南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# C/C++ Program Design

## Lab 8, SIMD and OpenMP

于仕琪，廖琪梅

intel

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

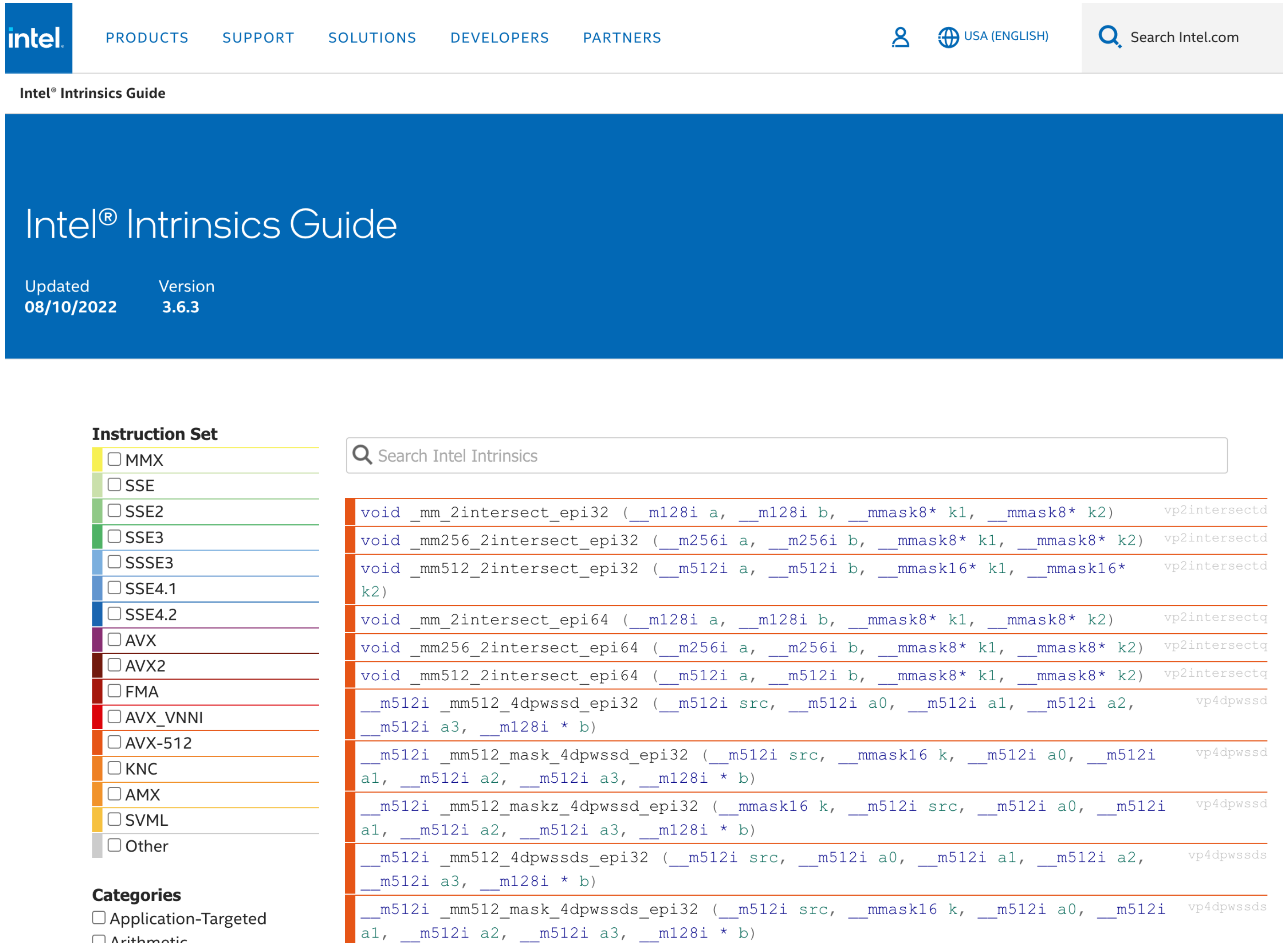
# Intel Intrinsics

## SIMD@Intel

- MMX: 1997, 8 registers, 64 bits,
- SSE (Streaming SIMD Extensions): 1999, 128 bits
- SSE2: 2000
- SSE3: 2004
- SSSE3: 2006
- SSE4.1: 2006
- SSE4.2
- AVX (Advanced Vector Extensions): 2011, 256 bits
- AVX2: 2013
- AVX-512: 2016

# Intel® Intrinsics Guide

- https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html



Intel® Intrinsics Guide

Intel® Intrinsics Guide

Updated 08/10/2022

Version 3.6.3

**Instruction Set**

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX_VNNI
- ☐ AVX-512
- ☐ KNC
- ☐ AMX
- ☐ SVML
- ☐ Other

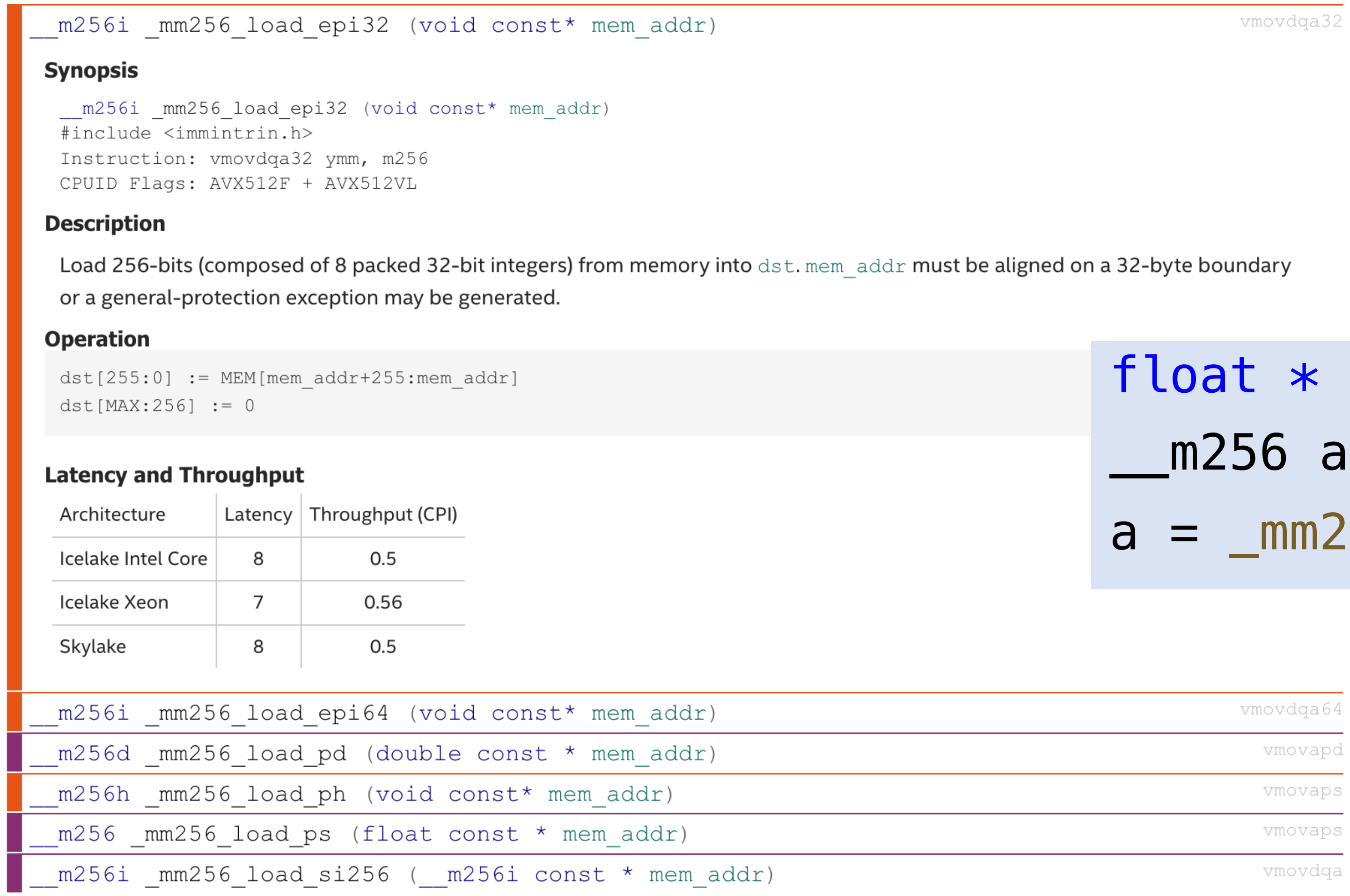
**Categories**

- ☐ Application-Targeted
- ☐ Arithmetic

Search Intel Intrinsics

```
void _mm_2intersect_epi32 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)          vp2intersectd
void _mm256_2intersect_epi32 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)       vp2intersectd
void _mm512_2intersect_epi32 (__m512i a, __m512i b, __mmask16* k1, __mmask16*         vp2intersectd
k2)
void _mm_2intersect_epi64 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)          vp2intersectq
void _mm256_2intersect_epi64 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)       vp2intersectq
void _mm512_2intersect_epi64 (__m512i a, __m512i b, __mmask8* k1, __mmask8* k2)       vp2intersectq
__m512i _mm512_4dpwssd_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2,           vp4dpwssd
__m512i a3, __m128i * b)
__m512i _mm512_mask_4dpwssd_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i         vp4dpwssd
a1, __m512i a2, __m512i a3, __m128i * b)
__m512i _mm512_maskz_4dpwssd_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i        vp4dpwssd
a1, __m512i a2, __m512i a3, __m128i * b)
__m512i _mm512_4dpwssds_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2,         vp4dpwssds
__m512i a3, __m128i * b)
__m512i _mm512_mask_4dpwssds_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i       vp4dpwssds
a1, __m512i a2, __m512i a3, __m128i * b)
```

# Load data from memory to registers



```
__m256i _mm256_load_epi32 (void const* mem_addr)                                    vmovdqa32
```

**Synopsis**

```
__m256i _mm256_load_epi32 (void const* mem_addr)
#include <immintrin.h>
Instruction: vmovdqa32 ymm, m256
CPUID Flags: AVX512F + AVX512VL
```

**Description**

Load 256-bits (composed of 8 packed 32-bit integers) from memory into `dst`. `mem_addr` must be aligned on a 32-byte boundary or a general-protection exception may be generated.

**Operation**

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

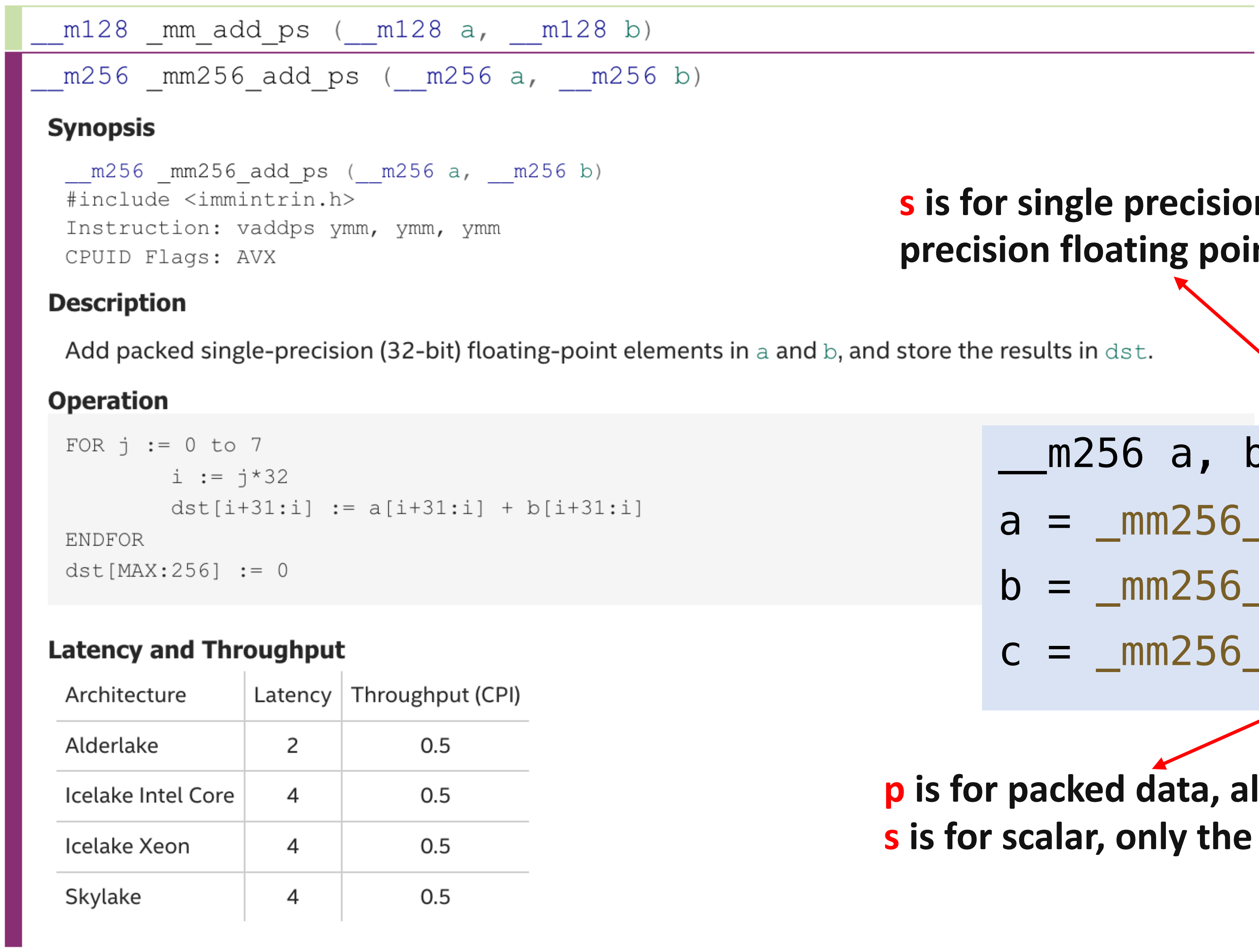
**Latency and Throughput**

| Architecture | Latency | Throughput (CPI) |
|---|---|---|
| Icelake Intel Core | 8 | 0.5 |
| Icelake Xeon | 7 | 0.56 |
| Skylake | 8 | 0.5 |

```
__m256i _mm256_load_epi64 (void const* mem_addr)                                    vmovdqa64
__m256d _mm256_load_pd (double const * mem_addr)                                    vmovapd
__m256h _mm256_load_ph (void const* mem_addr)                                       vmovaps
__m256 _mm256_load_ps (float const * mem_addr)                                      vmovaps
__m256i _mm256_load_si256 (__m256i const * mem_addr)                                vmovdqa
```

```
float * p = ...;
__m256 a;
a = _mm256_load_ps(p);
```

# Add operation



```
__m128 _mm_add_ps (__m128 a, __m128 b)
__m256 _mm256_add_ps (__m256 a, __m256 b)
```

**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
```

**Latency and Throughput**

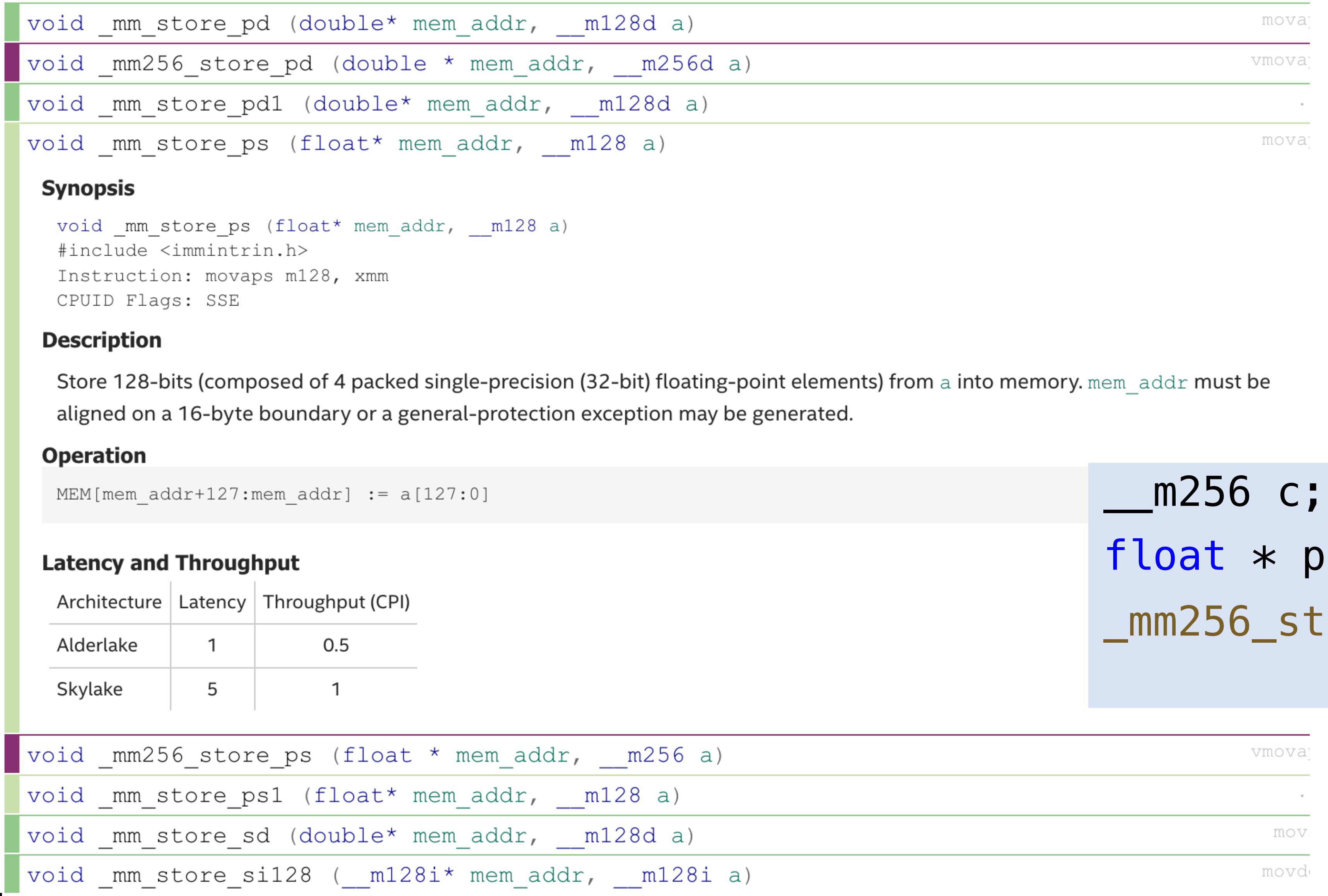
| Architecture | Latency | Throughput (CPI) |
|---|---|---|
| Alderlake | 2 | 0.5 |
| Icelake Intel Core | 4 | 0.5 |
| Icelake Xeon | 4 | 0.5 |
| Skylake | 4 | 0.5 |

**s is for single precision floating point (float); d is for double precision floating point (double)**

```
__m256 a, b, c;
a = _mm256_load_ps(p1 + i);
b = _mm256_load_ps(p2 + i);
c = _mm256_add_ps(a, b);
```

**p is for packed data, all scalars will be in the operation.**
**s is for scalar, only the first scaler will be involved.**

# Store data from registers to memory



```
void _mm_store_pd (double* mem_addr, __m128d a)                  mova
void _mm256_store_pd (double * mem_addr, __m256d a)             vmova
void _mm_store_pd1 (double* mem_addr, __m128d a)                    .
void _mm_store_ps (float* mem_addr, __m128 a)                    mova
```

**Synopsis**

```
void _mm_store_ps (float* mem_addr, __m128 a)
#include <immintrin.h>
Instruction: movaps m128, xmm
CPUID Flags: SSE
```

**Description**

Store 128-bits (composed of 4 packed single-precision (32-bit) floating-point elements) from `a` into memory. `mem_addr` must be aligned on a 16-byte boundary or a general-protection exception may be generated.

**Operation**

```
MEM[mem_addr+127:mem_addr] := a[127:0]
```

**Latency and Throughput**

| Architecture | Latency | Throughput (CPI) |
|---|---|---|
| Alderlake | 1 | 0.5 |
| Skylake | 5 | 1 |

```
void _mm256_store_ps (float * mem_addr, __m256 a)               vmova
void _mm_store_ps1 (float* mem_addr, __m128 a)                      .
void _mm_store_sd (double* mem_addr, __m128d a)                   mov
void _mm_store_si128 (__m128i* mem_addr, __m128i a)              movd
```

```
__m256 c;
float * p = ...;
_mm256_store_ps(p, c);
```

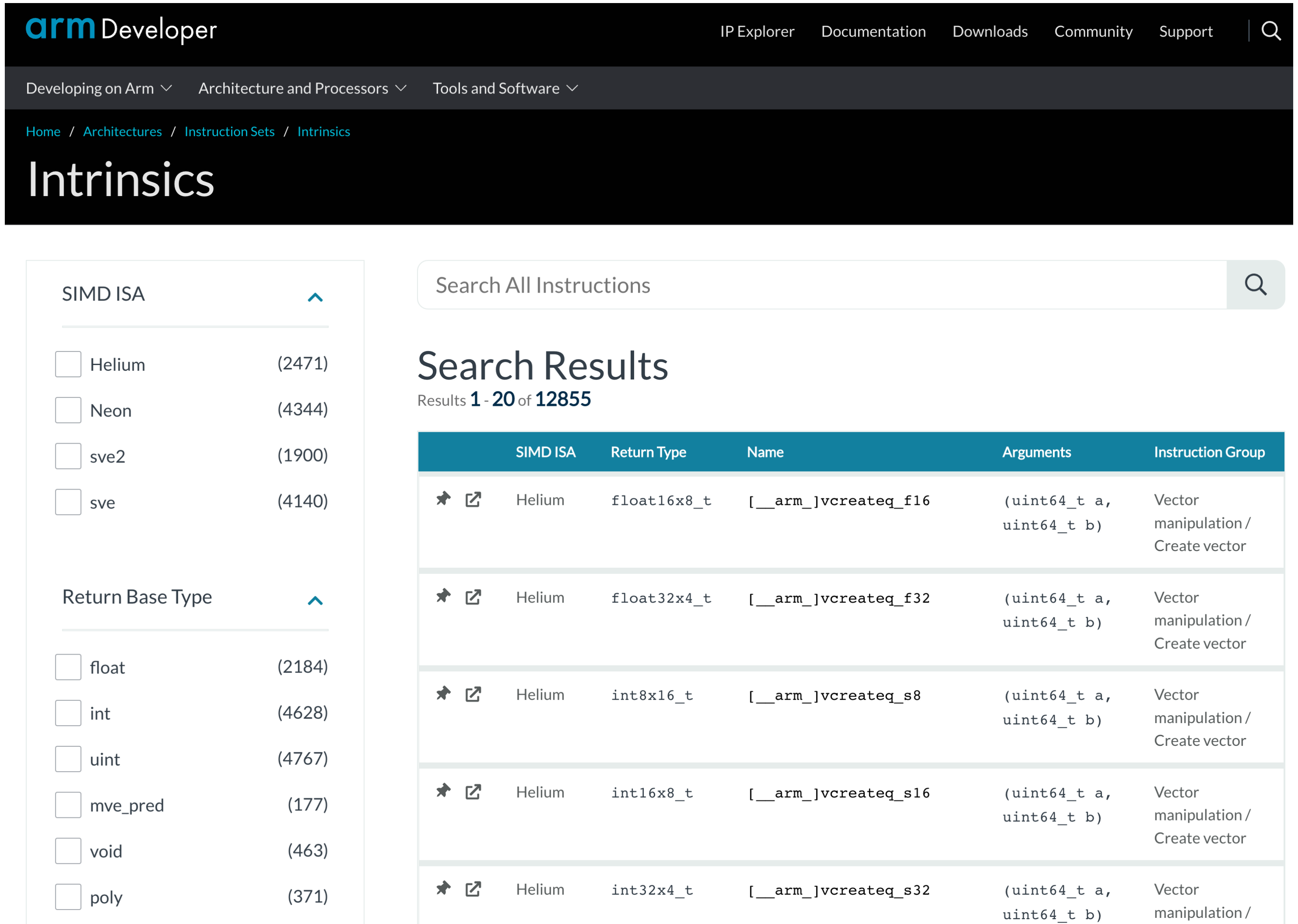
# ARM Neon Intrinsics

## SIMD@ARM

- Neon: 64 bits and 128 bits
- Helium (or MVE): More instructions
- SVE (Scalable Vector Extension):  128 bits to 2048 bits
- SVE2

# ARM Intrinsics

- https://developer.arm.com/architectures/instruction-sets/intrinsics/



Intrinsics

SIMD ISA

| | |
|---|---|
| Helium | (2471) |
| Neon | (4344) |
| sve2 | (1900) |
| sve | (4140) |

Return Base Type

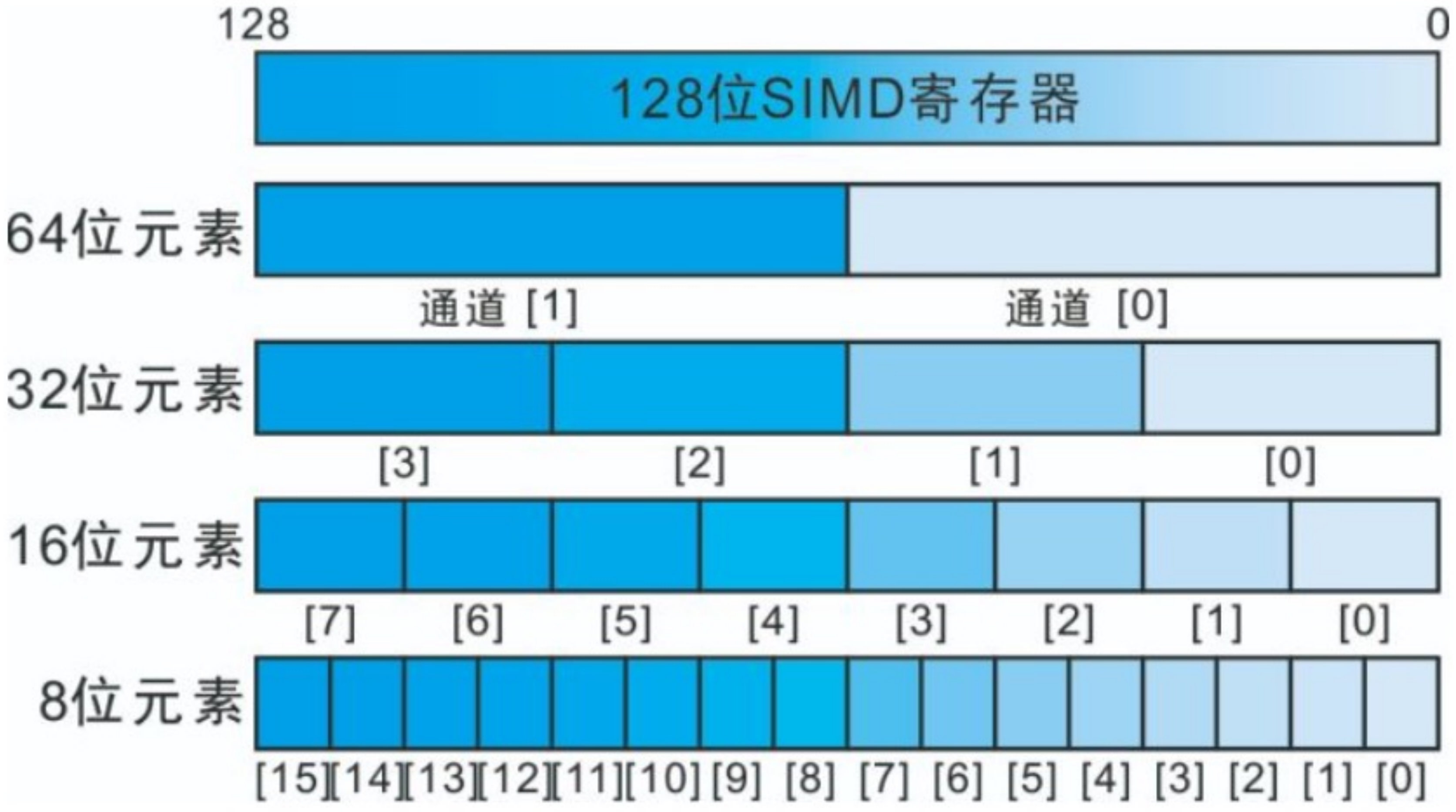
| | |
|---|---|
| float | (2184) |
| int | (4628) |
| uint | (4767) |
| mve_pred | (177) |
| void | (463) |
| poly | (371) |

Search Results

Results 1 - 20 of 12855

| SIMD ISA | Return Type | Name | Arguments | Instruction Group |
|---|---|---|---|---|
| Helium | float16x8_t | [__arm_]vcreateq_f16 | (uint64_t a, uint64_t b) | Vector manipulation / Create vector |
| Helium | float32x4_t | [__arm_]vcreateq_f32 | (uint64_t a, uint64_t b) | Vector manipulation / Create vector |
| Helium | int8x16_t | [__arm_]vcreateq_s8 | (uint64_t a, uint64_t b) | Vector manipulation / Create vector |
| Helium | int16x8_t | [__arm_]vcreateq_s16 | (uint64_t a, uint64_t b) | Vector manipulation / Create vector |
| Helium | int32x4_t | [__arm_]vcreateq_s32 | (uint64_t a, uint64_t b) | Vector manipulation / |

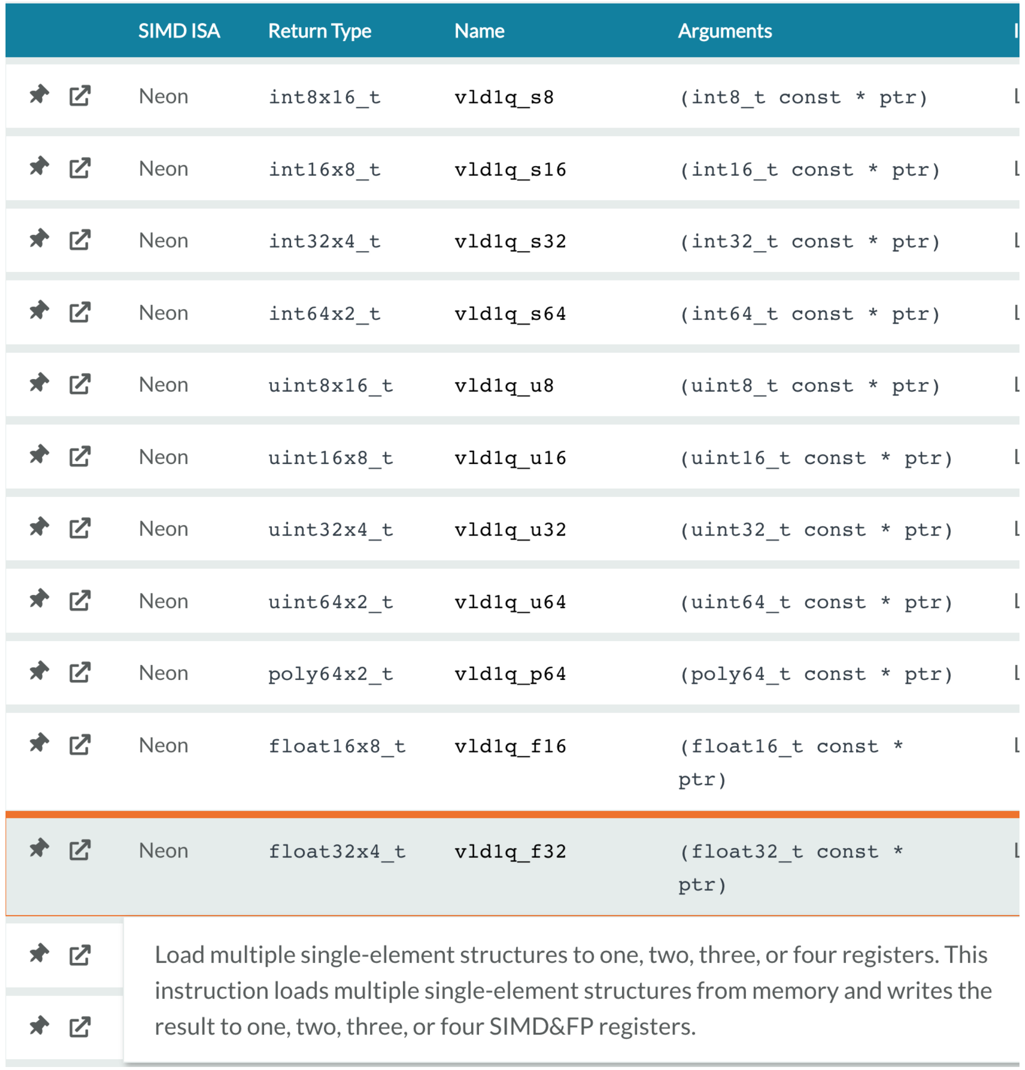
The NEON Intrinsics can operator on 128-bit registers



SIMD register

https://manzp.blog.csdn.net/article/details/114686930

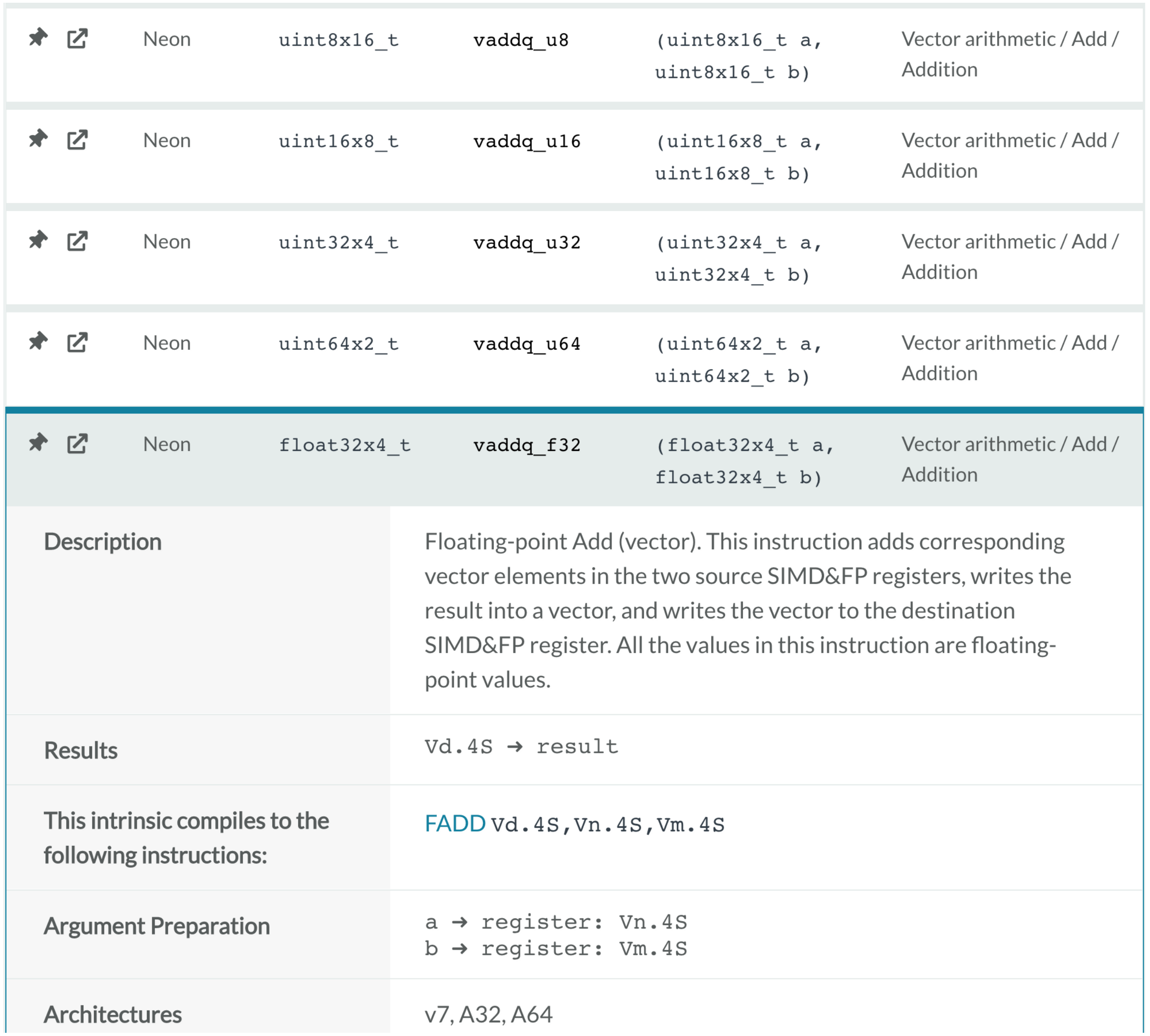
# Load data from memory to registers



| | SIMD ISA | Return Type | Name | Arguments | I |
|---|---|---|---|---|---|
| | Neon | int8x16_t | vld1q_s8 | (int8_t const * ptr) | l |
| | Neon | int16x8_t | vld1q_s16 | (int16_t const * ptr) | l |
| | Neon | int32x4_t | vld1q_s32 | (int32_t const * ptr) | l |
| | Neon | int64x2_t | vld1q_s64 | (int64_t const * ptr) | l |
| | Neon | uint8x16_t | vld1q_u8 | (uint8_t const * ptr) | l |
| | Neon | uint16x8_t | vld1q_u16 | (uint16_t const * ptr) | l |
| | Neon | uint32x4_t | vld1q_u32 | (uint32_t const * ptr) | l |
| | Neon | uint64x2_t | vld1q_u64 | (uint64_t const * ptr) | l |
| | Neon | poly64x2_t | vld1q_p64 | (poly64_t const * ptr) | l |
| | Neon | float16x8_t | vld1q_f16 | (float16_t const * ptr) | l |
| | Neon | float32x4_t | vld1q_f32 | (float32_t const * ptr) | l |

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

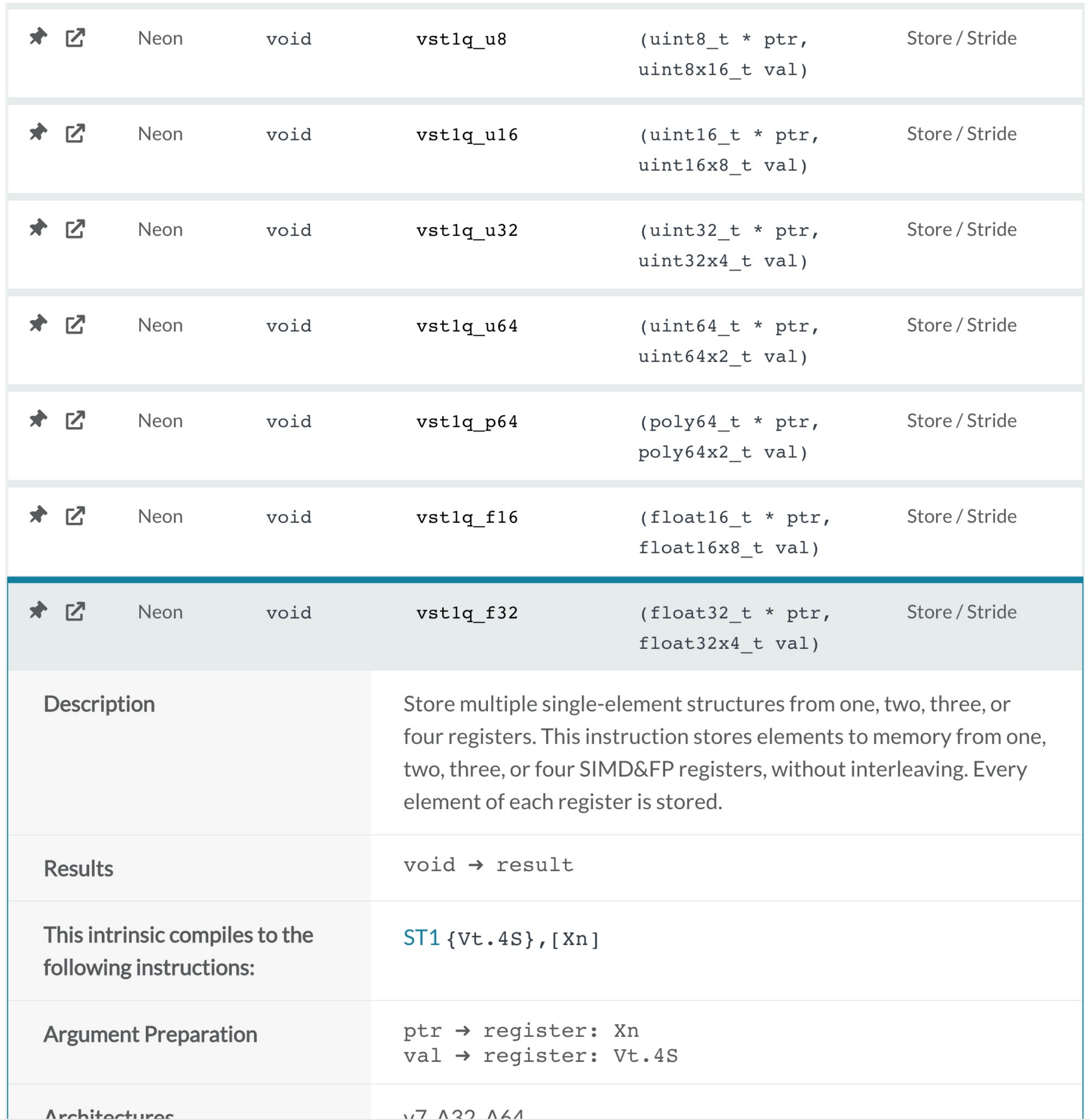
# Add operation



| | | | | | |
|---|---|---|---|---|---|
| Neon | uint8x16_t | vaddq_u8 | (uint8x16_t a, uint8x16_t b) | Vector arithmetic / Add / Addition |
| Neon | uint16x8_t | vaddq_u16 | (uint16x8_t a, uint16x8_t b) | Vector arithmetic / Add / Addition |
| Neon | uint32x4_t | vaddq_u32 | (uint32x4_t a, uint32x4_t b) | Vector arithmetic / Add / Addition |
| Neon | uint64x2_t | vaddq_u64 | (uint64x2_t a, uint64x2_t b) | Vector arithmetic / Add / Addition |
| Neon | float32x4_t | vaddq_f32 | (float32x4_t a, float32x4_t b) | Vector arithmetic / Add / Addition |

| | |
|---|---|
| Description | Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values. |
| Results | `Vd.4S → result` |
| This intrinsic compiles to the following instructions: | FADD `Vd.4S,Vn.4S,Vm.4S` |
| Argument Preparation | `a → register: Vn.4S`<br>`b → register: Vm.4S` |
| Architectures | v7, A32, A64 |

# Store data from registers to memory

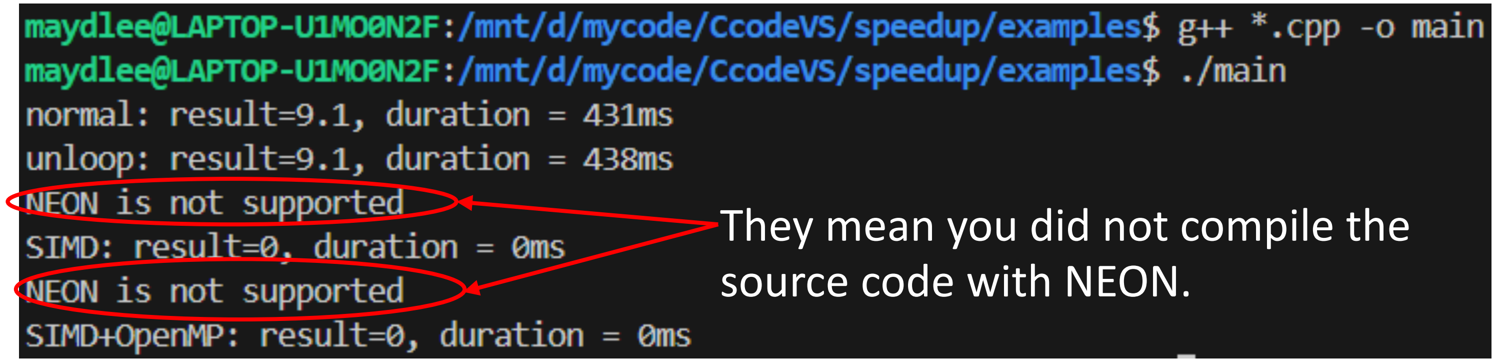


| | | | | |
|---|---|---|---|---|
| Neon | void | vst1q_u8 | (uint8_t * ptr, uint8x16_t val) | Store / Stride |
| Neon | void | vst1q_u16 | (uint16_t * ptr, uint16x8_t val) | Store / Stride |
| Neon | void | vst1q_u32 | (uint32_t * ptr, uint32x4_t val) | Store / Stride |
| Neon | void | vst1q_u64 | (uint64_t * ptr, uint64x2_t val) | Store / Stride |
| Neon | void | vst1q_p64 | (poly64_t * ptr, poly64x2_t val) | Store / Stride |
| Neon | void | vst1q_f16 | (float16_t * ptr, float16x8_t val) | Store / Stride |
| Neon | void | vst1q_f32 | (float32_t * ptr, float32x4_t val) | Store / Stride |

| | |
|---|---|
| Description | Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored. |
| Results | `void → result` |
| This intrinsic compiles to the following instructions: | ST1 `{Vt.4S},[Xn]` |
| Argument Preparation | `ptr → register: Xn`<br>`val → register: Vt.4S` |
| Architectures | v7 A32 A64 |

## Some tips for the example of Week 8

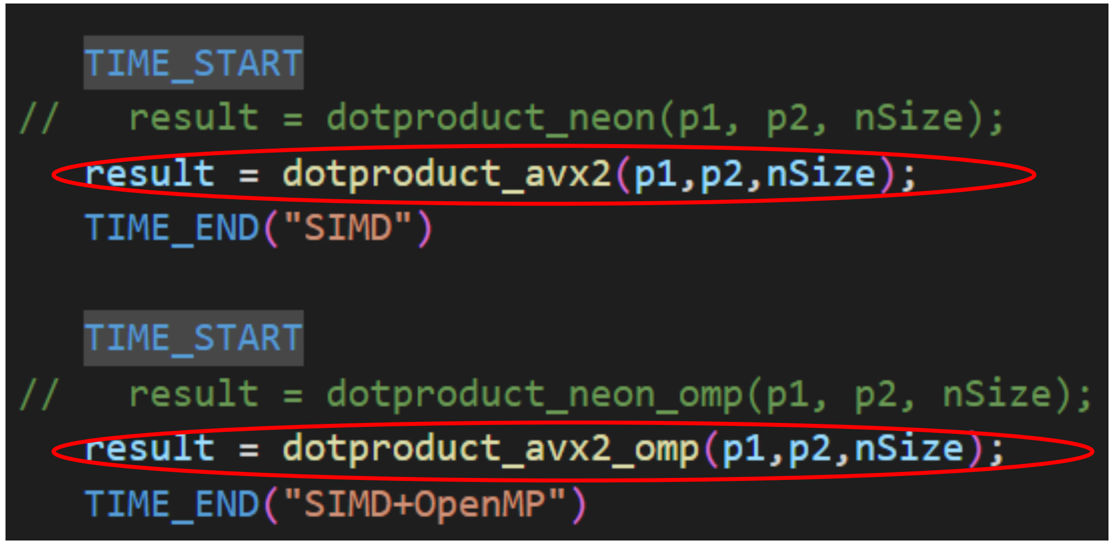
If you compile the source code using “g++ *.cpp –o main”



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ g++ *.cpp -o main
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ ./main
normal: result=9.1, duration = 431ms
unloop: result=9.1, duration = 438ms
NEON is not supported
SIMD: result=0, duration = 0ms
NEON is not supported
SIMD+OpenMP: result=0, duration = 0ms
```

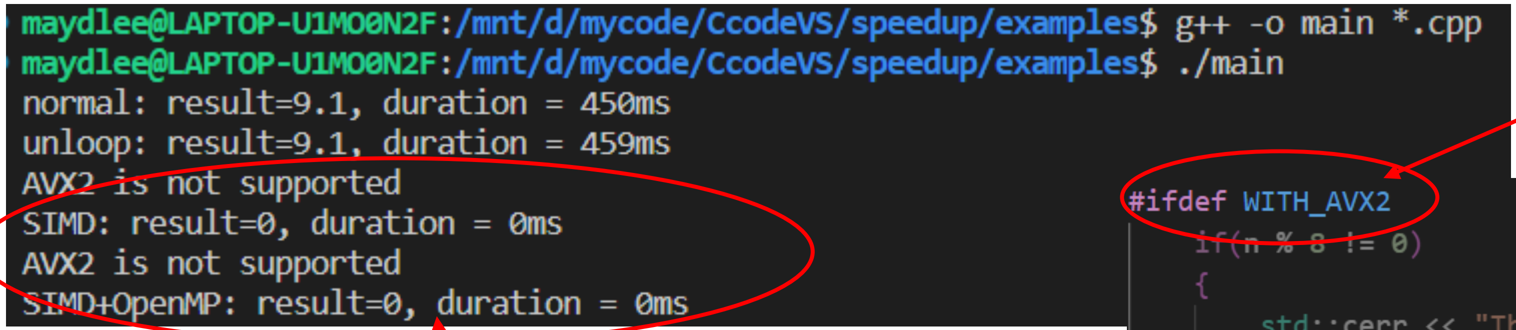
They mean you did not compile the source code with NEON.

If you run the example at Intel CPU, please enable the function call of dotproduct_avx2() in main.cpp.



```
   TIME_START
//   result = dotproduct_neon(p1, p2, nSize);
   result = dotproduct_avx2(p1,p2,nSize);
   TIME_END("SIMD")

   TIME_START
//   result = dotproduct_neon_omp(p1, p2, nSize);
   result = dotproduct_avx2_omp(p1,p2,nSize);
   TIME_END("SIMD+OpenMP")
```

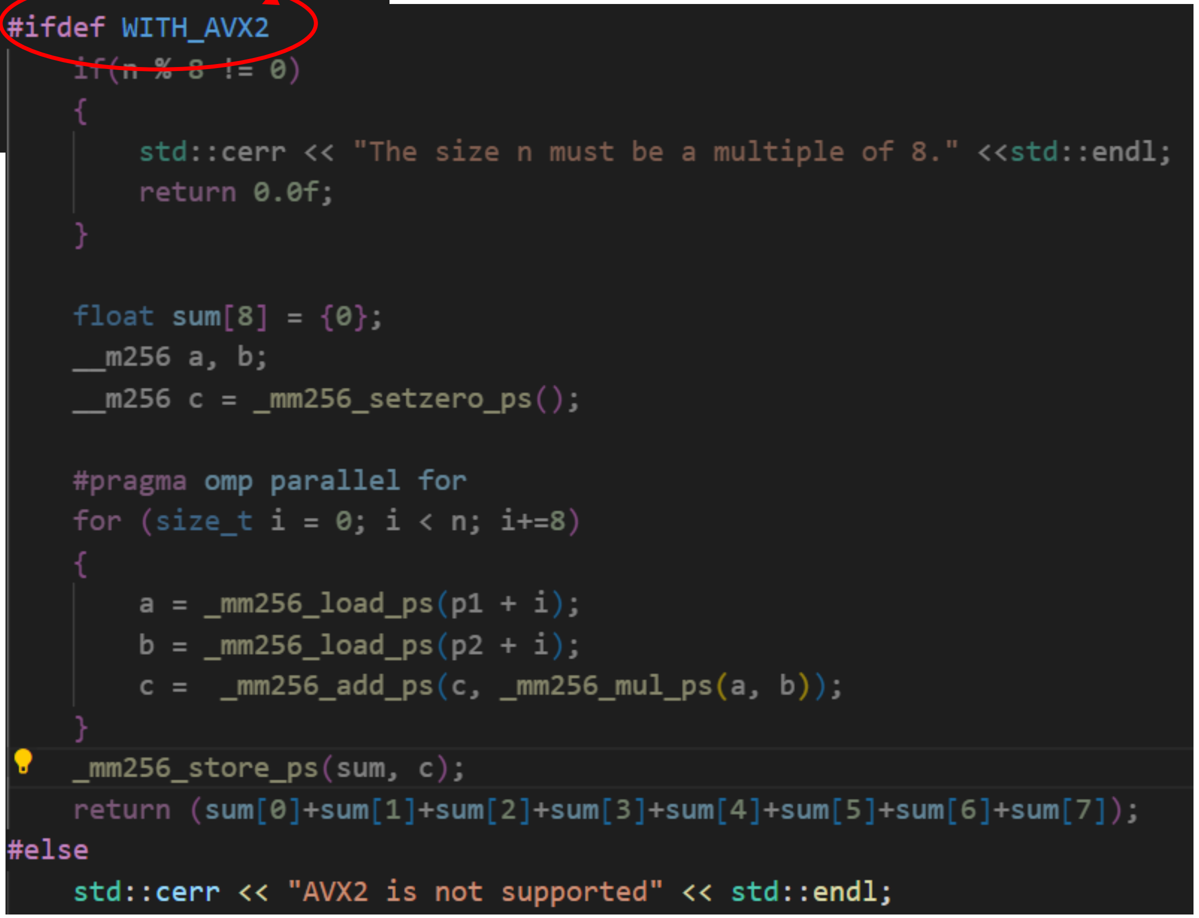
2. If you compile it again, the output still mentions AVX2 is not supported.



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ g++ -o main *.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ ./main
normal: result=9.1, duration = 450ms
unloop: result=9.1, duration = 459ms
AVX2 is not supported
SIMD: result=0, duration = 0ms
AVX2 is not supported
SIMD+OpenMP: result=0, duration = 0ms
```

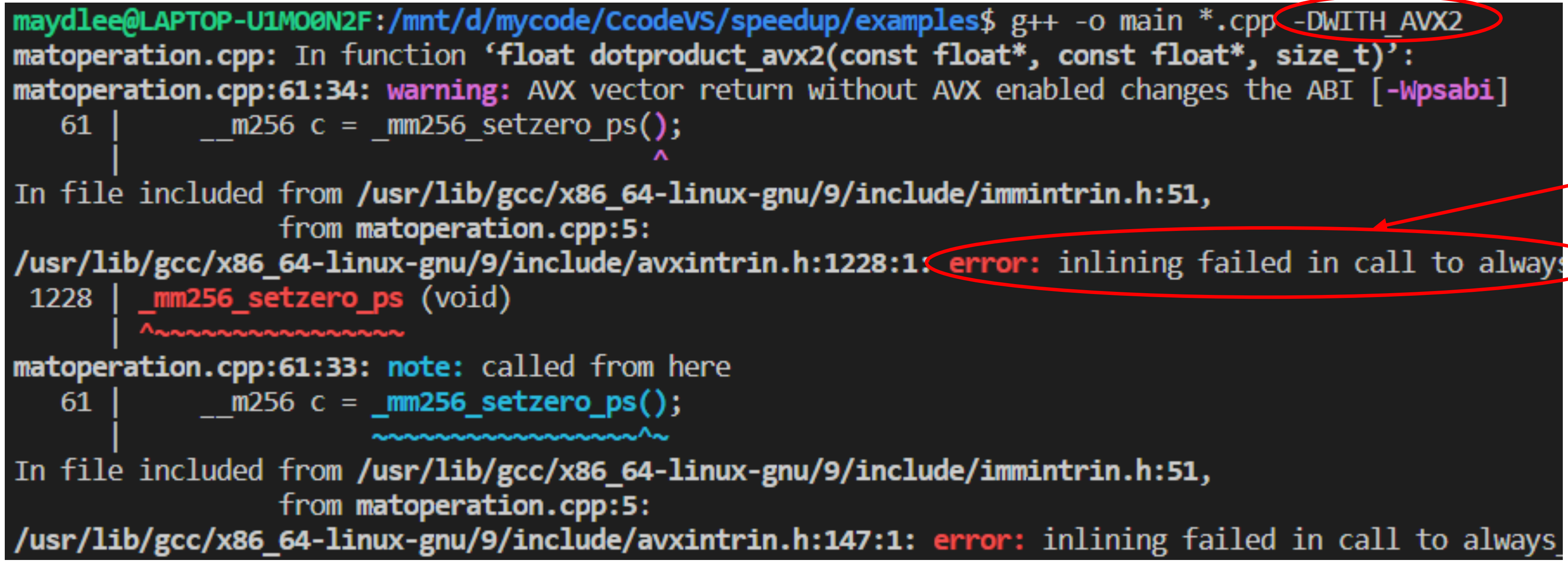
No AVX2 support

The macro WITH_AVX2 should be enabled when you compile.



```
#ifdef WITH_AVX2
    if(n % 8 != 0)
    {
        std::cerr << "The size n must be a multiple of 8." <<std::endl;
        return 0.0f;
    }

    float sum[8] = {0};
    __m256 a, b;
    __m256 c = _mm256_setzero_ps();

    #pragma omp parallel for
    for (size_t i = 0; i < n; i+=8)
    {
        a = _mm256_load_ps(p1 + i);
        b = _mm256_load_ps(p2 + i);
        c =  _mm256_add_ps(c, _mm256_mul_ps(a, b));
    }
    _mm256_store_ps(sum, c);
    return (sum[0]+sum[1]+sum[2]+sum[3]+sum[4]+sum[5]+sum[6]+sum[7]);
#else
    std::cerr << "AVX2 is not supported" << std::endl;
```

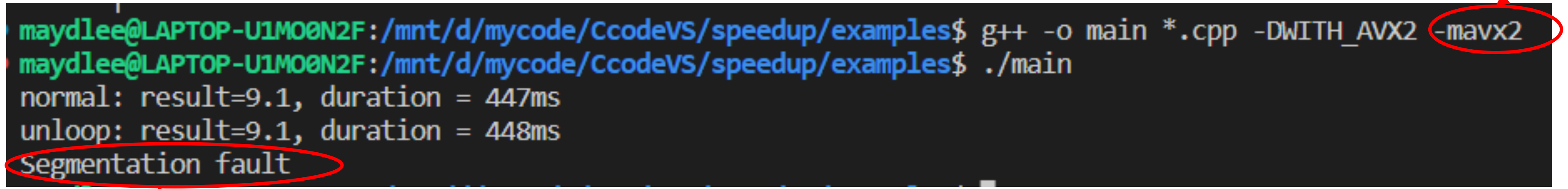
To enable the macro by the option -DWITH_AVX2



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ g++ -o main *.cpp -DWITH_AVX2
matoperation.cpp: In function ‘float dotproduct_avx2(const float*, const float*, size_t)’:
matoperation.cpp:61:34: warning: AVX vector return without AVX enabled changes the ABI [-Wpsabi]
   61 |     __m256 c = _mm256_setzero_ps();
      |                                  ^
In file included from /usr/lib/gcc/x86_64-linux-gnu/9/include/immintrin.h:51,
                 from matoperation.cpp:5:
/usr/lib/gcc/x86_64-linux-gnu/9/include/avxintrin.h:1228:1: error: inlining failed in call to always
 1228 | _mm256_setzero_ps (void)
      | ^~~~~~~~~~~~~~~~~
matoperation.cpp:61:33: note: called from here
   61 |     __m256 c = _mm256_setzero_ps();
      |                ~~~~~~~~~~~~~~~~~^~
In file included from /usr/lib/gcc/x86_64-linux-gnu/9/include/immintrin.h:51,
                 from matoperation.cpp:5:
/usr/lib/gcc/x86_64-linux-gnu/9/include/avxintrin.h:147:1: error: inlining failed in call to always_
```

You may get the error message: error:inlining failed .. because you didn’t tell the compiler to enable AVX2.

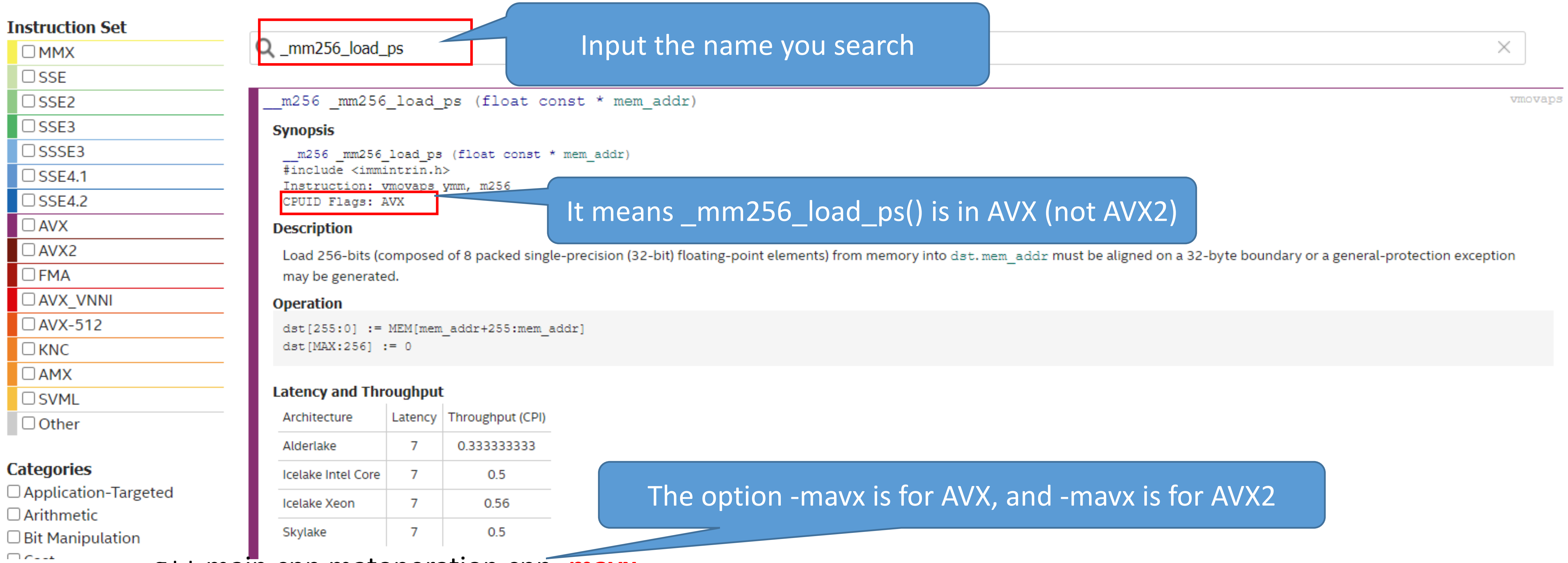
Please use the option -mavx2 to let g++ enable AVX2 support.



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ g++ -o main *.cpp -DWITH_AVX2 -mavx2
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ ./main
normal: result=9.1, duration = 447ms
unloop: result=9.1, duration = 448ms
Segmentation fault
```

运行到调用avx2指令时出现段错误

- https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html



**Instruction Set**

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX_VNNI
- ☐ AVX-512
- ☐ KNC
- ☐ AMX
- ☐ SVML
- ☐ Other

**Categories**

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation

_mm256_load_ps

Input the name you search

```
__m256 _mm256_load_ps (float const * mem_addr)
```

vmovaps

**Synopsis**

```
__m256 _mm256_load_ps (float const * mem_addr)
#include <immintrin.h>
Instruction: vmovaps ymm, m256
CPUID Flags: AVX
```

It means _mm256_load_ps() is in AVX (not AVX2)

**Description**

Load 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements) from memory into dst. mem_addr must be aligned on a 32-byte boundary or a general-protection exception may be generated.

**Operation**

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

**Latency and Throughput**

| Architecture | Latency | Throughput (CPI) |
|---|---|---|
| Alderlake | 7 | 0.333333333 |
| Icelake Intel Core | 7 | 0.5 |
| Icelake Xeon | 7 | 0.56 |
| Skylake | 7 | 0.5 |

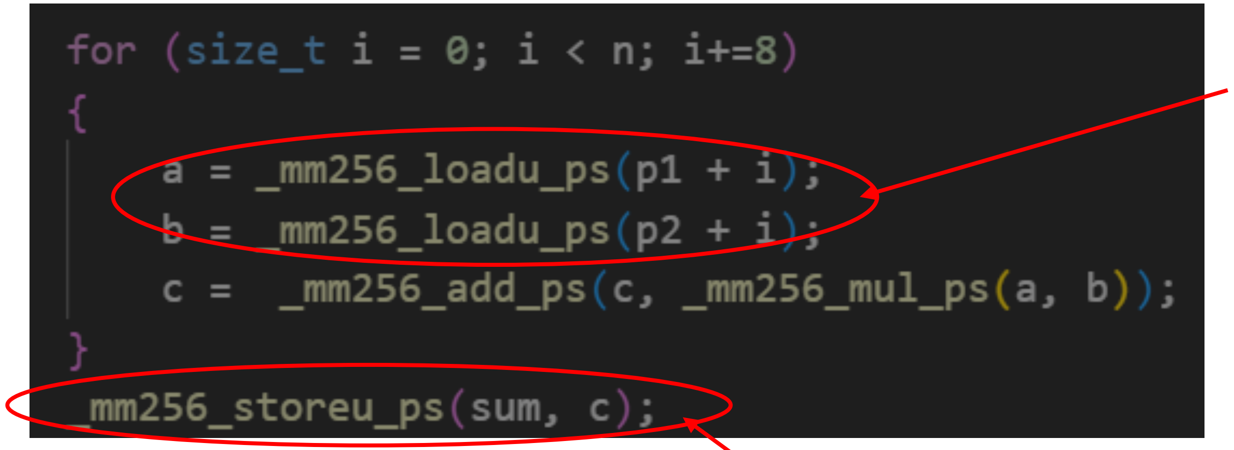
The option -mavx is for AVX, and -mavx is for AVX2

```
g++ main.cpp matoperation.cpp -mavx

g++ main.cpp matoperation.cpp -mavx2
```

You still may get segment fault or wrong results.

①For Intel CPU, it’s better to use loadu & storeu, nor load/store.

②load and store are for aligned memory only.

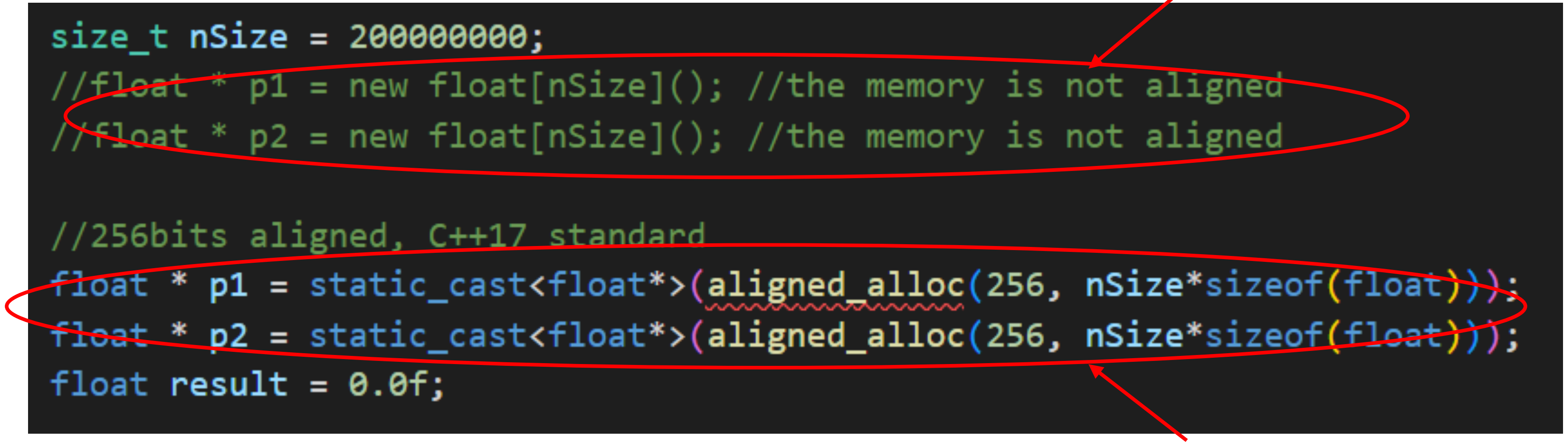


```
for (size_t i = 0; i < n; i+=8)
{
    a = _mm256_loadu_ps(p1 + i);
    b = _mm256_loadu_ps(p2 + i);
    c =  _mm256_add_ps(c, _mm256_mul_ps(a, b));
}
_mm256_storeu_ps(sum, c);
```

_loadu here is for unaligned memeory

_storeu here is for unaligned memory

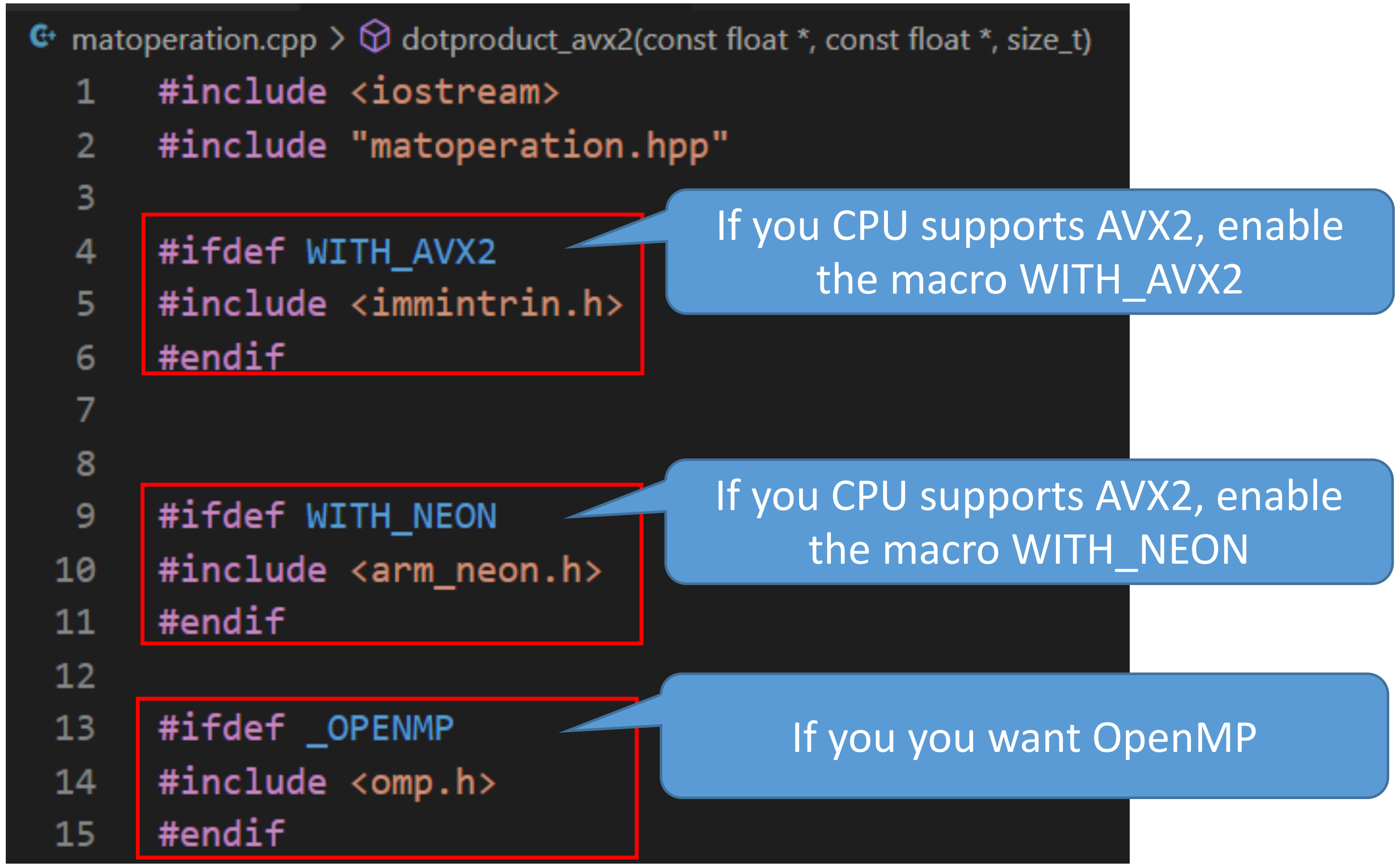
Unaligned memory allocation



```
size_t nSize = 200000000;
//float * p1 = new float[nSize](); //the memory is not aligned
//float * p2 = new float[nSize](); //the memory is not aligned

//256bits aligned, C++17 standard
float * p1 = static_cast<float*>(aligned_alloc(256, nSize*sizeof(float)));
float * p2 = static_cast<float*>(aligned_alloc(256, nSize*sizeof(float)));
float result = 0.0f;
```

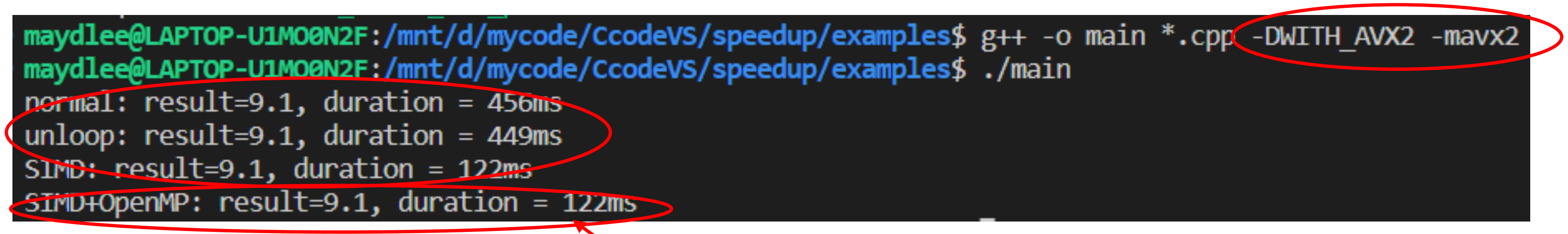
Aligned memory allocation

3. To include different header files by different macros.



```
matoperation.cpp > dotproduct_avx2(const float *, const float *, size_t)
#include <iostream>
#include "matoperation.hpp"

#ifdef WITH_AVX2
#include <immintrin.h>
#endif


#ifdef WITH_NEON
#include <arm_neon.h>
#endif

#ifdef _OPENMP
#include <omp.h>
#endif
```

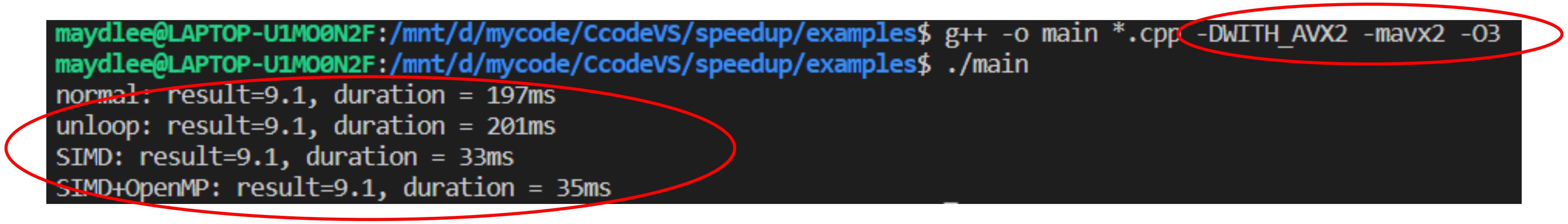
If you CPU supports AVX2, enable the macro WITH_AVX2
If you CPU supports AVX2, enable the macro WITH_NEON
If you you want OpenMP



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ g++ -o main *.cpp -DWITH_AVX2 -mavx2
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ ./main
normal: result=9.1, duration = 456ms
unloop: result=9.1, duration = 449ms
SIMD: result=9.1, duration = 122ms
SIMD+OpenMP: result=9.1, duration = 122ms
```

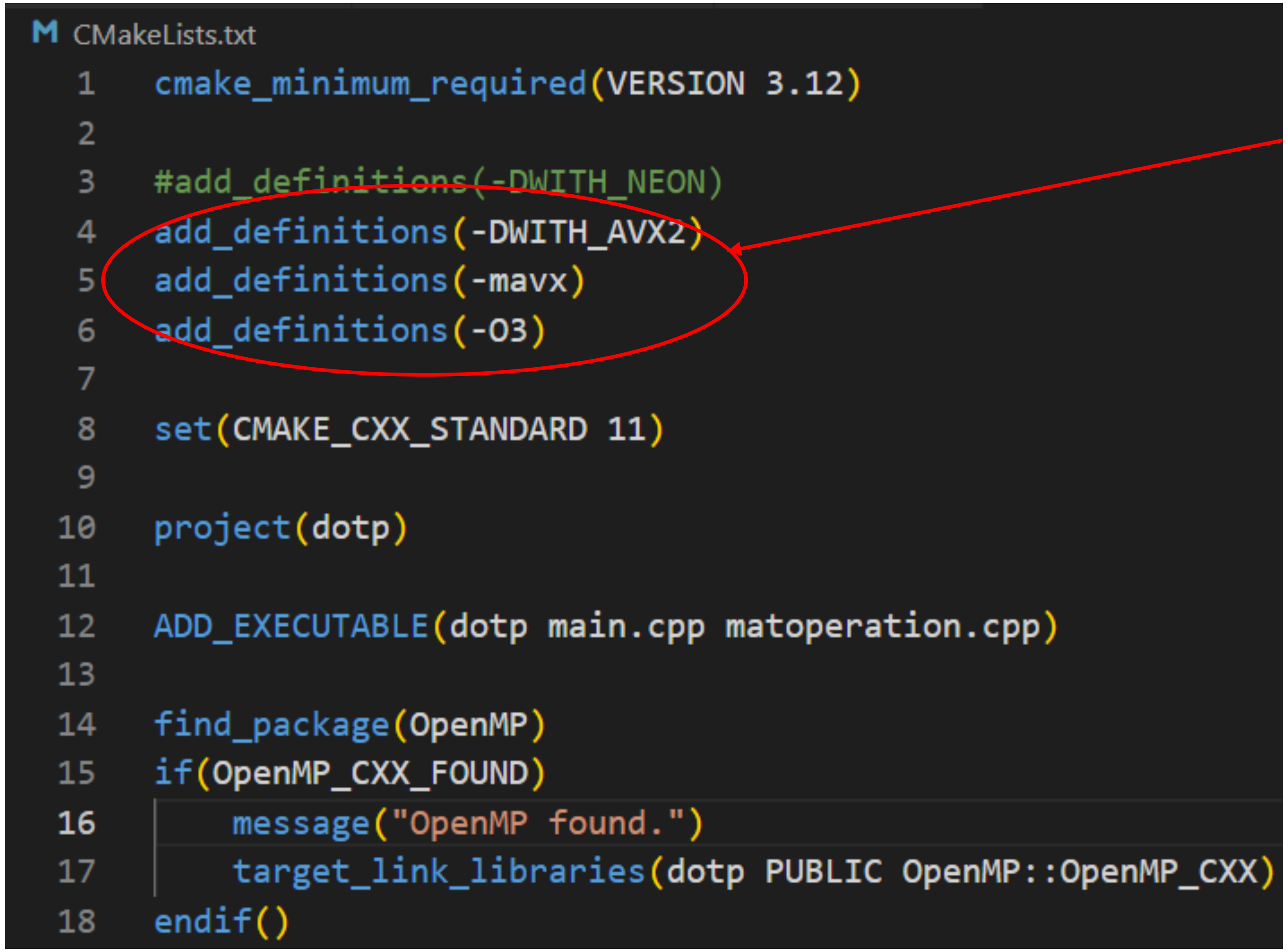
OpenMP没有启动

You can use –O3 to gain the maximum speed.



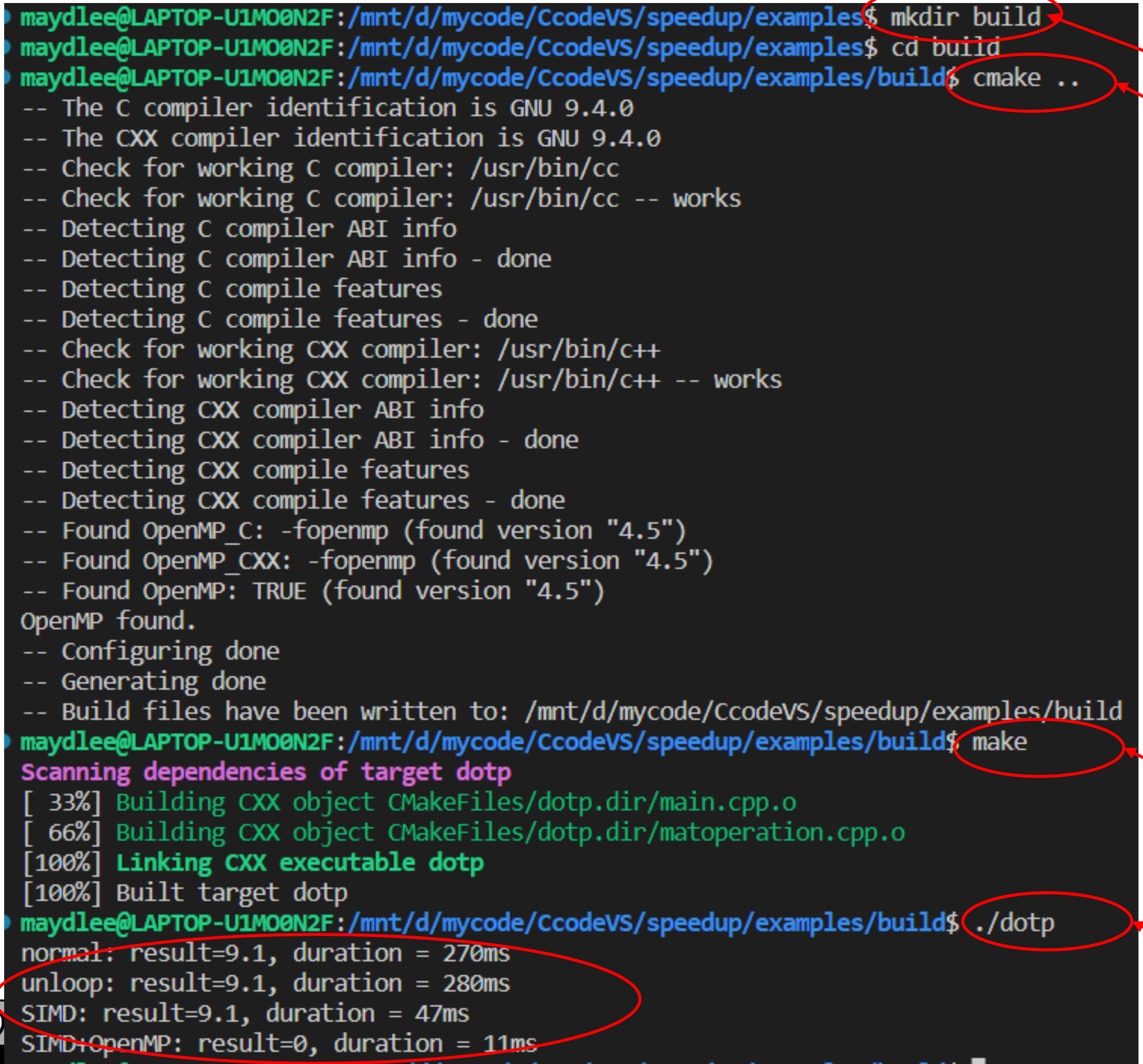
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ g++ -o main *.cpp -DWITH_AVX2 -mavx2 -O3
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ ./main
normal: result=9.1, duration = 197ms
unloop: result=9.1, duration = 201ms
SIMD: result=9.1, duration = 33ms
SIMD+OpenMP: result=9.1, duration = 35ms
```

How to do the previous mentioned in CMakeLists.txt


CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)

#add_definitions(-DWITH_NEON)
add_definitions(-DWITH_AVX2)
add_definitions(-mavx)
add_definitions(-O3)

set(CMAKE_CXX_STANDARD 11)

project(dotp)

ADD_EXECUTABLE(dotp main.cpp matoperation.cpp)

find_package(OpenMP)
if(OpenMP_CXX_FOUND)
    message("OpenMP found.")
    target_link_libraries(dotp PUBLIC OpenMP::OpenMP_CXX)
endif()
```

Add some options for the compiler



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ mkdir build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples$ cd build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples/build$ cmake ..
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found OpenMP_C: -fopenmp (found version "4.5")
-- Found OpenMP_CXX: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
OpenMP found.
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/mycode/CcodeVS/speedup/examples/build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples/build$ make
Scanning dependencies of target dotp
[ 33%] Building CXX object CMakeFiles/dotp.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/dotp.dir/matoperation.cpp.o
[100%] Linking CXX executable dotp
[100%] Built target dotp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/speedup/examples/build$ ./dotp
normal: result=9.1, duration = 270ms
unloop: result=9.1, duration = 280ms
SIMD: result=9.1, duration = 47ms
SIMD+OpenMP: result=0, duration = 11ms
```

You can create a directory for generated files by cmake

To use the file CMakeLists.txt in the parent directory

Make it!

Run it!

# Exercise:

Write a program to add 2 `float` vectors whose size should be more than 1M. You can initialize the two vectors with values like `0.f, 1.f, 2.f,` …

- Use pure C source code and SIMD (AVX2 or NEON) separately, and compare their speeds
- Use OpenMP to speed up the addition. Can you get the correct result?