando ho eseguito il numero corretto di iterazioni e terminare il *forpB*. Il codice relativo a forjB1 fa riferimento al presente codice C.

```
//prodotto scalare
for(j=0;j<lenght;j++)
        prodScal+=theta[j]*osservazioni[p*lenght+j];
```

```
        XOR     RCX, RCX
        ;-------------------
        ;CALCOLO DI V
        ;-------------------
        MOV     ECX, [nB]                   ;In ECX inserisco il valore v, contatore di elementi nel corrente passo di batch
        SUB     ECX, EBX
        CMP     ECX, [kB]
        JLE     saltoB

        MOV     ECX, [kB]
saltoB:
        VCVTSI2SD    XMM0, ECX              ;convertire integer to float
        MOVSD   [vB], XMM0

        MOV     EDX, EBX                    ;EDX indice p per scorrere qli elementi del corrente passo di batch faccio p=i
        MOV     ESI, EBX
        ADD     ESI, ECX                    ;In ESI ho inserito i+v per il forp
                                            ;edx ho p e in esi ho i+v condizione di inizio del forp

        MOV     [limiteB], ESI

forpB:
        VXORPD  YMM1, YMM1                  ;YMM1 contiene prodScal
        MOV     EDI, 0                      ;EDI indice j del for del prodotto scalare
        MOV     ESI, 0

        XOR     RCX, RCX

        MOV     ECX, EDX                    ;metto in ECX il valore di p
        IMUL    ECX, [lenghtB]              ;moltiplico per lenght
        IMUL    ECX, ECX, dim               ;IMUL ci salva p*lenght*dim


        MOV     [pB], EDX
        MOV     RBX,[thetaB]
        MOV     RDX,[osservB]


        MOV     [plenghtdimB], ECX
        ADD     RDX, [plenghtdimB]
forjB1:
        VMOVAPD YMM2, [RBX]
        VMULPD  YMM2, YMM2, [RDX]
        VADDPD  YMM1, YMM2
        VMOVAPD YMM3, [RBX + 32]
        VMULPD  YMM3, YMM3, [RDX + 32]
        VADDPD  YMM1, YMM3
        VMOVAPD YMM4, [RBX+ 64]
        VMULPD  YMM4, YMM4, [RDX + 64]
        VADDPD  YMM1, YMM4
        VMOVAPD YMM5, [RBX+ 96]
        VMULPD  YMM5, YMM5, [RDX + 96]
        VADDPD  YMM1, YMM5

        ;questo chiude il forj1
        ADD     ESI, 16
        ADD     RDX, elementiunrolling
        ADD     RBX, elementiunrolling
        CMP     ESI, [lenghtB]
        JL      forjB1
```

Effettuiamo ora il calcolo prodScal = prodScal – y[p] in modo da effettuare il calcolo relativo ad un'iterazione della sommatoria.

$$(\langle \Theta, \mathbf{x}_j^* \rangle - y_j)$$

Prepariamo adesso gli indici relativi al prossimo ciclo di for. Salvo il tutto in EDX *p*lenght*dim* ovvero indice di riga*numero elementi di riga* numero di byte per elemento. Abbiamo rispettato la proprietà di unrolling andando di volta in volta ad ogni iterazione, di ogni ciclo di for, ad aggiornare i puntatori per gli accessi in memoria sommando un fattore chiamato "*elementiunrolling*". In questo caso infatti sarà pari a 128 ovvero 16 double.

```
VHADDPD YMM1, YMM1
VHADDPD YMM1, YMM1                  ;nella prima posizione di YMM1 ho l'effettivo valore di prodScal

XOR     R9, R9
MOV     R9, [yB]

XOR     RDX, RDX
XOR     RBX, RBX

MOV     EDX, [pB]

IMUL    EDI, EDX, dim

MOV     [auxB], EDI
ADD     R9, [auxB]

VSUBSD  XMM1, [R9]          ;in YMM1 ho prodScal = prodScal - y[p]
VMOVSD  [prodScalB], XMM1

MOV     EDI, 0                      ;EDI indice j del for del prodotto scalare
IMUL    EDX, [lenghtB]              ;posso riutilizzare EDX tanto il valore dell'indice p e salvato nella variabile p
IMUL    EDX, EDX, dim               ;IMUL ci salva p*lenght*dim

MOV     RBX, [osservB]
ADD     RBX, RDX


MOV     RAX, [indirizzotempB]
```

Nel forjB2 andiamo ad effettuare l'aggiornamento di tmp, ovvero salviamo quindi il calcolo effettuato dalla sommatoria. Il riferimento al codice C di tale operazione è:

```
//moltiplico per xi*
for(j=0;j<lenght;j++){
    tmp[j]+=osservazioni[p*lenght+j]*prodScal;
}
```

```
forjB2:
        VBROADCASTSD    YMM0, [prodScalB]
        VMULPD          YMM0, [RBX]

        VBROADCASTSD    YMM2, [prodScalB]
        VMULPD          YMM2, [RBX + 32]

        VBROADCASTSD    YMM4, [prodScalB]
        VMULPD          YMM4, [RBX + 64]

        VBROADCASTSD    YMM6, [prodScalB]
        VMULPD          YMM6, [RBX +  96]

        VADDPD  YMM0, [RAX]
        VADDPD  YMM2, [RAX + 32]
        VADDPD  YMM4, [RAX + 64]
        VADDPD  YMM6, [RAX + 96]

        VMOVAPD [RAX], YMM0         ;salvataggio in tmp dei valori calcolati, ovvero tutto ciò che sta a destra della sommatoria
        VMOVAPD [RAX + 32], YMM2
        VMOVAPD [RAX + 64], YMM4
        VMOVAPD [RAX + 96], YMM6

        ;questo chiude il forj2
        ADD     EDI, 16
        ADD     RAX, elementiunrolling
        ADD     RBX, elementiunrolling
        CMP     EDI, [lenghtB]
        JL      forjB2

        XOR     RBX, RBX

        ; questo chiude il forp
        XOR     RDX, RDX
        MOV     EDX, [pB]
        MOV     ESI, [limiteB]
        ADD     EDX, 1
        CMP     EDX, ESI
        JL      forpB


        VMOVSD          XMM0, [rateB]
        VMOVSD          XMM1,[vB]
        VDIVSD          XMM0, XMM1                  ;questa è la variabile rapporto rate/v

        VMOVSD          [rapportoB], XMM0
        VBROADCASTSD    YMM0, [rapportoB]
        VMOVAPD         [rapportoB], YMM0
```

Tramite un'operazione di VBROADCASTSD ricopiamo il valore di prodScal, calcolato in precedenza, su tutte le celle di un registro YMM: in questo modo riusciremo ad effettuare le operazioni di moltiplicazione e somma su 4 double in parallelo. Dopo aver correttamente aggiornato il vettore tmp effettuiamo il calcolo del *rapporto* ovvero $\frac{rate}{v}$.

```
        MOV     ECX, 0


        MOV     RBX, [thetaB]
        MOV     RAX, [indirizzotempB]
forjB3:
        VMOVAPD     YMM2, [RAX]
        VMOVAPD     YMM3, [RAX + 32]
        VMOVAPD     YMM4, [RAX + 64]
        VMOVAPD     YMM5, [RAX + 96]

        VMULPD      YMM2, YMM2, [rapportoB]
        VMULPD      YMM3, YMM3, [rapportoB]
        VMULPD      YMM4, YMM4, [rapportoB]
        VMULPD      YMM5, YMM5, [rapportoB]

        VMOVAPD     YMM0, [RBX]
        VMOVAPD     YMM1, [RBX + 32]
        VMOVAPD     YMM6, [RBX + 64]
        VMOVAPD     YMM7, [RBX + 96]

        VSUBPD      YMM0, YMM2                  ;salvataggio in theta dei valori calcolati
        VSUBPD      YMM1, YMM3
        VSUBPD      YMM6, YMM4
        VSUBPD      YMM7, YMM5

        VMOVAPD     [RBX], YMM0
        VMOVAPD     [RBX + 32], YMM1
        VMOVAPD     [RBX + 64], YMM6
        VMOVAPD     [RBX + 96], YMM7

        ;questo chiude il forj3
        ADD     ECX, 16
        ADD     RAX, elementiunrolling
        ADD     RBX, elementiunrolling
        CMP     ECX, [lenghtB]
        JL      forjB3

        ;ripristino gli indici v e i in modo da trovarli aggiornati al prossimo for
        XOR     RBX, RBX
        MOV     EBX, [iB]

        ; questo chiude il fori
        ADD     EBX, [kB]
        CMP     EBX, [nB]
        JL      foriB
        ; ------------------------------------------------
        ; Sequenza di uscita dalla funzione
        ; ------------------------------------------------

        popaq                       ; ripristina i registri generali
        mov     rsp, rbp            ; ripristina lo Stack Pointer
        pop     rbp                 ; ripristina il Base Pointer
        ret                         ; torna alla funzione C chiamante
```

Infine andiamo ad aggiornare il valore del vettore theta con i nuovi valori calcolati. Il corrispondente frammento di codice C che replica tali istruzioni è:

```
//sottraggo a theta
for(i=0;i<lenght;i++)
    theta[i]=theta[i]-tmp[i]*rate/v;
```

PERCHE' NON FARE CACHING?

Dopo aver analizzato il problema e averlo interpretato, per fare caching avremmo potuto sfruttare il fatto che in un singolo passo di batch, il theta letto rimane invariato, scorrendo le varie osservazioni.

Per quanto riguarda le osservazioni, di esse non viene "riutilizzata" alcuna componente, ogni colonna di osservazione viene usata una volta, successivamente bisogna calcolare prodScal (che necessita lo scorrimento di TUTTE le coordinate dell'osservazione per essere corretta) e poi viene riutilizzata, quindi non si può fare caching sulle colonne di osservazioni.

Fare Caching sulle righe di osservazioni è altrettanto inutile, perché ogni osservazione differisce dalle altre, si potrebbe però pensare a fare caching su theta, in quanto ogni osservazione (di uno stesso passo di batch) accederà allo stesso theta.
Per fare ciò dovremmo selezionare un frammento delle colonne di osservazioni (per cui theta rimane fissato) e scorrere tutte le righe di osservazioni (non per intero, ma solo nell'intervallo del frammento selezionato).

Questa soluzione non è applicabile, in quanto il primo calcolo che bisogna fare per una determinata osservazione è quello di prodScal, che per essere calcolato (prima di sottrarre y[p]) deve aver cumulato il contributo di **tutte** le colonne della osservazione corrente, non possiamo quindi utilizzare in maniera utile "frammenti" di riga, per favorire caching.

Versione OMP Parallel

Grazie al parallelismo utilizzato nella versione OpenMP, è possibile individuare l'operazione più critica, che viene quindi migliorata con la seguente tecnica: l'iterazione sulle osservazioni del corrente passo di batch. È fondamentale individuare i componenti che non necessitano di eseguire in ordine, poiché si devono parallelizzare frammenti di codice che possono interfogliarsi senza causare errori nello stesso. Non si possono parallelizzare iterazioni differenti, poiché ogni interazione deve avvenire in funzione dei dati calcolati nella versione precedente. Inoltre, non si possono parallelizzare i passi di batch, perché ognuno deve avvenire in ordine, al fine di incrementare correttamente theta, su cui si baserà il passo di batch successivo. L'esecuzione delle iterazioni di *forP*, il ciclo che scorre le osservazioni del corrente passo di batch, è la componente di codice parallelizzabile. Ogni osservazione di fatti favorisce il calcolo di prodScal, ma quest'ultimo deve essere incrementato (n modo cumulativo, somma è ovviamente commutativa). Non è necessario (come operando) in nessuna operazione che l'osservazione successiva deve compiere.

Ispirandoci al meccanismo di gestione dei risultati delle singole osservazioni che abbiamo visto in Adagrad, realizziamo tmp come matrice, con numero di righe (pari al numero di osservazioni che scriveremo), dato dal passo di batch. Così facendo si esclude l'unico errore che il parallelismo potrebbe comportare (fatta eccezione di conflitti logici, date, operazioni indipendenti): l'accesso contemporaneo di più thread ad uno stesso indirizzo, che avverrebbe se tmp fosse array. Ciascun thread calcola i risultati parziali del corrente passo di batch e memorizza il risultato di ogni osservazione nella riga di tmp; in seguito si sommano i contributi delle osservazioni, sommando tutte le righe di tmp.

```
double* sqdBatch(params* input, int lenght){

    //start sqd
    double* osservazioni=input->xast;
    lenght=dimPadding(lenght);
    double* theta=(double*)get_block(sizeof(double), lenght);
    int it=0;
    int i;
    double* y=input->y;
    int iter=input->iter;
    int n=input->n;
    double rate=input->eta;
    int k=input->k;

    double* tmp=(double*)get_block(sizeof(double),k*lenght);

    for(int i=0;i<lenght;i++){
        theta[i]=0.0;
    }
    while(it<iter){

        for(i=0; i<n; i+=k){ //FOR dei passi di batch

            for(int l=0;l<k*lenght;l++)
                tmp[l]=0.0;

            int j;
            int v;
            if(n-i>k)
                v=k;
            else
                v=n-i;
            indice=i+v-1;

            Bmetodo2(i, v, theta, osservazioni, lenght, tmp, y);
            Bmetodo1(indice, i, v, theta, osservazioni, lenght, tmp, y);

            for(int j=0;j<lenght;j++){
                for(int i=1;i<v;i++){
                    tmp[j]+=tmp[i*lenght+j];
                }
            }

            double rapporto = rate/v;

            //sottraggo a theta
            for(j=0;j<lenght;j++){
                theta[j]=theta[j]-tmp[j]*rapporto;
            }
        }
        it++;
    }
    return theta;
}
```

Il metodo Bmetodo2, avvia la computazione parallela delle osservazioni del corrente passo di batch.

In funzione del grado di parallelismo introdotto (numero di osservazioni gestite da ogni thread), si determinano le osservazioni restanti (non gestite da thread) nel caso in cui la dimensione di batch non fosse multiplo esatto delle osservazioni analizzate dai singoli elementi

Si cumula sulla prima riga di tmp la sommatoria dei contributi di tutte le altre righe, infine si sottrae tmp*rapporto a theta, come nella versione non parallela.

NB: Come si può notare dai codici che seguono, è stata scelto (in base alle prove condotte) un numero di osservazioni per thread pari a 2. Questo parametro può essere aumentato, facendo in modo che un singolo thread si dedichi a più osservazioni, senza perdita di generalità nel ragionamento e senza intaccare il funzionamento del codice.
Ovviamente decrementando il numero di osservazioni per thread si aumenta l'efficienza, idealmente si arriva a far si che ogni riga del passo di batch sia eseguita in parallelo, questa scelta non è nella pratica applicabile, non avendo un numero infinito di thread lanciabili in parallelo e avendo passi di batch con molte osservazioni.

I metodi che seguono sono necessari per applicare correttamente la direttiva di parallelismo pragma, e bisogna fare in modo che la direttiva sia inserita in cima a un for privo di for innestati, per evitare che agisca su altri for e parallelizzi in profondità.

```
extern void batch64Prod(int p, int pfin, int i, double* osservazioni, int lenght, double* y, double* tmp, double* theta);

void BmetodoOPM2(int p, double* theta, double* osservazioni, int lenght, double* tmp, double* y, int i){
	batch64Prod(p,p+2,i, osservazioni, lenght,y,tmp,theta);
}

void BmetodoOPM1(int p, double* theta, double* osservazioni, int lenght, double* tmp, double* y, int i){
	batch64Prod(p,p+1,i, osservazioni, lenght, y, tmp, theta);
}

void Bmetodo2(int i, int v, double* theta, double*osservazioni, int lenght, double*tmp, double*y){
	#pragma omp paraller for
	for(int p=i;p<=i+v-2;p+=2){
		BmetodoOPM2(p, theta, osservazioni, lenght, tmp, y, i);
	}
}

void Bmetodo1(int indice, int i, int v, double* theta, double*osservazioni, int lenght, double*tmp, double*y){
	if(indice%2==1)
		BmetodoOPM1(indice, theta, osservazioni, lenght, tmp, y, i);
}
```

Bmetodo2 lancia il parallelismo sulle iterazioni del corrente passo di batch, incrementando gli indici di riga di 2 elementi alla volta, di modo che ogni thread si riferisca a due osservazioni del corrente passo di batch. Richiama al suo interno batch64, metodo appositamente modificato per interessarsi delle osservazioni che vanno dall'indice p all'indice pfin (escluso).

Bmetodo1 non sfrutta invece il parallelismo, ma gestisce un'eventuale osservazione rimasta, nel caso in cui il passo di batch dovesse essere dispari (quindi non divisibile esattamente per 2)

È di seguito riportato il metodo batch64Prod scritto in assembly e richiamato dal codice C appena analizzato. I è necessario perchè matrice tmp ha numero di righe pari alla dimensione di batch, se vi accedessimo in base a p, cercheremmo una riga non presente (nel DB di prova ad esempio p arriva a 2000, ma con batch 20 ho tmp di venti righe, quindi solo prendendo righe date da p-i ottengo sempre un valore valido fra 0 e 19)

Dopo una corretta lettura dei parametri effettuiamo le operazioni del tutto speculari alla versione senza parallelizzazione. Andiamo infatti a calcolare il prodotto scalare della sommatoria.

```
batch64Prod:
        ; ------------------------------------------------
        ; Sequenza di ingresso nella funzione
        ; ------------------------------------------------
        push    RBP                 ; salva il Base Pointer
        mov     RBP, RSP            ; il Base Pointer punta al Record di Attivazione corrente
        pushaq                      ; salva i registri generali


        MOV     [p], RDI

        MOV     [pfin], RSI

        MOV     [i], RDX

        MOV     [osservazioni], RCX

        MOV     [lenght], R8

        MOV     [y], R9

        MOV     RAX, [RBP + 16]
        MOV     [tmp], RAX

        MOV     RAX, [RBP + 24]
        MOV     [theta], RAX
forpB:
        MOV     RDX, [p]

        VXORPD  YMM1, YMM1          ;YMM1 contiene prodScal
        XOR     RDI, RDI            ;RDI indice j del for del prodotto scalare
        XOR     RSI, RSI
        MOV     RCX, RDX            ;metto in RCX il valore di p
        IMUL    RCX, [lenght]       ;moltiplico per lenght
        IMUL    RCX, RCX, dim               ;IMUL ci salva p*lenght*dimB

        MOV     RBX,[theta]
        MOV     RDX,[osservazioni]
forj1B:
        VMOVAPD YMM2, [RBX + RDI]
        VMULPD  YMM2, YMM2, [RDX + RCX]
        VADDPD  YMM1, YMM1, YMM2


        VMOVAPD YMM3, [RBX + RDI + 32]
        VMULPD  YMM3, YMM3, [RDX + RCX + 32]
        VADDPD  YMM1, YMM1, YMM3


        VMOVAPD YMM4, [RBX + RDI + 64]
        VMULPD  YMM4, YMM4, [RDX + RCX + 64]
        VADDPD  YMM1, YMM1, YMM4


        VMOVAPD YMM5, [RBX + RDI + 96]
        VMULPD  YMM5, YMM5, [RDX + RCX + 96]
        VADDPD  YMM1, YMM1, YMM5

        ;questo chiude il forj1
        ADD     RSI, 16
        ADD     RDI, elementiunrolling
        ADD     RCX, elementiunrolling
        CMP     RSI, [lenght]
        JL      forj1B
```

```
        VHADDPD YMM1, YMM1, YMM1
        VHADDPD YMM1, YMM1, YMM1                    ;nella prima posizione di XMM1 ho l'effettivo valore di prodScal

        MOV     RBX, [y]
        MOV     RDX, [p]

        XOR     RDI, RDI
        IMUL    RDI, RDX, dim

        VSUBSD  XMM1, XMM1, [RBX + RDI]         ;in XMM1 ho prodScal = prodScal - y[p]
        VMOVSD  [auxxxxx], XMM1
        VBROADCASTSD    YMM1, [auxxxxx]


        XOR     RDI,RDI                     ;EDI indice j del for del prodotto scalare
        MOV     RSI, RDX
        SUB     RSI, [i]
        IMUL    RSI, [lenght]

        IMUL    RSI, RSI, dim               ;calcolo indice = p-i * lenght * dim


        IMUL    RDX, [lenght]               ;posso riutilizzare EDX tanto il valore dell'indice p e salvato nella variabile p
        IMUL    RDX, RDX, dim               ;IMUL ci salva p*lenght*dim in EDX

        MOV     RAX, [tmp]
        MOV     RBX, [osservazioni]

forj2B:
        VMOVAPD YMM0, YMM1
        VMOVAPD YMM2, YMM1
        VMOVAPD YMM4, YMM1
        VMOVAPD YMM6, YMM1

        VMULPD  YMM0, YMM0, [RBX + RDX]
        VMULPD  YMM2, YMM2, [RBX + RDX + 32]
        VMULPD  YMM4, YMM4, [RBX + RDX + 64]
        VMULPD  YMM6, YMM6, [RBX + RDX + 96]

        VMOVAPD       [RAX + RSI], XMM0         ;salvataggio in tmp dei valori calcolati, ovvero tutto ciò che sta a destra della sommat
        VEXTRACTF128 XMM0,YMM0,1
        VMOVAPD       [RAX + RSI + 16], XMM0

        VMOVAPD       [RAX + RSI + 32], XMM2
        VEXTRACTF128 XMM2,YMM2,1
        VMOVAPD       [RAX + RSI + 48], XMM2

        VMOVAPD       [RAX + RSI + 64], XMM4
        VEXTRACTF128 XMM4,YMM4,1
        VMOVAPD       [RAX + RSI + 80], XMM4

        VMOVAPD       [RAX + RSI + 96], XMM6
        VEXTRACTF128 XMM6,YMM6,1
        VMOVAPD       [RAX + RSI + 112], XMM6

        ;questo chiude il forj2
        ADD     RDI, 16
        ADD     RSI, elementiunrolling
        ADD     RDX, elementiunrolling
        CMP     RDI, [lenght]
        JL      forj2B

        ; questo chiude il forp
        MOV     RDX, [p]
        INC     RDX
        MOV     [p], RDX
        MOV     RSI, [pfin]
        CMP     RDX, RSI
```

Il resto delle operazioni sfrutta le stesse meccaniche descritte nella versione non parallelizzata, con la sostanziale differenza che il nuovo estremo superiore del ciclo *forpB* è adesso ovviamente pFin.

SGD Adagrad (64)

L'implementazione dell'algoritmo scaturisce dallo studio approfondito della richiesta e da una prima realizzazione in codice C.

```
double* sgdAdagrad(params* input, double* osservazioni, int lenght){

    //start sgd
    double* theta=(double*)get_block(sizeof(double), lenght);
    int it=0;
    int i;
    double* y=input->y;
    int iter=input->iter;
    int n=input->n;
    double rate=input->eta;
    int k=input->k;
    double eps=1E-8;

    double Gj[k * lenght];
    double qj[k * lenght];
    double sommatoria[lenght];

    for(int p1=0;p1<k;p1++)
        for(int p2=0;p2<lenght;p2++){
            Gj[p1 * lenght + p2]=0.0;
            qj[p1 * lenght + p2]=0.0;
        }

    for(int it = 0; it<iter; it++){

        for(i=0; i<1; i=i+k){

            int j;
            double prodScal;
            int v;
            if(n-i>k)
                v=k;
            else
                v=n-i;

            for(int p=i; p<i+v;p++){
                prodScal=0;

                //prodotto scalare
                for(j=0;j<lenght;j++){
                    prodScal+=theta[j]*osservazioni[p*lenght+j];
                }

                //sottraggo yi
                prodScal=prodScal-y[p];

                //moltiplico per xi*
                for(j=0;j<lenght;j++){
                    indice = ((p-i) * lenght) +j;
                    qj[indice]=osservazioni[p*lenght + j]*prodScal;
                    Gj[indice]=Gj[indice]+qj[indice]*qj[indice];
                }
            }

            for(int j=0;j<lenght;j++)
                sommatoria[j]=0.0;

            for(int p=i; p<i+v;p++)
                for(int j=0;j<lenght;j++){
                    indice = ((p-i) * lenght) +j;
                    sommatoria[j]+=(rate/sqrt(Gj[indice]+eps))*qj[indice];
                }
            //sottraggo a theta
            for(j=0;j<lenght;j++){
                theta[j]=theta[j]-sommatoria[j]/v;
            }
        }
    }
    return theta;
}
```

Creiamo dapprima le strutture dati necessarie e nelle righe successive, viene descritto il calcolo effettuato per ogni singola iterazione.

Analogamente alla versione batch, troviamo un for di scorrimento delle iterazioni: al suo interno c'è un for che scorre i diversi passi di batch e infine, per ogni passo di batch, il for che ne scorre le osservazioni.

Per ogni osservazione, viene eseguito prodScal cambiando, contrariamente a quanto fatto in batch, l'aggiornamento delle strutture dati.

Sono presenti due matrici (linearizzate): **gj** e **Gj**, con il compito di rappresentare in ogni loro riga i risultati di una osservazione del corrente passo di batch. **gj** è l'esatta trasposizione di tmp (nel codice batch), cumula i risultati di una singola osservazione (del corrente passo di batch) e poi viene azzerata al successivo passo.

Gj è invece cumulativa, ogni sua riga contiene il contributo di tutte le osservazioni (di passi di batch precedenti, ma anche iterazioni precedenti) che corrispondevano alla riga stessa. Quanto descritto è evidente dalla descrizione dell'algoritmo fornita alla consegna del problema.

Ottimizzazioni in Assembly

Le ottimizzazioni compiute, la scelta delle strutture dati e delle tecniche di velocizzazione sono del tutto analoghe a quelle compiute per batch, sono altrettanto condivise le motivazioni del mancato caching.

Osserviamo quindi il codice assembly, concentrandoci sulle sole differenze sostanziali con batch.

```
    MOV     RAX, [qj]
    MOV     RBX, [osserv]
    MOV     RCX, [Gj]

    ADD     RBX, RDX
    ADD     RAX, RSI
    ADD     RCX, RSI
forj2:

    VMOVAPD YMM0, YMM1      ;in a
    VMOVAPD YMM2, YMM1
    VMOVAPD YMM4, YMM1
    VMOVAPD YMM6, YMM1

    VMULPD  YMM0, [RBX]
    VMULPD  YMM2, [RBX + 32]
    VMULPD  YMM4, [RBX + 64]
    VMULPD  YMM6, [RBX + 96]

    VMOVAPD [RAX], YMM0     ;sal
    VMOVAPD [RAX + 32], YMM2
    VMOVAPD [RAX + 64], YMM4
    VMOVAPD [RAX + 96], YMM6

    VMULPD  YMM0, YMM0
    VMULPD  YMM2, YMM2
    VMULPD  YMM4, YMM4
    VMULPD  YMM6, YMM6

    VADDPD  YMM0, [RCX]
    VADDPD  YMM2, [RCX + 32]
    VADDPD  YMM4, [RCX + 64]
    VADDPD  YMM6, [RCX + 96]

    VMOVAPD [RCX ], YMM0    ;sal
    VMOVAPD [RCX + 32], YMM2
    VMOVAPD [RCX + 64], YMM4
    VMOVAPD [RCX + 96], YMM6

    ;questo chiude il forj2
    ADD     EDI, 16
    ADD     RAX, elementiunrolling
    ADD     RBX, elementiunrolling
    ADD     RCX, elementiunrolling
    CMP     EDI, [lenght]
    JL      forj2
```

Dopo aver calcolato prodScal per ogni osservazione, vengono aggiornate le strutture dati Gj e gj, secondo le formule matematiche proprie dell'algoritmo.

Il calcolo di gj è lo stesso della struttura dati tmp (in batch), mentre Gj ha in ogni sua riga il quadrato della corrispondente riga in gj, sommato al valore precedente di Gj stesso.

Si noti come l'accesso alle matrici non debba dipendere dall'indice di riga "assoluto" della osservazione corrente (che va da 0 a n) ma dal suo indice relativo nel corrente passo di batch (che va da 0 a passo di batch, escluso), maniera simile a quanto avviene nella versione OMP.

```
    ; questo chiude il forp
    XOR     RDX, RDX
    XOR     RSI, RSI
    MOV     EDX, [p]
    MOV     ESI, [limite]
    ADD     EDX, 1
    CMP     EDX, ESI
    JL      forp

    XOR     RSI, RSI                ;ESI indirizzo di partenza dei 4 float di interesse
    XOR     RDI, RDI                ;EDI numero di valori che scorriamo in un ciclo, (indice l del for di azzeramento di tmp)
    MOV     RAX, [sommatoria]
forl:

    VXORPD  YMM0, YMM0              ;XMM0 ho sommatoria
    VMOVAPD [ RAX ], YMM0
    VMOVAPD [ RAX + 32], YMM0
    VMOVAPD [ RAX + 64], YMM0
    VMOVAPD [ RAX + 96], YMM0
    ADD     RAX, elementiunrolling
    ADD     EDI, 16
    CMP     EDI, [lenght]
    JL      forl

    XOR     RSI, RSI
    MOV     ESI, [i]
    MOV     [p], ESI
forp2:
    XOR     RAX, RAX
    MOV     EAX, [p]
    SUB     EAX, [i]
    IMUL    EAX, [lenght]
    IMUL    EAX, EAX, dim           ;calcolo indice = p-i * lenght * dim e lo metto in EAX

    MOV     [indice], EAX

    MOV     RCX, [gj]
    MOV     RDX, [Gj]

    ADD     RDX, [indice]
    ADD     RCX, [indice]

    MOV     RAX, [sommatoria]

    XOR     RSI, RSI
forj3:

    VMOVAPD YMM4, [RDX]             ;qui abbiamo Gj
    VMOVAPD YMM5, [RDX + 32]
    VMOVAPD YMM6, [RDX + 64]
    VMOVAPD YMM7, [RDX + 96]

    VADDPD  YMM4, [epsilon]
    VADDPD  YMM5, [epsilon]
    VADDPD  YMM6, [epsilon]
    VADDPD  YMM7, [epsilon]
```

Forj3 esegue la formula che troviamo alla destra della sommatoria, quella che ci consentirà

$$\Theta := \Theta - \frac{1}{\nu} \sum_{j=i}^{i+\nu-1} \frac{\eta}{\sqrt{G_j + \epsilon}} g_j$$

successivamente di aggiornare correttamente theta. Per fare ciò leggiamo sempre 16 elementi per volta di **Gj** e li sommiamo ad una quantità *epsilon*. Una volta fatto ciò eseguiamo, seguendo la formula matematica, la radice quadrata e la divisione di quest'ultima con eta. A fine *forj3* moltiplichiamo questo rapporto per **gj** e salviamo tutto all'interno della struttura dati **sommatoria**.

```
VSQRTPD  YMM4, YMM4
VSQRTPD  YMM5, YMM5
VSQRTPD  YMM6, YMM6
VSQRTPD  YMM7, YMM7

VBROADCASTSD YMM0, [rate]

VMOVAPD      YMM1, YMM0
VMOVAPD      YMM2, YMM0
VMOVAPD      YMM3, YMM0

VDIVPD   YMM0, YMM4
VDIVPD   YMM1, YMM5
VDIVPD   YMM2, YMM6
VDIVPD   YMM3, YMM7

VMOVAPD YMM4, [RCX]          ;qui abbiamo qj
VMOVAPD YMM5, [RCX + 32]
VMOVAPD YMM6, [RCX + 64]
VMOVAPD YMM7, [RCX + 96]


VMULPD   YMM0, YMM4
VMULPD   YMM1, YMM5
VMULPD   YMM2, YMM6
VMULPD   YMM3, YMM7


VADDPD   YMM0, [RAX]         ;qui abbiamo sommatoria
VADDPD   YMM1, [RAX + 32]
VADDPD   YMM2, [RAX + 64]
VADDPD   YMM3, [RAX + 96]

VMOVAPD [RAX], YMM0
VMOVAPD [RAX + 32], YMM1
VMOVAPD [RAX + 64], YMM2
VMOVAPD [RAX + 96], YMM3

;questo chiude forj3
ADD      ESI, 16
ADD      RAX, elementiunrolling
ADD      RCX, elementiunrolling
ADD      RDX, elementiunrolling
CMP      ESI, [lenght]
JL       forj3

;questo chiude forp2
XOR      RDI, RDI
MOV      EDI, [limite]
XOR      RSI, RSI
MOV      ESI, [p]
ADD      ESI, 1
MOV      [p],ESI
CMP      ESI, EDI
JL       forp2
```

Ultimo ciclo ci consente di aggiornare correttamente il valore di **theta** sottraendo al valore calcolato nell'iterazione precedente, ciò che abbiamo finora calcolato e salvato nella struttura dati **sommatoria**.

```
        MOV     RBX, [theta]
        MOV     RAX, [sommatoria]
forj4:
        VMOVAPD YMM2, [RAX]
        VMOVAPD YMM3, [RAX + 32]
        VMOVAPD YMM4, [RAX + 64]
        VMOVAPD YMM5, [RAX + 96]

        VDIVPD  YMM2, [v]
        VDIVPD  YMM3, [v]
        VDIVPD  YMM4, [v]
        VDIVPD  YMM5, [v]

        VMOVAPD  YMM0, [RBX]
        VMOVAPD  YMM1, [RBX + 32]
        VMOVAPD  YMM6, [RBX + 64]
        VMOVAPD  YMM7, [RBX + 96]

        VSUBPD  YMM0, YMM2                      ;salvataggio in theta dei valori calcolati
        VSUBPD  YMM1, YMM3
        VSUBPD  YMM6, YMM4
        VSUBPD  YMM7, YMM5

        VMOVAPD  [RBX], YMM0
        VMOVAPD  [RBX + 32], YMM1
        VMOVAPD  [RBX + 64], YMM6
        VMOVAPD  [RBX + 96], YMM7

        ;questo chiude il forj4
        ADD     ECX, 16
        ADD     RAX, elementiunrolling
        ADD     RBX, elementiunrolling
        CMP     ECX, [lenght]
        JL      forj4

        ;ripristino gli indici v e i in modo da trovarli aggiornati al prossimo for
        XOR     RBX, RBX
        MOV     EBX, [i]

        ; questo chiude il fori
        ADD     EBX, [k]
        CMP     EBX, [n]
        JL      fori

        ; ------------------------------------------------------------
        ; Sequenza di uscita dalla funzione
        ; ------------------------------------------------------------
        popaq                                   ; ripristina i registri generali
        mov     rsp, rbp                        ; ripristina lo Stack Pointer
        pop     rbp                             ; ripristina il Base Pointer
        ret                                     ; torna alla funzione C chiamante
```

Versione OMP Parallel

Valgono le considerazioni compiute per la versione di batch parallelizzata, analizziamo di seguito il codice prodotto in merito.

```
extern void adagrad64Prod(int p, int pfin, double* theta, double* osservazioni, int lenght, double* gj, double* Gj, double* y, int i);
extern void aggiornamentoG(int p, int pfin, int lenght, double* sommatoria, double rate, double* Gj, double* gj);

void metodoOPM2(int p, double* theta, double* osservazioni, int lenght, double* gj, double* Gj, double* y, int i){
	//adagrad64Prod(p,p+2,theta, osservazioni, lenght, gj, Gj,y,i);
	for(int x = p; x<p+2;x++){
		double prodScal=0;
		//prodotto scalare
		for(int j=0;j<lenght;j++){
			prodScal+=theta[j]*osservazioni[x*lenght+j];
		}
		//sottraggo yi
		prodScal=prodScal-y[x];

		//moltiplico per xi*
		for(int j=0;j<lenght;j++){
			indice=(x-i)*lenght+j;
			gj[indice]=osservazioni[x*lenght+j]*prodScal;
			Gj[indice]+=gj[indice]*gj[indice];
		}
	}
}

void metodoOPM1(int p, double* theta, double* osservazioni, int lenght, double* gj, double* Gj, double* y, int i){
	//adagrad64Prod(p,p+1,theta, osservazioni, lenght, gj, Gj,y,i);
	double prodScal=0;
	//prodotto scalare
	for(int j=0;j<lenght;j++){
		prodScal+=theta[j]*osservazioni[p*lenght+j];
	}
	//sottraggo yi
	prodScal=prodScal-y[p];

	//moltiplico per xi*
	for(int j=0;j<lenght;j++){
		indice=(p-i)*lenght+j;
		gj[indice]=osservazioni[p*lenght+j]*prodScal;
		Gj[indice]+=gj[indice]*gj[indice];
	}
}

void metodo2(int i, int v, double* theta, double*osservazioni, int lenght, double*gj, double* Gj, double*y){
	#pragma omp paraller for
	for(int p=i;p<=i+v-2;p+=2){
		metodoOPM2(p, theta, osservazioni, lenght, gj, Gj, y, i);
	}
}

void metodo1(int indice, int i, double* theta, double*osservazioni, int lenght, double*gj, double* Gj, double*y){
	metodoOPM1(indice, theta, osservazioni, lenght, gj, Gj, y, i);
}
```

I metodi iniziali servono a gestire correttamente il parallelismo e evitare vengano condivise fra thread delle variabili in maniera impropria.

Il metodo AdagradOmp alloca le strutture dati allo stesso modo della propria versione non OMP, e successivamente presenta il seguente corpo.

```
for(int it=0;it<iter;it++){
	for(i=0; i<n; i=i+k){

		int j;
		int v;
		if(n-i>k)
			v=k;
		else
			v=n-i;

		indice=i+v-1;

		metodo2(i, v, theta, osservazioni, lenght, gj, Gj, y);
		metodo1(indice, i, theta, osservazioni, lenght, gj, Gj, y);

		for(int j=0;j<lenght;j++)
			sommatoria[j]=0.0;

		int p=i;
		int pfin=i+v-1;

		for(int p=i; p<i+v;p++)
			for(int j=0;j<lenght;j++){
				indice = ((p-i) * lenght) +j;
				sommatoria[j]+=(rate/sqrt(Gj[indice]+eps))*gj[indice];
			}


		//sottraggo a theta
		for(j=0;j<lenght;j++){
			theta[j]=theta[j]-sommatoria[j]/v;
		}

	}
}
return theta;
```

Come avviene in batchOmp, vi è un metodo per analizzare con parallelismo tutte le osservazioni del corrente passo di batch (metodo2) e successivamente vi è un metodo per eseguire i calcoli necessari sulle osservazioni rimanenti;

Infatti nel codice è specificato “quante righe” della matrice osservazioni gestisca ogni thread lanciato in parallelo, e bisogna gestire il caso in cui questo numero non sia sottomultiplo delle osservazioni per passo di batch, quindi rimangano osservazioni in più

Analizziamo il codice nasm di **adagrad64Prod**, questo metodo riceve: p indice di osservazione di partenza, pfin (nel nostro caso p+2) indice di osservazione finale (analizzerà osservazioni da p a pfin), l’indice i del corrente passo di batch e tutte le strutture dati definite già in precedenza.

```
forp:
        MOV     RDX, [p]

        VXORPD  YMM1, YMM1              ;YMM1 contiene prodScal
        XOR     RSI, RSI
        XOR     RDI, RDI                ;EDI indice j del for del prodotto scalare
        MOV     ECX, EDX                ;metto in ECX il valore di p
        IMUL    ECX, [lenght]           ;moltiplico per lenght
        IMUL    ECX, ECX, dim           ;IMUL ci salva p*lenght*dim

        MOV     [plenghtdim], ECX

        XOR     RBX, RBX
        XOR     RDX, RDX

        MOV     RBX, [theta]
        MOV     RDX, [osservazioni]

        ADD     RDX, [plenghtdim]

forj1:
        VMOVAPD YMM2, [RBX]
        VMULPD  YMM2, YMM2, [RDX]
        VADDPD  YMM1, YMM1, YMM2
        VMOVAPD YMM3, [RBX + 32]
        VMULPD  YMM3, YMM3, [RDX + 32]
        VADDPD  YMM1, YMM1, YMM3
        VMOVAPD YMM4, [RBX + 64]
        VMULPD  YMM4, YMM4, [RDX + 64]
        VADDPD  YMM1, YMM1, YMM4
        VMOVAPD YMM5, [RBX + 96]
        VMULPD  YMM5, YMM5, [RDX + 96]
        VADDPD  YMM1, YMM1, YMM5

        ;questo chiude il forj1
        ADD     ESI, 16
        ADD     RDX, elementiunrolling
        ADD     RBX, elementiunrolling
        CMP     ESI, [lenght]
        JL      forj1


        VHADDPD YMM1, YMM1
        VHADDPD YMM1, YMM1              ;nella prima posizione di YMM1 ho l'effettivo valore di prodScal

        XOR     R9, R9
        XOR     RDX, RDX
        XOR     RBX, RBX


        MOV     R9, [y]

        MOV     EDX, [p]
        IMUL    EDI, EDX, dim

        VSUBSD  XMM1, [R9 + RDI]        ;in XMM1 ho prodScal = prodScal - y[p]

        VMOVSD  [aux], XMM1
```

forp scorre le osservazioni di interesse per il metodo, inizializzando per ciascuna di esse prodScal

Forj1 si occupa del calcolo di prodScal, come visto in precedenza

Dopo aver calcolato prodScal si esegue la differenza con la coordinata di y corrispondente all'osservazione

```
        VBROADCASTSD    YMM1, [aux]

        XOR     RDI, RDI
        XOR     RSI, RSI
        MOV     ESI, EDX
        SUB     ESI, [i]
        IMUL    ESI, [lenght]
        IMUL    ESI, ESI, dim

        IMUL    EDX, [lenght]
        IMUL    EDX, EDX, dim

        MOV     RAX, [qj]
        MOV     RBX, [osservazioni]
        MOV     RCX, [Gj]

        ADD     RBX, RDX
        ADD     RAX, RSI
        ADD     RCX, RSI

forj2:
        VMOVAPD YMM0, YMM1
        VMOVAPD YMM2, YMM1
        VMOVAPD YMM4, YMM1
        VMOVAPD YMM6, YMM1

        VMULPD  YMM0, YMM0, [RBX]
        VMULPD  YMM2, YMM2, [RBX + 32]
        VMULPD  YMM4, YMM4, [RBX + 64]
        VMULPD  YMM6, YMM6, [RBX + 96]

        VMOVAPD [RAX], YMM0         ;salvataqq
        VMOVAPD [RAX + 32], YMM2
        VMOVAPD [RAX + 64], YMM4
        VMOVAPD [RAX + 96], YMM6

        VMULPD  YMM0, YMM0
        VMULPD  YMM2, YMM2
        VMULPD  YMM4, YMM4
        VMULPD  YMM6, YMM6

        VADDPD  YMM0, [RCX]
        VADDPD  YMM2, [RCX + 32]
        VADDPD  YMM4, [RCX + 64]
        VADDPD  YMM6, [RCX + 96]

        VMOVAPD       [RCX], XMM0       ;salv
        VEXTRACTF128 XMM0,YMM0,1
        VMOVAPD       [RCX + 16], XMM0

        VMOVAPD       [RCX + 32], XMM2
        VEXTRACTF128 XMM2,YMM2,1
        VMOVAPD       [RCX + 48], XMM2
```

Forj2 aggiorna la singola riga di Gj, associata alla osservazione corrente

```
VMOVAPD       [RCX + 64], XMM4
VEXTRACTF128 XMM4,YMM4,1
VMOVAPD       [RCX + 80], XMM4

VMOVAPD       [RCX + 96], XMM6
VEXTRACTF128 XMM6,YMM6,1
VMOVAPD       [RCX + 112], XMM6

;questo chiude il forj2
ADD       EDI, 16
ADD       RAX, elementiunrolling
ADD       RBX, elementiunrolling
ADD       RCX, elementiunrolling
CMP       EDI, [lenght]
JL        forj2


; questo chiude il forp
XOR       RDX, RDX

MOV       EDX, [p]
INC       EDX
MOV       [p], EDX

XOR       RSI, RSI
MOV       ESI, [pfin]
CMP       EDX, ESI
JL        forp

; ------------------------------------------------------------
; Sequenza di uscita dalla funzione
; ------------------------------------------------------------
popaq
mov rsp, rbp
pop rbp
```

Conclusioni

In conclusione, pur avendo mostrando le differenti implementazioni delle versioni batch e Adagrad in assembly utilizzando AVX abbiamo riscontrato un errore di approssimazione derivante dall'uso delle operazioni di tale repertorio. Ci è sembrato doveroso quindi, al fine di non inficiare una corretta progettazione e applicazione del parallelismo nei codici C-OMP, lasciare l'implementazione effettiva in linguaggio C e mostrare il codice assembly all'interno di tale relazione.
L'obbiettivo di tale scelta è dimostrare di saper ragionare in ottica parallela anche con assembly (evitando race condition e scritture parallele) e validare l'algoritmo parallelo proposto, realizzandolo in C e mostrando come il risultato in quel caso risulti corretto e il codice non presenti errori concettuali che potrebbero causare errore ("segmentation fault").