Topik : Simple Instruction Multiple Data

| **Data Type** | **Description** |
| --- | --- |
| \_\_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 |

\_mm<bit\_width>\_<name>\_<data\_type>

The parts of this format are given as follows:

1. <bit\_width> identifies 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.
2. <name> describes the operation performed by the intrinsic
3. <data\_type> identifies the data type of the function's primary arguments

This last part, <data\_type>, is a little complicated. It identifies the content of the input values, and can be set to any of the following values:

* ps - vectors contain floats (ps stands for packed single-precision)
* pd - vectors contain doubles (pd stands for packed double-precision)
* epi8/epi16/epi32/epi64 - vectors contain 8-bit/16-bit/32-bit/64-bit signed integers
* epu8/epu16/epu32/epu64 - vectors 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

| **Function** | **Description** |
| --- | --- |
| \_mm256\_setzero\_ps/pd | Returns a floating-point vector filled with zeros |
| \_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 \_mm256\_set1\_epi32/epi64 | Fill a vector with an integer |
| \_mm256\_set\_ps/pd | Initialize a vector with eight floats (ps) or four doubles (pd) |
| \_mm256\_set\_epi8/epi16 \_mm256\_set\_epi32/epi64 | Initialize a vector with integers |
| \_mm256\_set\_m128/m128d/ \_mm256\_set\_m128i | Initialize a 256-bit vector with two 128-bit vectors |
| \_mm256\_setr\_ps/pd | Initialize a vector with eight floats (ps) or four doubles (pd) in reverse order |
| \_mm256\_setr\_epi8/epi16 \_mm256\_setr\_epi32/epi64 | Initialize a vector with integers in reverse order |

| **Data Type** | **Description** |
| --- | --- |
| \_mm256\_load\_ps/pd | Loads a floating-point vector from an  aligned memory address |
| \_mm256\_load\_si256 | Loads an integer vector from an aligned memory address |
| \_mm256\_loadu\_ps/pd | Loads a floating-point vector from an  unaligned memory address |
| \_mm256\_loadu\_si256 | Loads an integer vector from an unaligned memory address |
| \_mm\_maskload\_ps/pd \_mm256\_maskload\_ps/pd | Load portions of a 128-bit/256-bit floating-point vector according to a mask |
| (2)\_mm\_maskload\_epi32/64 (2)\_mm256\_maskload\_epi32/64 | Load portions of a 128-bit/256-bit integer vector according to a mas |

Data TypeDescription\_mm256\_add\_ps/pdAdd two floating-point vectors\_mm256\_sub\_ps/pdSubtract two floating-point vectors(2)\_mm256\_add\_epi8/16/32/64 Add two integer vectors(2)\_mm236\_sub\_epi8/16/32/64Subtract two integer vectors(2)\_mm256\_adds\_epi8/16  
(2)\_mm256\_adds\_epu8/16 Add two integer vectors with saturation(2)\_mm256\_subs\_epi8/16  
(2)\_mm256\_subs\_epu8/16

Subtract two integer vectors with saturation

\_mm256\_hadd\_ps/pdAdd two floating-point vectors horizontally\_mm256\_hsub\_ps/pdSubtract two floating-point vectors horizontally(2)\_mm256\_hadd\_epi16/32Add two integer vectors horizontally(2)\_mm256\_hsub\_epi16/32Subtract two integer vectors horizontally(2)\_mm256\_hadds\_epi16Add two vectors containing shorts horizontally with saturation(2)\_mm256\_hsubs\_epi16Subtract two vectors containing shorts horizontally with saturation\_mm256\_addsub\_ps/pdAdd and subtract two floating-point vectors

Data TypeDescription\_mm256\_mul\_ps/pdMultiply two floating-point vectors(2)\_mm256\_mul\_epi32/  
(2)\_mm256\_mul\_epu32 Multiply the lowest four elements of vectors containing 32-bit integers(2)\_mm256\_mullo\_epi16/32Multiply integers and store low halves(2)\_mm256\_mulhi\_epi16/  
(2)\_mm256\_mulhi\_epu16

Multiply integers and store high halves

(2)\_mm256\_mulhrs\_epi16Multiply 16-bit elements to form 32-bit elements\_mm256\_div\_ps/pdDivide two floating-point vectors

Data TypeDescription(2)\_mm\_fmadd\_ps/pd/  
(2)\_mm256\_fmadd\_ps/pdMultiply two vectors and add the product to a third (res = a \* b + c)(2)\_mm\_fmsub\_ps/pd/  
(2)\_mm256\_fmsub\_ps/pd

Multiply two vectors and subtract a vector from the product (res = a \* b - c)

(2)\_mm\_fmadd\_ss/sd Multiply and add the lowest element in the vectors (res[0] = a[0] \* b[0] + c[0])(2)\_mm\_fmsub\_ss/sdMultiply and subtract the lowest element in the vectors (res[0] = a[0] \* b[0] - c[0])(2)\_mm\_fnmadd\_ps/pd  
(2)\_mm256\_fnmadd\_ps/pd

Multiply two vectors and add the negated product to a third (res = -(a \* b) + c)

(2)\_mm\_fnmsub\_ps/pd/  
(2)\_mm256\_fnmsub\_ps/pdMultiply two vectors and add the negated product to a third (res = -(a \* b) - c)(2)\_mm\_fnmadd\_ss/sdMultiply the two lowest elements and add the negated product to the lowest element of the third vector (res[0] = -(a[0] \* b[0]) + c[0])(2)\_mm\_fnmsub\_ss/sdMultiply the lowest elements and subtract the lowest element of the third vector from the negated product (res[0] = -(a[0] \* b[0]) - c[0])(2)\_mm\_fmaddsub\_ps/pd/  
(2)\_mm256\_fmaddsub\_ps/pdMultiply two vectors and alternately add and subtract from the product (res = a \* b - c)(2)\_mm\_fmsubadd\_ps/pd/  
(2)\_mmf256\_fmsubadd\_ps/pd Multiply two vectors and alternately subtract and add from the product (res = a \* b - c)

| **Data Type** | **Description** |
| --- | --- |
| \_mm\_permute\_ps/pd/ \_mm256\_permute\_ps/pd | Select elements from the input vector based on an 8-bit control value |
| (2)\_mm256\_permute4x64\_pd/ (2)\_mm256\_permute4x64\_epi64 | Select 64-bit elements from the input vector based on an 8-bit control value |
| \_mm256\_permute2f128\_ps/pd | Select 128-bit chunks from two input vectors based on an 8-bit control value |
| \_mm256\_permute2f128\_si256 | Select 128-bit chunks from two input vectors based on an 8-bit control value |
| \_mm\_permutevar\_ps/pd \_mm256\_permutevar\_ps/pd | Select elements from the input vector based on bits in an integer vector |
| (2)\_mm256\_permutevar8x32\_ps/ (2)\_mm256\_permutevar8x32\_epi32 | Select 32-bit elements (floats and ints) using indices in an integer vector |

| **Data Type** | **Description** |
| --- | --- |
| \_mm256\_shuffle\_ps/pd | Select floating-point elements according to an 8-bit value |
| \_mm256\_shuffle\_epi8/ \_mm256\_shuffle\_epi32 | Select integer elements according to an 8-bit value |
| (2)\_mm256\_shufflelo\_epi16/   (2)\_mm256\_shufflehi\_epi16 | Select 128-bit chunks from two input vectors based on an 8-bit control value |

* gcc -mavx -o hello\_avx hello\_avx.c
* gcc -mavx2 -o mask\_load mask\_load.c
* gcc -mfma -o fmatest fmatest.c
* gcc -mavx -o complex\_mult complex\_mult.c

CONCLUSION:

The scalar\_search function uses a simple loop to sequentially search through the array, comparing each element to the target value until a match is found. This approach has a time complexity of O(n), where n is the size of the array. This means that as the size of the array increases, the time taken to search the array increases linearly. The memory usage is also minimal, as only the input array and a few variables are needed.

On the other hand, the simd\_search function uses SIMD instructions to search the array in parallel. This approach can significantly reduce the search time for large arrays, as multiple elements are compared simultaneously. The time complexity of the SIMD approach is O(n/B), where B is the block size (4 in this case), which means that the time taken to search the array increases much slower as the size of the array increases. However, the SIMD approach requires additional memory for storing the SSE registers and loading the array into the registers. This can result in increased memory usage compared to the scalar approach.

In general, the SIMD approach can offer significant performance gains for large arrays, but may not be worth the extra memory usage and complexity for small arrays. The scalar approach is simple and efficient for small arrays, but may become impractical for very large arrays due to its linear time complexity.

A picture containing graphical user interface

Description automatically generated

A screenshot of a computer screen

Description automatically generated with low confidence

Text

Description automatically generated