Vectorization and AVX2 Intrinsics Programming

UNIVERSITÀ DEGLI STUDI
DI SALERNO

Biagio Cosenza
Department of Computer Science
University of Salerno, Italy

Outline

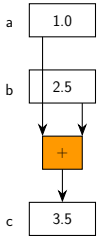
SIMD Instructions

Single Instruction, Multiple Data

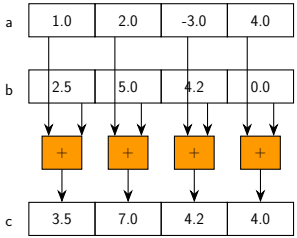
Vector instruction

- instruction that work on multiple data elements (pack) simultaneously
- typically: short, length fixed by the hardware, not necessarily fixed by the ISA

Example: `c = a + b;`



Scalar instruction



SIMD instruction

SIMD Extensions

ISA	SIMD Extension	Year
PA-RISC	Multimedia Acceleration eXtensions (MAX)	1994
SPARC	Visual Instruction Set (VIS)	1995
Alpha	Motion Video Instructions (MVI)	1996
MIPS	Multimedia Acceleration Instructions (MDMX)	1996
x86	MultiMedia eXtensions (MMX)	1996
PowerPC	Altivec	1998
X86	Streaming SIMD Extensions (SSE, SSE2, SSE3, SSE4)	1999-2007
ARM	Advanced SIMD Extension (NEON)	2005
X86	Advanced Vector eXtensions (AVX, AVX2, AVX-512)	2011-2013-2017
ARM	Scalable Vector Extension (SVE, SVE2)	2020-2021
RISC-V	RVV: RISC-V “V” Vector Extension (0.7, 1.0)	2021-2023

Intel AVX, AVX2 and AVX-512

AVX (Advanced Vector Extensions)

- introduced with Intel's Sandy Bridge (2011)
- AVX extends SIMD operations to 256-bit registers
- improving floating-point performance for workloads such as multimedia and scientific computing

AVX2

- introduced with Haswell (2013)
- AVX2 enhances AVX by adding full 256-bit integer support
- new instructions such as gather for improved memory access patterns

AVX-512

- since Intel's Skylake-X (2017)
- AVX-512 doubles the register width to 512 bits
- introduces new mask registers for finer control over operations, and specialized subsets targeting HPC, cryptography, ...

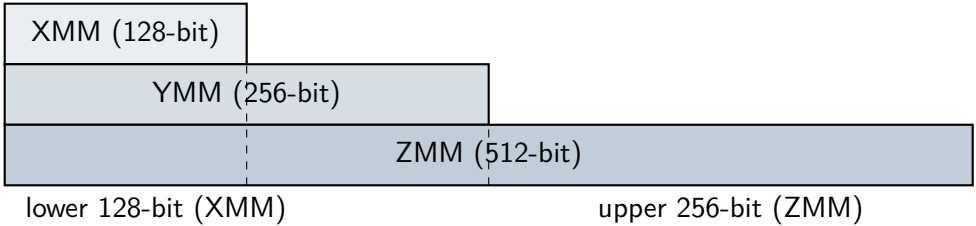
AVX Registers

Vector registers of different size, from SSE to AVX-512

SSE XMM0 — XMM15, 128-bit registers

AVX & AVX2 YMM0 — YMM15, 256-bit registers

AVX-512 ZMM0 — ZMM31, 512-bit registers



AVX/AVX2 Registers and Data Type

Vector registers can be mapped on different data types

YMM register can contain up to 256 bits of data, e.g.

- four doubles (4 x 64 bits = 256)
- eight floats (8 x 32 bits = 256)
- eight ints (8 x 32 bits = 256)

255 YMM0 0

float	float	float	float	float	float	float	float
double		double		double		double	
int32	int32	int32	int32	int32	int32	int32	int32
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8

YMM registers are versatile and can be used for both floating-point and integer operations depending on the instruction operating on them.

Programming Models for SIMD

From high to low level

Automatic vectorization

- SIMD instruction automatically generated by the compiler

SIMD-optimized libraries

- e.g., FFTW, oneMKL, oneDNN

Data parallel programming models with SIMD support

- e.g., OpenCL, SYCL, ispc, TBB, OpenMP SIMD pragma
- implicit use of vector

SIMD intrinsics

- explicit use of vector, manually write code with SIMD instruction
- leave register allocation and instruction scheduling to the compiler

Assembly

- instruction selection, scheduling and register allocation by the developer

SIMD Intrinsics Programming

Low-level C functions for SIMD programming

- an intrinsic function is mapped on one (sometime more than one) vectorial instruction

Example:

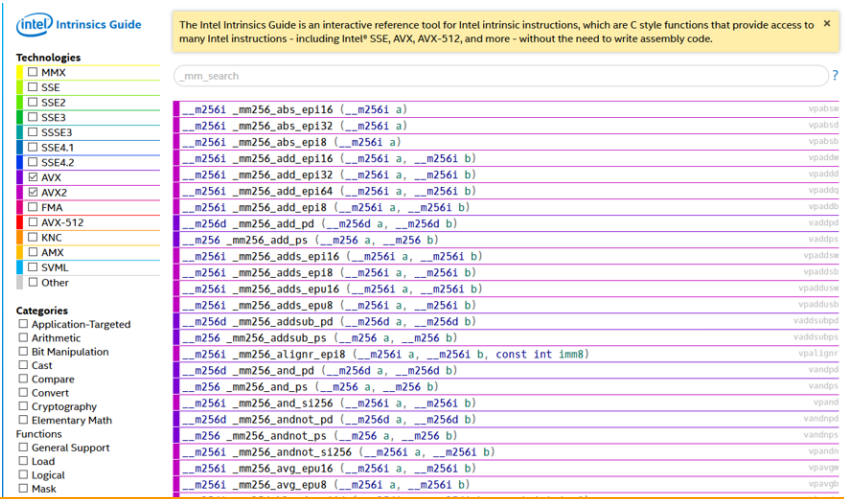
- the intrinsic function `_mm256_add_ps()` maps directly to `vaddps`

Combining the performance of assembly with the convenience of a high-level function

- register allocation and instruction scheduling performed by the compiler
- instruction selection enforced by intrinsics

Intel Intrinsics Guide

Overview: https://software.intel.com/sites/landingpage/IntrinsicsGuide



Intel Intrinsics Guide

vaddps

`__m256 _mm256_add_ps (__m256 a, __m256 b)` vaddps

Synopsis

```
__m256 _mm256_add_ps (__m256 a, __m256 b)
#include <immintrin.h>
Instruction: vaddps ymm, ymm, ymm
CPUID Flags: AVX
```

Description

Add packed single-precision (32-bit) floating-point elements in a and b, and store the results in dst.

Operation

```
FOR j := 0 to 7
        i := j*32
        dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

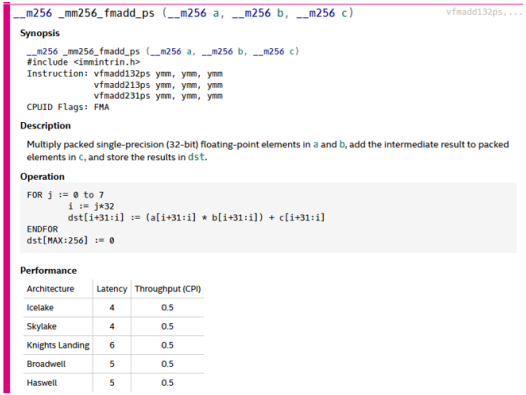
Performance

Architecture	Latency	Throughput (CPI)
Icelake	4	0.5
Skylake	4	0.5
Broadwell	3	1
Haswell	3	1
Ivy Bridge	3	1

Intrinsic to Instruction Mapping

Typically one-to-one mapping, but there are expections: the intrinsic `_mm256_fmadd_ps` maps to three instructions

- `vfmadd132ps`
- `vfmadd213ps`
- `vfmadd231ps`



`__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c)` vfmadd132ps,...

Synopsis

```
__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c)
#include <immintrin.h>
Instruction: vfmadd132ps ymm, ymm, ymm
             vfmadd213ps ymm, ymm, ymm
             vfmadd231ps ymm, ymm, ymm
CPUID Flags: FMA
```

Description

Multiply packed single-precision (32-bit) floating-point elements in a and b, add the intermediate result to packed elements in c, and store the results in dst.

Operation

```
FOR j := 0 to 7
        i := j*32
        dst[i+31:i] := (a[i+31:i] * b[i+31:i]) + c[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	4	0.5
Skylake	4	0.5
Knights Landing	6	0.5
Broadwell	5	0.5
Haswell	5	0.5

AVX/AVX2 Data Types

Each type starts with two underscores (__), an m, and the width of the vector in bits

- AVX/AVX2 registers are 256 bits, AVX512 registers are 512 bits

A vector type ending in d contains doubles; if it doesn't have a suffix, it contains floats

- _m128i and _m256i vectors contain integers
- _m256i may contain 32 **chars**, 16 **shorts**, 8 32-bit **ints**, or 4 64-bit **longs**, either signed or unsigned

__m128	128-bit vector containing 4 floats
__m128d	128-bit vector containing 2 doubles
__m128i	128-bit vector containing integers
__m256	256-bit vector containing 8 floats
__m256d	256-bit vector containing 4 doubles
__m256i	256-bit vector containing integers

AVX/AVX2 Function Naming Conventions

Function name: `_mm<bit_width>_<name>_<data_type>`

- `<bit_width>` the size of the vector returned by the function; for 128-bit vectors, this is empty; for 256-bit vectors, this is set to 256
- `<name>` describes the operation performed by the intrinsic
- `<data_type>` is the data type of the function's primary arguments

`<data_type>` identifies the content of the input values:

- `ps` packed single-precision (floats)
- `pd` packed double-precision (doubles)
- `epi8`/`epi16`/`epi32`/`epi64` contain 8-bit/16-bit/32-bit/64-bit signed integers
- `epu8`/`epu16`/`epu32`/`epu64` contain 8-bit/16-bit/32-bit/64-bit unsigned integers
- `si128`/`si256` unspecified 128-bit vector or 256-bit vector
- `m128`/`m128i`/`m128d`/`m256`/`m256i`/`m256d` identifies input vector types when they're different than the type of the returned vector

AVX Example: Vector Difference

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

int main() {
 /* Initialize the two argument vectors */
 __m256 evens = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
 __m256 odds  = _mm256_set_ps(1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);

 /* Compute the difference between the two vectors */
  __m256 result = _mm256_sub_ps(evens, odds);

 /* Display the elements of the result vector */
 float* f = (float*)\&result;
 printf("%f %f %f %f %f %f %f %f\n", f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7]);
 return 0;
}
```

Compile with specific AVX flag:

https://godbolt.org/z/nMjrvbcG4

```
> gcc -mavx -o hello_avx hello_avx.c
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
```

⚠ In `__mm256_set_ps`, the ordering of the input parameters is right-to-left

Initialization to Scalar Values

256-bit register can be arranged in different ways

- by 32- or 64-bit float
- by 8-, 16-, 32- or 64-bit integer (either signed or unsigned)

`_mm256_setzero_ps/pd`	Returns a floating-point vector filled with zero
`_mm256_setzero_si256`	Returns an integer vector whose bytes are set to zero
`_mm256_set1_ps/pd`	Fill a vector with a floating-point value
`_mm256_set1_epi8/epi16/epi32/epi64`	Fill a vector with an integer
`_mm256_set_ps/pd`	Initialize a vector with 8 floats (`ps`) or 4 doubles (`pd`)
`_mm256_set_m128/m128d/m128i`	Initialize a 256-bit vector with two 128-bit vectors
`_mm256_setr_ps/pd`	Initialize a vector with 8 floats (`ps`) or 4 doubles (`pd`) in re
`_mm256_setr_epi8/epi16/epi32/epi64`	Initialize a vector with integers in reverse order

Load from Memory

Instruction for aligned, unaligned and masked load

Different kind of load instructions

- aligned load
- unaligned load
- masked load

`_mm256_load_ps/pd`	Loads a floating-point vector from an aligne
`_mm256_load_si256`	Loads an integer vector from an aligned me
`_mm256_loadu_ps/pd`	Loads a floating-point vector from an unalig
`_mm256_loadu_si256`	Loads an integer vector from an unaligned r
`_mm_maskload_ps/pd _mm256_maskload_ps/pd`	Load portions of a 128-bit/256-bit floating-p
`_mm_maskload_epi32/64 _mm256_maskload_epi32/64`	Load portions of a 128-bit/256-bit integer v

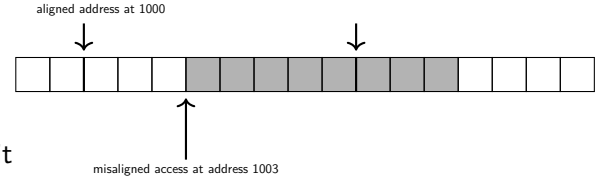
Memory Alignment

Data must be aligned at an N-byte boundary:

MMX 64-bit (8 bytes)

SSE, Altivec 128-bit (16 bytes)

AVX 256-bit (32 bytes)



Handling alignment

- software: the programmer has to do it
- OS exception: very slow
- hardware: transparent to the programmer

Aligned loads are faster because they can be fetched in a single memory access

- data is fetched directly into the SIMD register without needing additional work
- unaligned accesses could involve multiple memory accesses

The compiler aligns `__mm256` (local and global) to 32-byte boundaries

- ⚠ to align integer, float, or double arrays, you must use `__declspec(align(32))`

Memory Alignment

When loading data into vectors, memory alignment is important

- each `_mm256_load_*` intrinsic accepts a memory address that must be aligned on a 32-byte boundary, i.e., the address must be divisible by 32
- any attempt to load unaligned data with `_m256_load_*` produces a segmentation fault

Example with C11 aligned allocation `aligned_alloc` (alternative are POSIX `posix_memalign`, Windows `_aligned_malloc` and Intel `__mm_malloc` / `__mm_free`):

```
1 float* aligned_floats = (float*)aligned_alloc(32, 64 * sizeof(float));
2 //... initialize data ...
3 __m256 vec = _mm256_load_ps(aligned_floats);
```

If the data isn't aligned at a 32-bit boundary, the `_m256_loadu_*` functions should be used instead, for example:

```
1 float* unaligned_floats = (float*)malloc(64 * sizeof(float));
2 //... initialize data ...
3 __m256 vec = _mm256_loadu_ps(unaligned_floats);
```

Masked Load

Suppose you want to process a float array of 11 float in AVX

- 11 isn't divisible by 8, the last five floats of the second `__m256` vector need to be set to zero so they don't affect the computation

This selective loading can be accomplished with the `_maskload_` functions, which accept two arguments:

1. a memory address
2. an integer vector with the same number of elements as the returned vector (called *mask*). For each element in the integer vector whose *highest* bit is one, the corresponding element in the returned vector is read from memory. If the *highest bit* in the integer vector is zero, the corresponding element in the returned vector is set to zero

Masked Load Example

```
int array[8]   = { 1, 2, 3, 4, 5, 6, 7, 8 };
__m256i mask   = _mm256_setr_epi32(-2, -42, 10, 0, -100, 0, 4, 8);
__m256i result = _mm256_maskload_epi32(array, mask);

int* res = (int*)\&result;
printf("%d %d %d %d %d %d %d %d\n",
  res[0], res[1], res[2], res[3], res[4], res[5], res[6], res[7]);
```

Output:

```
1 2 0 0 5 0 0 0
```

Elements are zeroed out when the *highest bit* is not set in the corresponding mask element `setr` loads the mask in reversed order, e.g.:

- -100 is **1**1111111 11111111 11111111 10011100
- 8 is **0**0000000 00000000 00000000 00000100
- the most significant bit is 0 for non-negative number

Reversed Order in Set

_mm256_set_epi32 vs _mm256_setr_epi32

`__m256 _mm256_set_ps (float e7, float e6, float e5, float e4, float e3, float e2, float e1, float e0)`

Synopsis

```
__m256 _mm256_set_ps (float e7, float e6, float e5,
float e4, float e3, float e2, float e1, float e0)
#include <immintrin.h>
Instruction: Sequence
CPUID Flags: AVX
```

Description

Set packed single-precision (32-bit) floating-point elements in `dst` with the supplied values.

Operation

```
dst[31:0] := e0
dst[63:32] := e1
dst[95:64] := e2
dst[127:96] := e3
dst[159:128] := e4
dst[191:160] := e5
dst[223:192] := e6
dst[255:224] := e7
dst[MAX:256] := 0
```

`__m256 _mm256_setr_ps (float e7, float e6, float e5, float e4, float e3, float e2, float e1, float e0)`

Synopsis

```
__m256 _mm256_setr_ps (float e7, float e6, float
e5, float e4, float e3, float e2, float e1, float
e0)
#include <immintrin.h>
Instruction: Sequence
CPUID Flags: AVX
```

Description

Set packed single-precision (32-bit) floating-point elements in `dst` with the supplied values in reverse order.

Operation

```
dst[31:0] := e7
dst[63:32] := e6
dst[95:64] := e5
dst[127:96] := e4
dst[159:128] := e3
dst[191:160] := e2
dst[223:192] := e1
dst[255:224] := e0
dst[MAX:256] := 0
```

Store into Memory

Instruction for aligned, unaligned and masked store

Different kind of store instructions

- aligned store
- unaligned store
- masked store

_mm256_store_ps/pd	Store a floating-point vector into an align
_mm256_store_si256	Store an integer vector into an aligned m
_mm256_storeu_ps/pd	Store a floating-point vector into an unal
_mm256_storeu_si256	Store an integer vector into an unaligned
_mm_maskstore_ps/pd _mm256_maskstore_ps/pd	Store portions of a 128-bit/256-bit floati
_mm_maskstore_epi32/64 _mm256_maskstore_epi32/64	Store portions of a 128-bit/256-bit intege

Arithmetic Intrinsics

Addition, subtraction and saturated operations

Packed add and substract, with saturated version for integers.

`_mm256_add_ps/pd`	Add two floating-point vectors
`_mm256_sub_ps/pd`	Subtract two floating-point vectors
`_mm256_add_epi8/16/32/64`	Add two integer vectors
`_mm236_sub_epi8/16/32/64`	Subtract two integer vectors
`_mm256_adds_epi8/16` `_mm256_adds_epu8/16`	Add two integer vectors with saturation
`_mm256_subs_epi8/16` `_mm256_subs_epu8/16`	Subtract two integer vectors with saturation

The `s` stands for saturated arithmetic: `_adds_` / `_subs_`

- limited to a fixed range between a minimum and maximum value
- clamp the result to the minimum/maximum value that can be stored

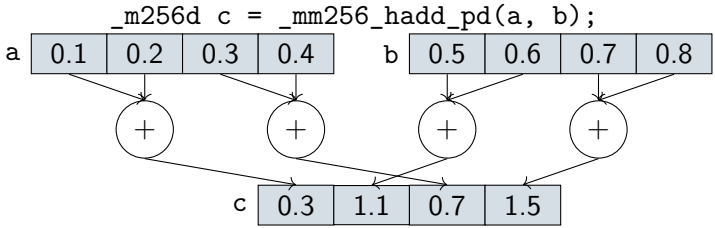
Horizontal Addition and Subtraction

Horizontal functions `_hadd_` / `_hsub_` perform addition and subtraction horizontally

- add or subtract adjacent elements within each vector
- results are stored in an interleaved fashion

Example: `_mm256_hadd_pd` horizontally adds double vectors a and b

- horizontally add adjacent pairs of double-precision (64-bit) floating-point elements in a and b, and pack the results in dst



AVX Example: Horizontal Addition

```c
#include <immintrin.h>
#include <stdio.h>
int main() {
  __m256 vec_a = _mm256_setr_pd(0.1, 0.2, 0.3, 0.4);
  __m256 vec_b = _mm256_setr_pd(0.5, 0.6, 0.7, 0.8);

  /* Compute the horizontal add between the two vectors */
  __m256 result = _mm256_hadd_pd(vec_a, vec_b);

  /* Display the elements of the result vector */
  double* d = (double*)&result;
  printf("%f %f %f %f \n", d[0], d[1], d[2], d[3]);
  return 0;
}
```

Compile with specific AVX flag:

https://godbolt.org/z/edTs13je9

```
> gcc -mavx horizontal_add.c
0.300000 1.100000 0.700000 1.500000
```

Multiplication and Division

For floating point and integer

Floating point multiplication (`ps` and `pd`) works like other arithmetic instructions.
However, integer multiplication is subject to overflow

`_mm256_mul_ps/pd`	Multiply two floating-point vectors
`_mm256_mul_epi32/epu32`	Multiply the lowest four elements of vectors containing 32-bit integers
`_mm256_mullo_epi16/32`	Multiply integers and store low halves
`_mm256_mulhi_epi16/epu16`	Multiply integers and store high halves
`_mm256_mulhrs_epi16`	Multiply 16-bit elements to form 32-bit elements
`_mm256_div_ps/pd`	Divide two floating-point vectors

⚠ If you multiply 2 N-bit numbers, the results can take up to 2N bits
Low and high bits:

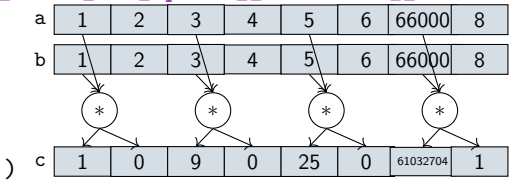
- `_mullo_` multiply every element but only store the low half of each product
- `_mulhi_` multiply every element but only store the high half of each product
- ⚠ there is no `mulhi_epi32`

Low and High with Integer Multiplication

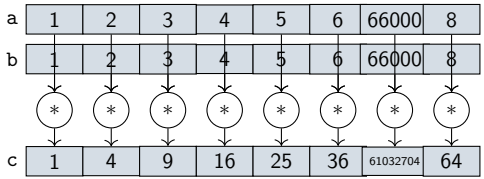
mul vs mullo

⚠ `_mm256_mul_epi32` only multiply the low signed 32-bit integers from each packed 64-bit element, and store the signed 64-bit in the result.

`_mm256_mul_epi32(__m256i a, __m256i b)`



`_mm256_mullo_epi32(__m256i a, __m256i b)`



Note that in C/C++, integer overflow is

- unsigned integer: modulo power of two
- signed integer: undefined behavior

Example: Low and High with Integer Multiplication

```c
#include <immintrin.h>
#include <stdio.h>
int main() {
  // Initialize the two integer vectors
  __m256i a = _mm256_setr_epi32(1, 2, 3, 4, 5, 6, 66000, 8);
  __m256i b = _mm256_setr_epi32(1, 2, 3, 4, 5, 6, 66000, 8);

  // Compute multiply with mullow and mul
  __m256i lo = _mm256_mullo_epi32(a, b);
  __m256i hi = _mm256_mul_epi32(a, b);

  // Print results, note overflow on the 7th element
  int32_t* i = (int32_t*)&hi;
  printf("%d %d %d %d %d %d %d %d\n", i[0], i[1], i[2], i[3], i[4], i[5], i[6], i[7]);
  i = (int32_t*)&lo;
  printf("%d %d %d %d %d %d %d %d\n", i[0], i[1], i[2], i[3], i[4], i[5], i[6], i[7]);
  return 0;
}
```

Compile with specific AVX flag:

https://godbolt.org/z/MEYPb8YaY

```
> gcc -mavx2 mullo_add.c
1 0 9 0 25 0 61032704 1
1 4 9 16 25 36 61032704 64
```

FMA: Fused Multiply Add Instruction

FMA provides greater speed and accuracy than performing multiplication and addition separately. In particular, for accuracy

- when you multiply two floating-point values, a and b, the result is really `round(a*b)`, where `round(x)` returns the floating-point value closest to x
- with FMA: instead of `round(round(a*b)+c)`, they return `round(a*b+c)`

`_mm_fmadd_ps/pd` `_mm256_fmadd_ps/pd`	Multiply two vectors and add the product to a third vector (`res = a * b + c`)
`_mm_fmsub_ps/pd` `_mm256_fmsub_ps/pd`	Multiply two vectors and subtract a vector from the product (`res = a * b - c`)

FMA Example

Example without FMA support

```
void multiply_and_add(const float*a, const float*b, const float*c, float*d) {
    for(int i=0; i<8; i++) {
        d[i] = a[i] * b[i];
        d[i] = d[i] + c[i];
    }
}
```

Example with AVX FMA

```
__m256 multiply_and_add(__m256 a, __m256 b, __m256 c) {
    return _mm256_fmadd_ps(a, b, c);
}
```

Exercise: AVX2 Intrinsics Programming

Write a C/C++ program that

1. implements a function `vec_add(float *a, float *b, float *c, int size)` that add the elements of two arrays and writes the results in a third array of the same size. You can assume that the input is not aligned and the size a multiple of 8.
2. implements a function `vec_add_aligned` , which is the same as of (1), but assume the arrays are aligned
3. implements a function `vec_add_anysize`, which is the same of (2), but support input/output arrays of any size.

For all three functions, evaluate the performance against a scalar code compiler in `-O0` and a scalar code compiled in `-O3`.