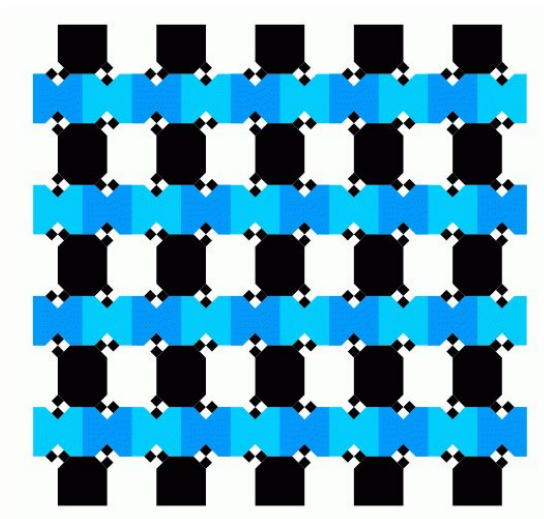


Introduzione alla Programmazione Parallela

Ultimo aggiornamento: 6 gennaio 2016



Camil Demetrescu

*Dipartimento di Ingegneria Informatica, Automatica e Gestionale “A. Ruberti”
Sapienza Università di Roma*

Indice

[2 Vettorizzazione](#)

[2.1 Compilazione di programmi che usano estensioni vettoriali](#)

[2.2 Tipi vettoriali](#)

[2.3 Istruzioni per leggere e scrivere su oggetti vettoriali](#)

[2.3.2 Copia da memoria a vettore \(load\)](#)

[2.3.3 Copia da vettore a memoria \(store\)](#)

[2.4 Operazioni aritmetiche](#)

[2.4.1 Addizione e sottrazione](#)

[2.4.2 Moltiplicazione](#)

[2.4.2 Minimo e massimo](#)

[2.5 Esempi](#)

[2.5.1 Somma vettoriale di array di dimensione arbitraria](#)

[2.5.2 Prodotto scalare di array di dimensione arbitraria](#)

[Bibliografia](#)

2 Vettorizzazione

La **vettorizzazione** è una classica forma di parallelismo SIMD a livello di istruzione. Consente di effettuare operazioni supportate nativamente dall'hardware su piccoli vettori di numeri. Ad esempio, con una singola istruzione macchina è possibile sommare due vettori di 32 char, 16 short, o 8 int. La vettorizzazione può essere effettuata **manualmente dal programmatore**, ma **spesso sono i compilatori a farlo in automatico**, generando codice assembly che usa istruzioni vettoriali. Ad esempio, molte funzioni della libreria C, come quelle di manipolazione delle stringhe, sono compilate in forma vettorizzata.

Le CPU moderne sono equipaggiate con **estensioni** che forniscono un ampio set di **registri e istruzioni macchina vettoriali**, originariamente introdotti per supportare lo sviluppo di applicazioni multimediali. In particolare, le architetture x86 hanno una lunga storia di estensioni successive, ognuna delle quali estende le precedenti con nuove funzionalità: MMX (1996), 3DNow! (1998), SSE (1999), SSE2 (2001), SSE3 (2004), SSSE3 (2006), SSE4 (2006), AVX (2008), AVX2 (2012), AVX-512 (2015).

Per svincolare il programmatore dal dover sviluppare direttamente in assembly le applicazioni che usano estensioni vettoriali, i compilatori forniscono una batteria di **intrinsic**, cioè particolari costrutti di alto livello (come tipi e funzioni C) che vengono tradotti direttamente in termini di costrutti dell'instruction set architecture (ISA) della macchina sottostante.

Esempio 1. Per sommare due vettori *a* e *b* di 4 int usando intrinsic SSE2 basta semplicemente scrivere `c = _mm_add_epi32(a,b)`, dove *a*, *b* e *c* sono variabili del tipo vettoriale `__m128i`. La chiamata `_mm_add_epi32` viene tradotta nell'istruzione macchina `paddq`. L'esempio seguente mostra come leggere, scrivere ed effettuare operazioni su variabili vettoriali:

```
#include <immintrin.h>
#include <stdio.h>

void somma(int A[4], int B[4], int C[4]) {
    __m128i a, b, c;           // dich. variabili vettoriali
    a = _mm_load_si128((const __m128i*)A); // copia primo vettore in a
    b = _mm_load_si128((const __m128i*)B); // copia secondo vettore in b
    c = _mm_add_epi32(a,b);     // calcola la somma vett. di a e b
    _mm_store_si128((__m128i*)C, c); // copia risultato c in C
}

int main() {
    int A[4] = { 1, 2, 3, 4 };
    int B[4] = { 4, 3, 2, 1 };
    int C[4];
```

```

    somma(A,B,C);
    printf("%d %d %d %d\n", C[0], C[1], C[2], C[3]);    // stampa 5 5 5 5
    return 0;
}

```

Si noti l'uso degli intrinsic load e store per trasferire dati da memoria a variabile vettoriale e viceversa. Vedremo le istruzioni più in dettaglio nel seguito. In questa dispensa analizzeremo gli intrinsic più frequentemente usati, per una trattazione più completa si rimanda al sito Web [\[IntelIntr\]](#), che fornisce una documentazione dettagliata degli intrinsic della famiglia x86.

2.1 Compilazione di programmi che usano estensioni vettoriali

In questa dispensa assumiamo di usare una piattaforma equipaggiata con compilatore `gcc`. Per compilare un programma che usa estensioni vettoriali, includeremo la header `immintrin.h` che rende visibili gli intrinsic e aggiungeremo alla riga di comando `gcc` un'opzione per abilitare il supporto alla vettorizzazione. Se `x` è il nome dell'estensione da usare, l'opzione da usare è `-mX`. Ad esempio, per usare AVX2 scriveremo `-mavx2`. Si noti che ogni opzione include tutte quelle precedenti. Ad esempio, usando `-msse2` si possono usare tutte le funzionalità offerte da SSE e SSE2.

Esempio 2. Il programma dell'esempio 1 può essere compilato con `gcc -msse2 vecsum.c -O2 -o vecsum` ed eseguito con `./vecsum`.

Compilando il programma con `gcc -msse2 vecsum.c -O2 -S`, si ottiene il seguente codice x86-64 per la funzione `somma`¹:

```

somma:
    movdqa (%rdi), %xmm0    # <-- _mm_load_si128
    paddq  (%rsi), %xmm0    # <-- _mm_add_epi32
    movdqa %xmm0, (%rdx)    # <-- _mm_store_si128
    ret

```

Si noti come gli intrinsic si mappino su istruzioni macchina e registri (`xmm0` nell'esempio).

2.2 Tipi vettoriali

I **tipi vettoriali** descrivono oggetti vettoriali che possono essere tenuti in particolari registri della CPU. I tipi vettoriali vengono anche chiamati **packed data type**, poiché consentono di impaccare più valori scalari dello stesso tipo in un unico oggetto. I tipi principali sono:

¹ Si noti che, secondo le convenzioni AMD64 ABI, i primi tre parametri della funzione `somma` (indirizzi degli array `A`, `B`, `C`) sono passati nei registri `rdi`, `rsi` e `rdx`, rispettivamente.

Tipo			Dimensione	Registri usati	Estensione vettoriale x86
Intero	Precisione singola	Precisione doppia			
__m64			8 byte	8 (MM0-MM7)	MMX
__m128i	__m128	__m128d	16 byte	16 (XMM0-XMM15)	SSE
__m256i	__m256	__m256d	32 byte	16 (YMM0-YMM15)	AVX
__m512i	__m512	__m512d	64 byte	32 (ZMM0-ZMM31)	AVX-512

La loro capienza in termini vettoriali è come segue:

Intero	Capienza			Precis. singola	Capienza	Precis. doppia	Capienza
__m64	char[8]	short[4]	int[2]				
__m128i	char[16]	short[8]	int[4]	__m128	float[4]	__m128d	double[2]
__m256i	char[32]	short[16]	int[8]	__m256	float[8]	__m256d	double[4]
__m512i	char[64]	short[32]	int[16]	__m512	float[16]	__m512d	double[8]

Si noti che i tipi a 512 bit non sono ancora supportati, ma è previsto il loro inserimento nell'architettura Knights Landing di Intel nel 2016. Tecnicamente, i tipi vettoriali sono `struct C`. E' quindi possibile assegnare un oggetto vettoriale ad un altro mediante l'operatore di assegnamento `=`.

Esempio.

```
__m128i a = ...;
__m128i b = a;    // ok, è possibile assegnare un oggetto vett. a un
altro
```

Inoltre, passare un oggetto vettoriale come **parametro a una funzione** implica passare una copia dell'intero oggetto, e non del solo indirizzo come avverrebbe passando un array come parametro.

2.3 Istruzioni per leggere e scrivere su oggetti vettoriali

Per poter effettuare un calcolo vettoriale è innanzitutto necessario caricare l'input in opportuni oggetti vettoriali. Allo stesso modo, è necessario poter estrarre il risultato dagli oggetti vettoriali che sono il risultato del calcolo.

2.3.1 Allineamento in memoria

Le estensioni vettoriali forniscono istruzioni che permettono di trasferire dati da memoria a oggetto vettoriale, e viceversa. Ci sono **due versioni** di queste istruzioni: quelle che richiedono che l'indirizzo di memoria acceduto sia **allineato** a un multiplo della dimensione dell'oggetto vettoriale copiato, e quelle che possono accedere a **qualsiasi indirizzo anche non allineato**. Le versioni allineate garantiscono prestazioni migliori. Se si passa un indirizzo non allineato a una funzione che richiede allineamento, potrebbe generarsi un errore di accesso alla memoria durante l'esecuzione (**segmentation fault**, o general-protection exception).

Allineamento di variabili.

Il compilatore `gcc` fornisce supporto per dichiarare variabili allineate a indirizzi multipli di una costante `n` usando la direttiva `__attribute__((aligned(n)))`.

Esempio. La seguente dichiarazione introduce un array di interi allineato a 16 byte:

```
int v[4] __attribute__((aligned(16))) = { 3, 5, 7, 1 };
```

Allineamento di blocchi allocati dinamicamente.

La funzione `malloc` potrebbe non garantire il corretto allineamento di oggetti vettoriali come quelli a 256 bit. Per allocare dinamicamente blocchi con l'allineamento desiderato è possibile usare la funzione `posix_memalign` definita dallo standard POSIX 1003.1d [[PosixMemalign](#)], oppure funzioni come `memalign` o `valloc` fornite dalla GNU standard library (che potrebbero non essere tuttavia disponibili su tutte le piattaforme) [[GNUMemalign](#)].

2.3.2 Copia da memoria a vettore (load)

Queste operazioni consentono di copiare dati da memoria a oggetto vettoriale.

<code>__m128i __mm_load_si128 (__m128i const* mem_addr)</code> <code>__m128i __mm_loadu_si128 (__m128i const* mem_addr)</code>		array interi all. → <code>__m128i</code> array interi → <code>__m128i</code>	
Opzione gcc	<code>-msse2</code>	Istruzione	<code>movdqa xmm, m128</code> <code>movdqu xmm, m128</code>
Descrizione	Restituisce oggetto <code>__m128i</code> inizializzato con i 16 byte di dati interi (array di <code>char</code> , <code>short</code> , <code>int</code> , <code>long</code> , con o senza segno) presi dall'indirizzo di memoria <code>mem_addr</code> . La versione <code>__mm_load_si128</code> richiede che <code>mem_addr</code> sia allineato a un multiplo di 16 byte.		
Esempio	<pre>int va[4] __attribute__((aligned(16))) = { 3, 5, 7, 1 }; __m128i a = __mm_load_si128((__m128i const*)va); int vu[4] = { 3, 5, 7, 1 }; __m128i b = __mm_loadu_si128((__m128i const*)vu);</pre>		

__m256i __mm256_load_si256 (__m256i const* mem_addr) __m256i __mm256_loadu_si256 (__m256i const* mem_addr)		array interi all. → __m256i array interi → __m256i	
Opzione gcc	-mavx	Istruzione	vmovdqa ymm, m256 vmovdqu ymm, m256
Descrizione	Restituisce oggetto __m256i inizializzato con i 32 byte di dati interi (array di char, short, int, long, con o senza segno) presi dall'indirizzo di memoria mem_addr. La versione __mm256_load_si256 richiede che mem_addr sia allineato a un multiplo di 32 byte.		
Esempio	<pre>int va[8] __attribute__((aligned(32))) = { 3, 5, 7, 1, 3, 2, 6, 1 }; __m256i a = __mm256_load_si256((__m256i const*)va); int vu[8] = { 3, 5, 7, 1, 3, 2, 6, 1 }; __m256i b = __mm256_loadu_si256((__m256i const*)vu);</pre>		

Si noti che le stesse istruzioni possono essere usate per trasferire array di tipi interi diversi. Per questo motivo, è richiesto un cast dell'indirizzo (es. (__m256i const*)v).

__m128 __mm_load_ps (float const* mem_addr) __m128 __mm_loadu_ps (float const* mem_addr)		float[4] all. → __m128 float[4] → __m128	
Opzione gcc	-msse	Istruzione	movaps xmm, m128 movups xmm, m128
Descrizione	Restituisce oggetto __m128 inizializzato con l'array di 4 float all'indirizzo di memoria mem_addr. La versione __mm_load_ps richiede che mem_addr sia allineato a un multiplo di 16 byte.		
Esempio	<pre>float va[4] __attribute__((aligned(16))) = { 3.1, 5.3, 7.9, 1 }; __m128 a = __mm_load_ps(va); float vu[4] = { 3.1, 5.3, 7.9, 1 }; __m128 b = __mm_loadu_ps(vu);</pre>		

__m128d __mm_load_pd (double const* mem_addr) __m128d __mm_loadu_pd (double const* mem_addr)		double[2] all. → __m128d double[2] → __m128d	
Opzione gcc	-msse2	Istruzione	movapd xmm, m128 movupd xmm, m128
Descrizione	Restituisce oggetto __m128d inizializzato con l'array di 2 double all'indirizzo di memoria mem_addr. La versione __mm_load_pd richiede che mem_addr sia allineato a un multiplo di 16 byte.		
Esempio	<pre>double va[2] __attribute__((aligned(16))) = { 3.1, 5.3 }; __m128d a = __mm_load_pd(va); double vu[2] = { 3.1, 5.3 }; __m128d b = __mm_loadu_pd(vu);</pre>		

	<code>__m128d b = _mm_loadu_pd(vu);</code>
--	--

<code>__m256 _mm256_load_ps (float const* mem_addr)</code> <code>__m256 _mm256_loadu_ps (float const* mem_addr)</code>		<code>float[8] all. → __m256</code> <code>float[8] → __m256</code>	
Opzione gcc	<code>-mavx</code>	Istruzione	<code>vmovaps ymm, m256</code> <code>vmovups ymm, m256</code>
Descrizione	Restituisce oggetto <code>__m256</code> inizializzato con l'array di 8 float all'indirizzo di memoria <code>mem_addr</code> . La versione <code>_mm256_load_ps</code> richiede che <code>mem_addr</code> sia allineato a un multiplo di 32 byte.		
Esempio	<pre>float va[8] __attribute__((aligned(32))) = { 3.1, 5.3, 7.9, 1, 3.1, 5.3, 7.9, 1 }; __m256 a = _mm256_load_ps(va); float vu[8] = { 3.1, 5.3, 7.9, 1, 3.1, 5.3, 7.9, 1 }; __m256 b = _mm256_loadu_ps(vu);</pre>		

<code>__m256d _mm256_load_pd (double const* mem_addr)</code> <code>__m256d _mm256_loadu_pd (double const* mem_addr)</code>		<code>double[4] all. → __m256d</code> <code>double[4] → __m256d</code>	
Opzione gcc	<code>-mavx</code>	Istruzione	<code>vmovapd ymm, m256</code> <code>vmovupd ymm, m256</code>
Descrizione	Restituisce oggetto <code>__m256d</code> inizializzato con l'array di 4 double all'indirizzo di memoria <code>mem_addr</code> . La versione <code>_mm256_load_pd</code> richiede che <code>mem_addr</code> sia allineato a un multiplo di 32 byte.		
Esempio	<pre>double va[4] __attribute__((aligned(32))) = { 3.1, 5.3, 7.9, 1 }; __m256d a = _mm256_load_pd(va); double vu[4] = { 3.1, 5.3, 7.9, 1 }; __m256d b = _mm256_loadu_pd(vu);</pre>		

2.3.3 Copia da vettore a memoria (store)

Queste operazioni consentono di copiare dati da un oggetto vettoriale a memoria. Come per la load, ci sono due versioni, a seconda che l'indirizzo destinazione sia allineato o meno.

<code>void _mm_store_si128 (__m128i* mem_addr, __m128i a)</code> <code>void _mm_storeu_si128 (__m128i* mem_addr, __m128i a)</code>		<code>__m128i → array interi all.</code> <code>__m128i → array interi</code>	
Opzione gcc	<code>-msse2</code>	Istruzione	<code>movdqa m128, xmm</code> <code>movdqu m128, xmm</code>
Descrizione	Copia l'oggetto a nei 16 byte di dati interi (array di char, short, int, long, con o senza		

	segno) all'indirizzo di memoria <code>mem_addr</code> . La versione <code>_mm_store_si128</code> richiede che <code>mem_addr</code> sia allineato a un multiplo di 16 byte.
Esempio	<pre> __m128i a = _mm_set_epi32(2, 5, 3, 1); int va[4] __attribute__((aligned(16))); _mm_store_si128((__m128i*)va, a); __m128i b = _mm_set_epi32(2, 5, 3, 1); int vu[4]; _mm_storeu_si128((__m128i*)vu, b); </pre>

<pre> void _mm256_store_si256 (__m256i* mem_addr, __m256i a) __m256i → array interi all. void _mm256_storeu_si256(__m256i* mem_addr, __m256i a) __m256i → array interi </pre>			
Opzione gcc	-msse2	Istruzione	<pre> vmovdqa m256, ymm vmovdqu m256, ymm </pre>
Descrizione	Copia l'oggetto a nei 32 byte di dati interi (array di char, short, int, long, con o senza segno) all'indirizzo di memoria <code>mem_addr</code> . La versione <code>_mm_store_si128</code> richiede che <code>mem_addr</code> sia allineato a un multiplo di 32 byte.		
Esempio	<pre> __m256i a = _mm256_set_epi32(2, 5, 3, 1, 7, 9, 4, 6); int va[8] __attribute__((aligned(32))); _mm256_store_si256((__m256i*)va, a); __m256i b = _mm256_set_epi32(2, 5, 3, 1, 7, 9, 4, 6); int vu[8]; _mm256_storeu_si256((__m256i*)vu, b); </pre>		

<pre> void _mm_store_ps (float* mem_addr, __m128 a) __m128 → float[4] all. void _mm_storeu_ps(float* mem_addr, __m128 a) __m128 → float[4] </pre>			
Opzione gcc	-msse	Istruzione	<pre> movaps m128, xmm movups m128, xmm </pre>
Descrizione	Copia l'oggetto a nell'array di 4 float all'indirizzo di memoria <code>mem_addr</code> . La versione <code>_mm_store_ps</code> richiede che <code>mem_addr</code> sia allineato a un multiplo di 16 byte.		
Esempio	<pre> __m128 a = _mm_set_ps(2.1, 5.2, 3.1, 1.9); float va[4] __attribute__((aligned(16))); _mm_store_ps(va, a); __m128 b = _mm_set_ps(2.1, 5.2, 3.1, 1.9); float vu[4]; _mm_storeu_ps(vu, b); </pre>		

void _mm256_store_ps (float* mem_addr, __m256 a) __m256 → float[8] all. void _mm256_storeu_ps (float* mem_addr, __m256 a) __m256 → float[8]			
Opzione gcc	-mavx	Istruzione	vmovaps m256, ymm vmovups m256, ymm
Descrizione	Copia l'oggetto a nell'array di 8 float all'indirizzo di memoria mem_addr. La versione _mm256_store_ps richiede che mem_addr sia allineato a un multiplo di 32 byte.		
Esempio	<pre>__m256 a = _mm256_set_ps(2.1, 5.2, 3.1, 1.9, 7.3, 9.5, 4.3, 6.7); float va[8] __attribute__((aligned(32))); _mm256_store_ps(va, a); __m256 b = _mm256_set_ps(2.1, 5.2, 3.1, 1.9, 7.3, 9.5, 4.3, 6.7); float vu[8]; _mm256_storeu_ps(vu, b);</pre>		

void _mm_store_pd (double* mem_addr, __m128d a) __m128d → double[2] all. void _mm_storeu_pd (double* mem_addr, __m128d a) __m128d → double[2]			
Opzione gcc	-msse2	Istruzione	movapd m128, xmm movupd m128, xmm
Descrizione	Copia l'oggetto a nell'array di 2 double all'indirizzo di memoria mem_addr. La versione _mm_store_pd richiede che mem_addr sia allineato a un multiplo di 16 byte.		
Esempio	<pre>__m128d a = _mm_set_pd(2.1, 5.2); double va[2] __attribute__((aligned(16))); _mm_store_pd(va, a); __m128d b = _mm_set_pd(2.1, 5.2); double vu[2]; _mm_storeu_pd(vu, b);</pre>		

void _mm256_store_pd (double* mem_addr, __m256d a) __m256d → double[4] all. void _mm256_storeu_pd (double* mem_addr, __m256d a) __m256d → double[4]			
Opzione gcc	-mavx	Istruzione	vmovapd m256, ymm vmovupd m256, ymm
Descrizione	Copia l'oggetto a nell'array di 4 double all'indirizzo di memoria mem_addr. La versione _mm256_store_ps richiede che mem_addr sia allineato a un multiplo di 32 byte.		
Esempio	<pre>__m256d a = _mm256_set_pd(2.1, 5.2, 3.1, 1.9); double va[4] __attribute__((aligned(32))); _mm256_store_pd(va, a); __m256d b = _mm256_set_pd(2.1, 5.2, 3.1, 1.9);</pre>		

```
double vu[4];
__mm256_storeu_pd(vu, b);
```

2.4 Operazioni aritmetiche

2.4.1 Addizione e sottrazione

Le operazioni di **sottrazione** sono del tutto analoghe alle seguenti, rimpiazzando `add` con `sub`.

<pre>__m128i __mm_add_epi8 (__m128i a, __m128i b) __m128i __mm_add_epi16(__m128i a, __m128i b) __m128i __mm_add_epi32(__m128i a, __m128i b) __m128i __mm_add_epi64(__m128i a, __m128i b)</pre>			
		__m128i + __m128i	
Opzione gcc	-msse2	Istruzione	<p> <code>paddb xmm, xmm</code> <code>paddw xmm, xmm</code> <code>paddq xmm, xmm</code> <code>paddq xmm, xmm</code> </p>
Descrizione	Calcola <code>a+b</code> come vettori di <code>char</code> (epi8), <code>short</code> (epi16), <code>int</code> (epi32), oppure <code>long</code> (epi64) e restituisce il risultato.		
Esempio	<pre>__m128i a = __mm_set_epi32(4, 3, 2, 1); __m128i b = __mm_set_epi32(1, 2, 3, 4); __m128i c = __mm_add_epi32(a,b); // c = [5, 5, 5, 5]</pre>		

<pre>__m256i __mm256_add_epi8 (__m256i a, __m256i b) __m256i __mm256_add_epi16(__m256i a, __m256i b) __m256i __mm256_add_epi32(__m256i a, __m256i b) __m256i __mm256_add_epi64(__m256i a, __m256i b)</pre>			
		__m256i + __m256i	
Opzione gcc	-mavx2	Istruzione	<p> <code>vpaddb ymm, ymm, ymm</code> <code>vpaddw ymm, ymm, ymm</code> <code>vpaddq ymm, ymm, ymm</code> <code>vpaddq ymm, ymm, ymm</code> </p>
Descrizione	Calcola <code>a+b</code> come vettori di <code>char</code> (epi8), <code>short</code> (epi16), <code>int</code> (epi32), oppure <code>long</code> (epi64) e restituisce il risultato.		
Esempio	<pre>__m256i a = __mm256_set_epi32(1, 2, 3, 4, 5, 6, 7, 8); __m256i b = __mm256_set_epi32(8, 7, 6, 5, 4, 3, 2, 1); __m256i c = __mm256_add_epi32(a,b); // c = [5, 5, 5, 5, 5, 5, 5, 5]</pre>		

<pre>__m128 __mm_add_ps (__m128 a, __m128 b)</pre>			
		__m128 + __m128	
Opzione gcc	-msse	Istruzione	<code>addps xmm, xmm</code>
Descrizione	Calcola <code>a+b</code> come vettori di <code>float</code> e restituisce il risultato.		

Esempio	<pre>__m128 a = _mm_set_ps(4.0, 3.0, 2.0, 1.0); __m128 b = _mm_set_ps(1.0, 2.0, 3.0, 4.0); __m128 c = _mm_add_ps(a,b); // c = [5.0, 5.0, 5.0, 5.0]</pre>
----------------	---

__m128d _mm_add_pd (__m128d a, __m128d b) __m128d + __m128d			
Opzione gcc	-msse2	Istruzione	addpd xmm, xmm
Descrizione	Calcola a+b come vettori di double e restituisce il risultato.		
Esempio	<pre>__m128d a = _mm_set_pd(4.0, 3.0); __m128d b = _mm_set_pd(1.0, 2.0); __m128d c = _mm_add_pd(a,b); // c = [5.0, 5.0]</pre>		

__m256 _mm256_add_ps (__m256 a, __m256 b) __m256 + __m256			
Opzione gcc	-mavx	Istruzione	vaddps ymm, ymm, ymm
Descrizione	Calcola a+b come vettori di float e restituisce il risultato.		
Esempio	<pre>__m256 a = _mm256_set_ps(4.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0); __m256 b = _mm256_set_ps(1.0, 2.0, 3.0, 4.0, 1.0, 2.0, 3.0, 4.0); __m256 c = _mm256_add_ps(a,b); // c = [5.0, 5.0, 5.0, 5.0, ...]</pre>		

__m256d _mm256_add_pd (__m256d a, __m256d b) __m256d + __m256d			
Opzione gcc	-mavx	Istruzione	vaddpd ymm, ymm, ymm
Descrizione	Calcola a+b come vettori di double e restituisce il risultato.		
Esempio	<pre>__m256d a = _mm256_set_pd(4.0, 3.0, 2.0, 1.0); __m256d b = _mm256_set_pd(1.0, 2.0, 3.0, 4.0); __m256d c = _mm256_add_pd(a,b); // c = [5.0, 5.0, 5.0, 5.0]</pre>		

2.4.2 Moltiplicazione

<pre>__m128i _mm_mullo_epi16(__m128i a, __m128i b) __m128i _mm_mullo_epi32(__m128i a, __m128i b)</pre> __m128i * __m128i			
Opzione gcc	-msse2 (epi16) -msse4.1 (epi32)	Istruzione	pmullw xmm, xmm pmulld xmm, xmm
Descrizione	Moltiplica gli elementi corrispondenti in a e b come vettori di short (epi16) o int (epi32) e restituisce il risultato. Ciascun prodotto di valori a n bit è calcolato a 2n bit, prendendo i n bit meno significativi del risultato.		

Esempio	<pre>__m128i a = _mm_set_epi32(4, 3, 2, 1); __m128i b = _mm_set_epi32(1, 2, 3, 4); __m128i c = _mm_mullo_epi32(a,b); // c = [4, 6, 6, 4]</pre>
----------------	---

<pre>__m256i _mm256_mullo_epi16(__m256i a, __m256i b) __m256i * __m256i __m256i _mm256_mullo_epi32(__m256i a, __m256i b)</pre>			
Opzione gcc	-mavx2	Istruzione	<pre>vpmullw ymm, ymm, ymm vpmulld ymm, ymm, ymm</pre>
Descrizione	Moltiplica gli elementi corrispondenti in a e b come vettori di short (epi16) o int (epi32) e restituisce il risultato. Ciascun prodotto di valori a n bit è calcolato a 2n bit, prendendo i n bit meno significativi del risultato.		
Esempio	<pre>__m256i a = _mm256_set_epi32(4, 3, 2, 1, 4, 3, 2, 3); __m256i b = _mm256_set_epi32(1, 2, 3, 4, 2, 2, 0, 3); __m256i c = _mm256_mullo_epi32(a,b); // c = [4, 6, 6, 4, 8, 6, 0, 9]</pre>		

Per altre versioni delle operazioni di moltiplicazione, come quelle in virgola mobile, si rimanda a [\[IntelIntr\]](#).

2.4.2 Minimo e massimo

Le operazioni di **minimo** sono del tutto analoghe a quelle che seguono, rimpiazzando `max` con `min`.

<pre>__m128i _mm_max_epi8 (__m128i a, __m128i b) __m128i _mm_max_epi16(__m128i a, __m128i b) __m128i _mm_max_epi32(__m128i a, __m128i b) __m128i _mm_max_epu8 (__m128i a, __m128i b) __m128i _mm_max_epu16(__m128i a, __m128i b) __m128i _mm_max_epu32(__m128i a, __m128i b)</pre>			
Opzione gcc	-msse4.1 -msse2 (epu8, epi16)	Istruzione	<pre>pmaxsb xmm, xmm pmaxsw xmm, xmm pmaxsd xmm, xmm pmaxub xmm, xmm pmaxuw xmm, xmm pmaxud xmm, xmm</pre>
Descrizione	Calcola il massimo degli elementi corrispondenti in a e b come vettori di char (epi8), short (epi16), int (epi32), unsigned char (epu8), unsigned short (epu16), unsigned int (epu32), e restituisce il risultato.		
Esempio	<pre>__m128i a = _mm_set_epi32(4, -3, 2, -1); __m128i b = _mm_set_epi32(1, 2, 3, -4); __m128i c = _mm_max_epi32(a,b); // c = [4, 2, 3, -1] __m128i d = _mm_max_epu32(a,b); // d = [4, -3, 3, -1]</pre>		

<pre> __m256i _mm256_max_epi8 (__m256i a, __m256i b) __m256i _mm256_max_epi16(__m256i a, __m256i b) __m256i _mm256_max_epi32(__m256i a, __m256i b) __m256i _mm256_max_epu8 (__m256i a, __m256i b) __m256i _mm256_max_epu16(__m256i a, __m256i b) __m256i _mm256_max_epu32(__m256i a, __m256i b) </pre>			
Opzione gcc	-mavx2	Istruzione	<pre> vpmaxsb ymm, ymm, ymm vpmaxsw ymm, ymm, ymm vpmaxsd ymm, ymm, ymm vpmaxub ymm, ymm, ymm vpmaxuw ymm, ymm, ymm vpmaxud ymm, ymm, ymm </pre>
Descrizione	Calcola il massimo degli elementi corrispondenti in a e b come vettori di char (epi8), short (epi16), int (epi32), unsigned char (epu8), unsigned short (epu16), unsigned int (epu32), e restituisce il risultato.		
Esempio	<pre> __m256i a = _mm256_set_epi32(4, -3, 2, -1, 4, -3, 2, -1); __m256i b = _mm256_set_epi32(1, 2, 3, -4, 1, 2, 3, -4); __m256i c = _mm256_max_epi32(a,b); // c = [4, 2, 3, -1, ...] __m256i f = _mm256_max_epu32(a,b); // c = [4, -3, 3, -1, ...] </pre>		

Per altre operazioni si rimanda a [\[IntelIntr\]](#).

2.5 Tabella riassuntiva

Riassumiamo nella seguente tabella le operazioni viste nei paragrafi precedenti:

Copia da memoria a vettore (load)	
<pre> __m128i _mm_load_si128 (__m128i const* mem_addr) __m128i _mm_loadu_si128(__m128i const* mem_addr) </pre>	<pre> array interi all. → __m128i array interi → __m128i </pre>
<pre> __m256i _mm256_load_si256 (__m256i const* mem_addr) __m256i _mm256_loadu_si256(__m256i const* mem_addr) </pre>	<pre> array interi all. → __m256i array interi → __m256i </pre>
<pre> __m128 _mm_load_ps (float const* mem_addr) __m128 _mm_loadu_ps(float const* mem_addr) </pre>	<pre> float[4] allin. → __m128 float[4] → __m128 </pre>
<pre> __m128d _mm_load_pd (double const* mem_addr) __m128d _mm_loadu_pd(double const* mem_addr) </pre>	<pre> double[2] allin. → __m128d double[2] → __m128d </pre>

__m256 __mm256_load_ps (float const* mem_addr) __m256 __mm256_loadu_ps(float const* mem_addr)	float[8] allin. → __m256 float[8] → __m256
__m256d __mm256_load_pd (double const* mem_addr) __m256d __mm256_loadu_pd(double const* mem_addr)	double[4] allin. → __m256d double[4] → __m256d
Copia da vettore a memoria (store)	
void __mm_store_si128 (__m128i* mem_addr, __m128i a) void __mm_storeu_si128(__m128i* mem_addr, __m128i a)	__m128i → array interi allin. __m128i → array interi
void __mm256_store_si256 (__m256i* mem_addr, __m256i a) void __mm256_storeu_si256(__m256i* mem_addr, __m256i a)	__m256i → array interi allin. __m256i → array interi
void __mm_store_ps (float* mem_addr, __m128 a) void __mm_storeu_ps(float* mem_addr, __m128 a)	__m128 → float[4] allin. __m128 → float[4]
void __mm256_store_ps (float* mem_addr, __m256 a) void __mm256_storeu_ps(float* mem_addr, __m256 a)	__m256 → float[8] allin. __m256 → float[8]
void __mm_store_pd (double* mem_addr, __m128d a) void __mm_storeu_pd(double* mem_addr, __m128d a)	__m128d → double[2] allin. __m128d → double[2]
void __mm256_store_pd (double* mem_addr, __m256d a) void __mm256_storeu_pd(double* mem_addr, __m256d a)	__m256d → double[4] allin. __m256d → double[4]
Addizione	
__m128i __mm_add_epi8 (__m128i a, __m128i b) __m128i __mm_add_epi16(__m128i a, __m128i b) __m128i __mm_add_epi32(__m128i a, __m128i b) __m128i __mm_add_epi64(__m128i a, __m128i b)	char[16] + char[16] short[8] + short[8] int[4] + int[4] long[2] + long[2]
__m256i __mm256_add_epi8 (__m256i a, __m256i b) __m256i __mm256_add_epi16(__m256i a, __m256i b) __m256i __mm256_add_epi32(__m256i a, __m256i b) __m256i __mm256_add_epi64(__m256i a, __m256i b)	char[32] + char[32] short[16] + short[16] int[8] + int[8] long[4] + long[4]
__m128 __mm_add_ps(__m128 a, __m128 b)	float[4] + float[4]
__m128d __mm_add_pd(__m128d a, __m128d b)	double[2] + double[2]
__m256 __mm256_add_ps(__m256 a, __m256 b)	float[8] + float[8]
__m256d __mm256_add_pd(__m256d a, __m256d b)	double[4] + double[4]
Prodotto	
__m128i __mm_mullo_epi16(__m128i a, __m128i b) __m128i __mm_mullo_epi32(__m128i a, __m128i b)	short[8] * short[8] int[4] * int[4]
__m256i __mm256_mullo_epi16(__m256i a, __m256i b) __m256i __mm256_mullo_epi32(__m256i a, __m256i b)	short[16] * short[16] int[8] * int[8]

Massimo

<code>__m128i __mm_max_epi8 (__m128i a, __m128i b)</code>	<code>max(char[16])</code>
<code>__m128i __mm_max_epi16(__m128i a, __m128i b)</code>	<code>max(short[8])</code>
<code>__m128i __mm_max_epi32(__m128i a, __m128i b)</code>	<code>max(int[4])</code>
<code>__m128i __mm_max_epu8 (__m128i a, __m128i b)</code>	<code>max(unsigned char[16])</code>
<code>__m128i __mm_max_epu16(__m128i a, __m128i b)</code>	<code>max(unsigned short[8])</code>
<code>__m128i __mm_max_epu32(__m128i a, __m128i b)</code>	<code>max(unsigned int[4])</code>
<code>__m256i __mm256_max_epi8 (__m256i a, __m256i b)</code>	<code>max(char[32])</code>
<code>__m256i __mm256_max_epi16(__m256i a, __m256i b)</code>	<code>max(short[16])</code>
<code>__m256i __mm256_max_epi32(__m256i a, __m256i b)</code>	<code>max(int[8])</code>
<code>__m256i __mm256_max_epu8 (__m256i a, __m256i b)</code>	<code>max(unsigned char[32])</code>
<code>__m256i __mm256_max_epu16(__m256i a, __m256i b)</code>	<code>max(unsigned short[16])</code>
<code>__m256i __mm256_max_epu32(__m256i a, __m256i b)</code>	<code>max(unsigned int[8])</code>

2.5 Esempi

2.5.1 Somma vettoriale di array di dimensione arbitraria

Si vuole realizzare una funzione che, dati tre array `A`, `B` e `C` di `float` di dimensione `n`, calcola la somma vettoriale `C=A+B` utilizzando istruzioni AVX. Partiamo da una semplice versione sequenziale e vediamo come vettorizzarla:

```
void somma_float(const float* A, const float* B, float* C, size_t n) {
    long i;
    for (i=0; i<n; i++) C[i] = A[i] + B[i];
}
```

Usando istruzioni AVX, possiamo effettuare somme di 8 `float` alla volta impaccati in oggetti vettoriali di tipo `__m256`. Partiamo con una trasformazione del codice chiamata **loop unrolling parziale di ragione k**, che consiste nel ridurre di un fattore `k` il numero di iterazioni effettuando `k` operazioni per iterazione:

```
void somma_float(const float* A, const float* B, float* C, size_t n) {
    long i;
    for (i=0; i+7<n; i+=8) {                                // loop unrolling di ragione 8
        C[i]   = A[i]   + B[i];
        C[i+1] = A[i+1] + B[i+1];
        C[i+2] = A[i+2] + B[i+2];
        C[i+3] = A[i+3] + B[i+3];
        C[i+4] = A[i+4] + B[i+4];
        C[i+5] = A[i+5] + B[i+5];
        C[i+6] = A[i+6] + B[i+6];
        C[i+7] = A[i+7] + B[i+7];
    }
    for (; i<n; i++) C[i] = A[i] + B[i]; // serve se 8 non divide n
}
```

Osserviamo che, se `n` non è divisibile per 8, non è possibile effettuare tutte le somme a 8 a 8. Un secondo ciclo effettua quindi le somme rimanenti, che sono al più 7.

Rimpiazziamo ora il corpo del primo ciclo con istruzioni vettoriali che effettuano la somma:

```
#include <immintrin.h>

void somma_float(const float* A, const float* B, float* C, size_t n) {
    long i;
```

```

for (i=0; i+7<n; i+=8) { // loop unrolling di ragione 8
    __m256 a = _mm256_loadu_ps(A+i); // copia A[i..i+7] in a
    __m256 b = _mm256_loadu_ps(B+i); // copia B[i..i+7] in b
    __m256 c = _mm256_add_ps(a,b); // calcola c=a+b
    _mm256_storeu_ps(C+i, c); // copia c in C[i..i+7]
}
for (; i<n; i++) C[i] = A[i] + B[i]; // serve se 8 non divide n
}

```

Possiamo testare il programma con un semplice main di prova:

```

int main() {
    long i;
    float A[17] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 };
    float B[17] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 };
    float C[17];
    somma_float((const float*)A, (const float*)B, C, 17);
    for (i=0; i<17; i++) printf("%f ", C[i]); // stampa 2 4 6 8 10 12 ...
    printf("\n");
}

```

Si noti che tutte le operazioni vettoriali utilizzate usano istruzioni AVX, pertanto possiamo usare l'opzione `-mavx`. Compiliamo con `gcc somma_float.c -o somma_float -O1 -mavx` ed eseguiamo con `./somma_float`.

Approfondimento.

E' interessante analizzare il codice assembly generato con `gcc somma_float.c -S -O1 -mavx` (gcc versione 4.8.4 in Ubuntu). Il codice del loop vettorizzato è:

```

.L15:
    vmovups (%rdi,%rax), %ymm1
    vmovups (%rsi,%rax), %ymm0
    vaddps %ymm0, %ymm1, %ymm0
    vmovups %ymm0, (%rdx,%rax)
    addq $8, %r8
    addq $32, %rax
    cmpq %r9, %r8
    jne .L15

```

dove `%rdi` è A, `%rsi` è B, `%rdx` è C, `%r8` è i ed `%r9` è n.

2.5.2 Prodotto scalare di array di dimensione arbitraria

Si vuole realizzare una funzione che, dati due array `A`, `B` di `int` di dimensione `n`, calcola il prodotto scalare $s=A \cdot B$ utilizzando istruzioni AVX. Partiamo da una semplice versione sequenziale e vediamo come vettorizzarla:

```
int prods_int(const int* A, const int* B, size_t n) {
    long i;
    int s = 0;
    for (i=0; i<n; i++) s += A[i] * B[i];
    return s;
}
```

Usando istruzioni AVX, Procediamo con il loop unrolling come mostrato nel paragrafo 2.5.1:

```
int prods_int(const int* A, const int* B, size_t n) {
    long i;
    int s[8] = { 0 };           // array s inizializzato a zero
    for (i=0; i+7<n; i+=8) {    // loop unrolling parziale di ragione 8
        s[0] += A[i] * B[i];
        s[1] += A[i+1] * B[i+1];
        s[2] += A[i+2] * B[i+2];
        s[3] += A[i+3] * B[i+3];
        s[4] += A[i+4] * B[i+4];
        s[5] += A[i+5] * B[i+5];
        s[6] += A[i+6] * B[i+6];
        s[7] += A[i+7] * B[i+7];
    }
    for (; i<n; i++) s[0] += A[i] * B[i]; // serve se 8 non divide n
    return s[0]+s[1]+s[2]+s[3]+s[4]+s[5]+s[6]+s[7];
}
```

La costruzione mostrata usa un vettore di appoggio di 8 `int` per calcolare le somme parziali, dove `s[0]` contiene le somme dei prodotti degli elementi di `A` e `B` con indice `i` tale che $i \% 8 == 0$, `s[1]` quelle degli elementi con $i \% 8 == 1$, ecc. Gli elementi rimanenti considerati nel secondo ciclo vengono accumulati in `s[0]`.

Così formulato, il codice si vettorizza direttamente come segue:

```

#include <immintrin.h>

int prods_int(const int* A, const int* B, size_t n) {
    long i;
    int s[8] __attribute__((aligned(32))) = { 0 };
    __m256i a, b, c, d = _mm256_load_si256((const __m256i*)s); // copia s in d

    for (i=0; i+7<n; i+=8) { // loop unrolling di ragione 8
        a = _mm256_loadu_si256((const __m256i*)(A+i)); // copia A[i..i+7] in a
        b = _mm256_loadu_si256((const __m256i*)(B+i)); // copia B[i..i+7] in b
        c = _mm256_mullo_epi32(a,b); // c=a*b
        d = _mm256_add_epi32(c,d); // d=d+c
    }
    _mm256_store_si256((__m256i*)s, d); // copia d in s
    for (; i<n; i++) s[0] += A[i] * B[i];
    return s[0]+s[1]+s[2]+s[3]+s[4]+s[5]+s[6]+s[7];
}

```

Poiché **non abbiamo alcuna garanzia** che gli array A e B siano allineati a indirizzi multipli di 32, dobbiamo utilizzare istruzioni load che **non richiedono allineamento** (`_mm256_loadu_si256`). Allineiamo invece l'array s usando la direttiva `__attribute__((aligned(32)))`, così possiamo usare la `_mm256_load_si256` che richiede allineamento².

Per un test, possiamo usare il seguente semplice programma di prova:

```

int main() {
    int A[17] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 };
    int B[17] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 };

    int s = prods_int((const int*)A, (const int*)B, 17);

    printf("%d\n", s); // stampa 1785
    return 0;
}

```

Poiché l'operazione `_mm256_mullo_epi32` è disponibile solo in AVX2, compiliamo con `gcc prods_int.c -O1 -mavx2 -o prods_int`.

² La cosa è tuttavia praticamente irrilevante a livello prestazionale, poiché la load da s è esterna al ciclo.

Bibliografia

[[IntelIntr](#)] The Intel Intrinsics Guide

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/> (verificato 5 gennaio 2016)

[[PosixMemalign](#)] The Open Group Base Specifications Issue 7, `posix_memalign`

http://pubs.opengroup.org/onlinepubs/9699919799/functions/posix_memalign.html

(verificato 5 gennaio 2016)

[[GNUMemalign](#)] The GNU standard library: Allocating Aligned Memory Blocks

http://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html

(verificato 5 gennaio 2016)