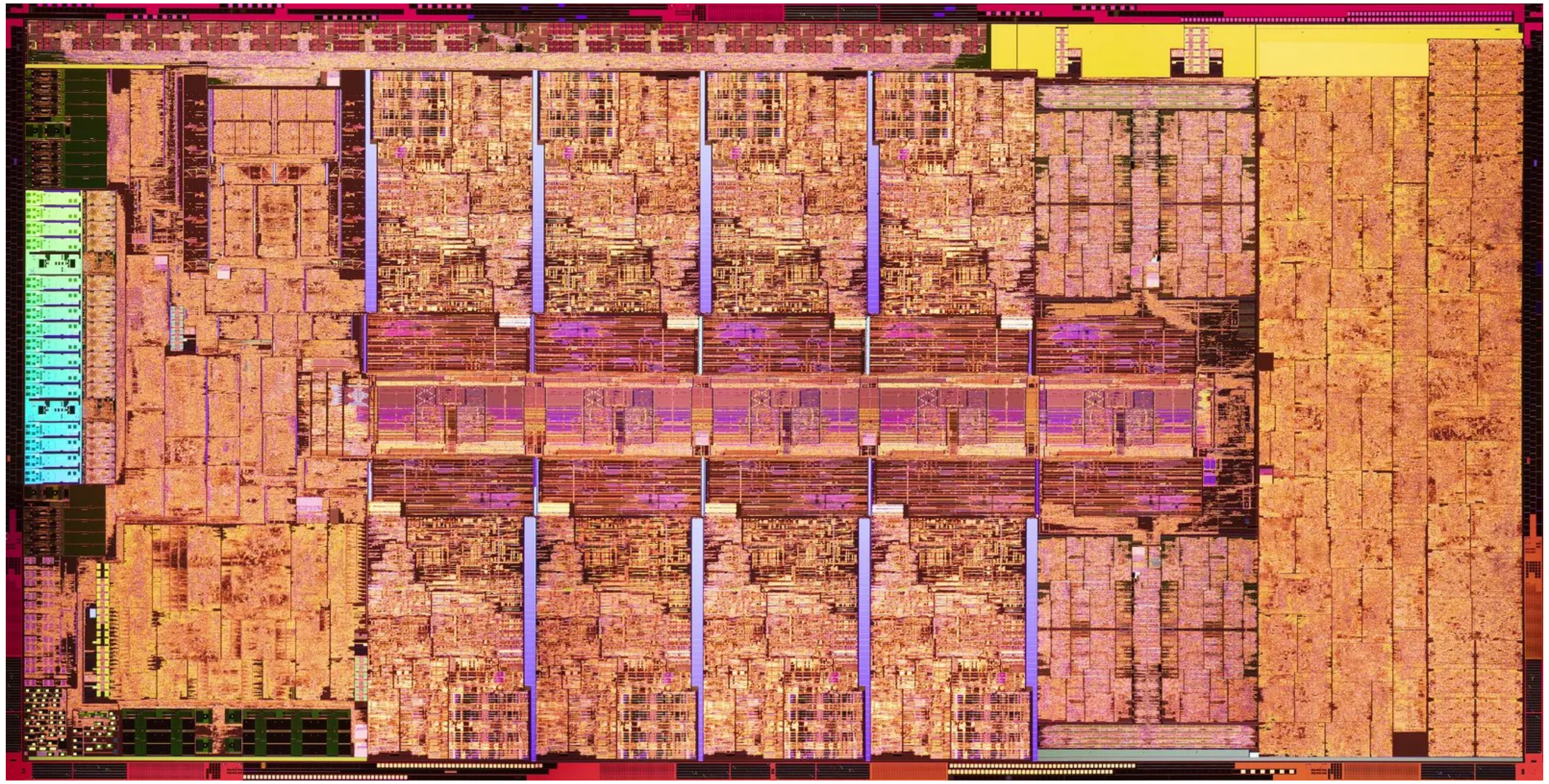
Intro to SIMD Programming

Parallelism Hierarchy

High Level: Distributed Computing

- separate computers working in parallel
- distributed memory
- MPI, Legion, MapReduce

Parallelism Hierarchy



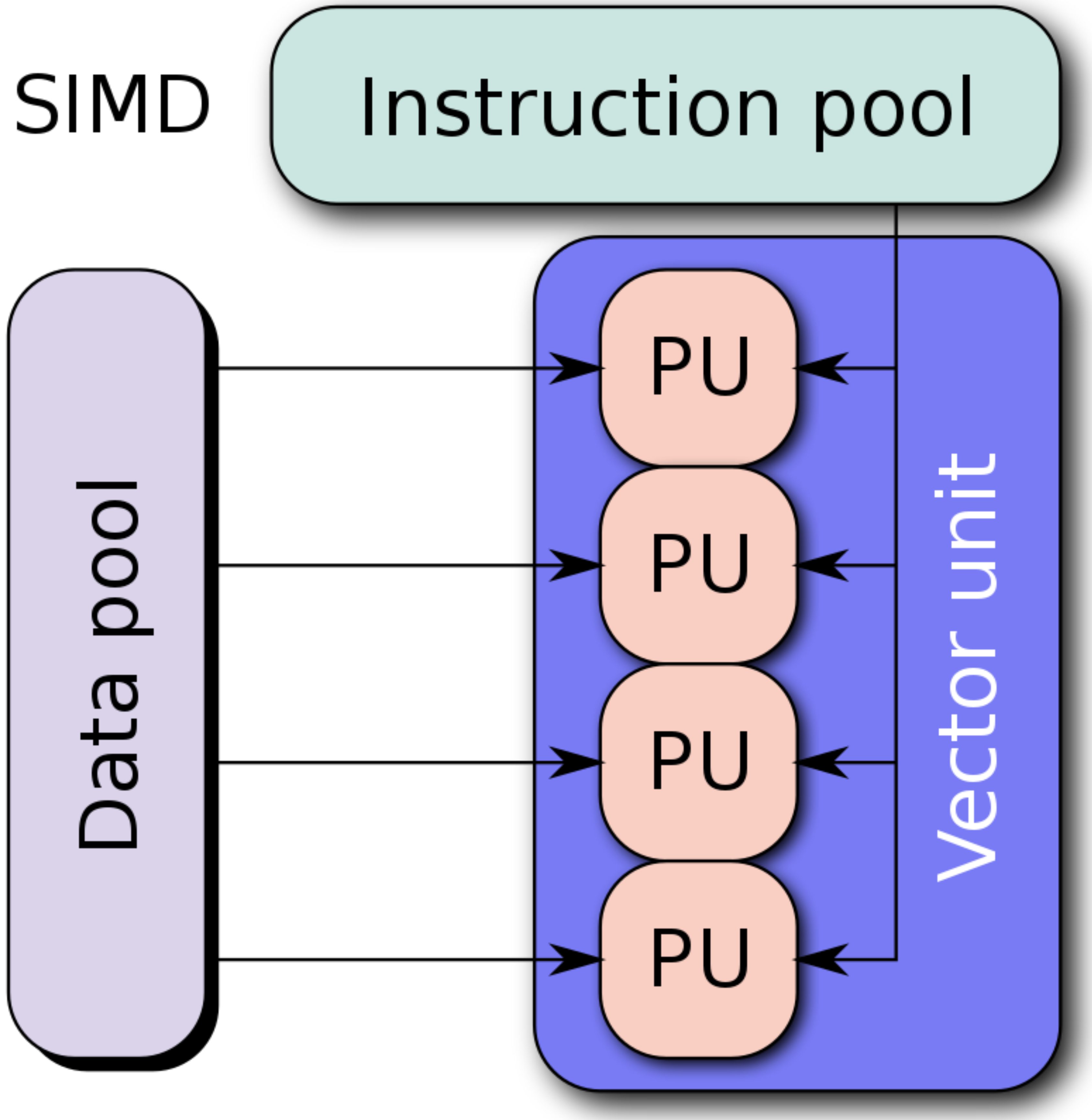
Mid Level: Thread-level Parallelism

- separate threads working in parallel
- shared memory
- pthreads, OpenMP, TBB

Parallelism Hierarchy

Low Level: SIMD

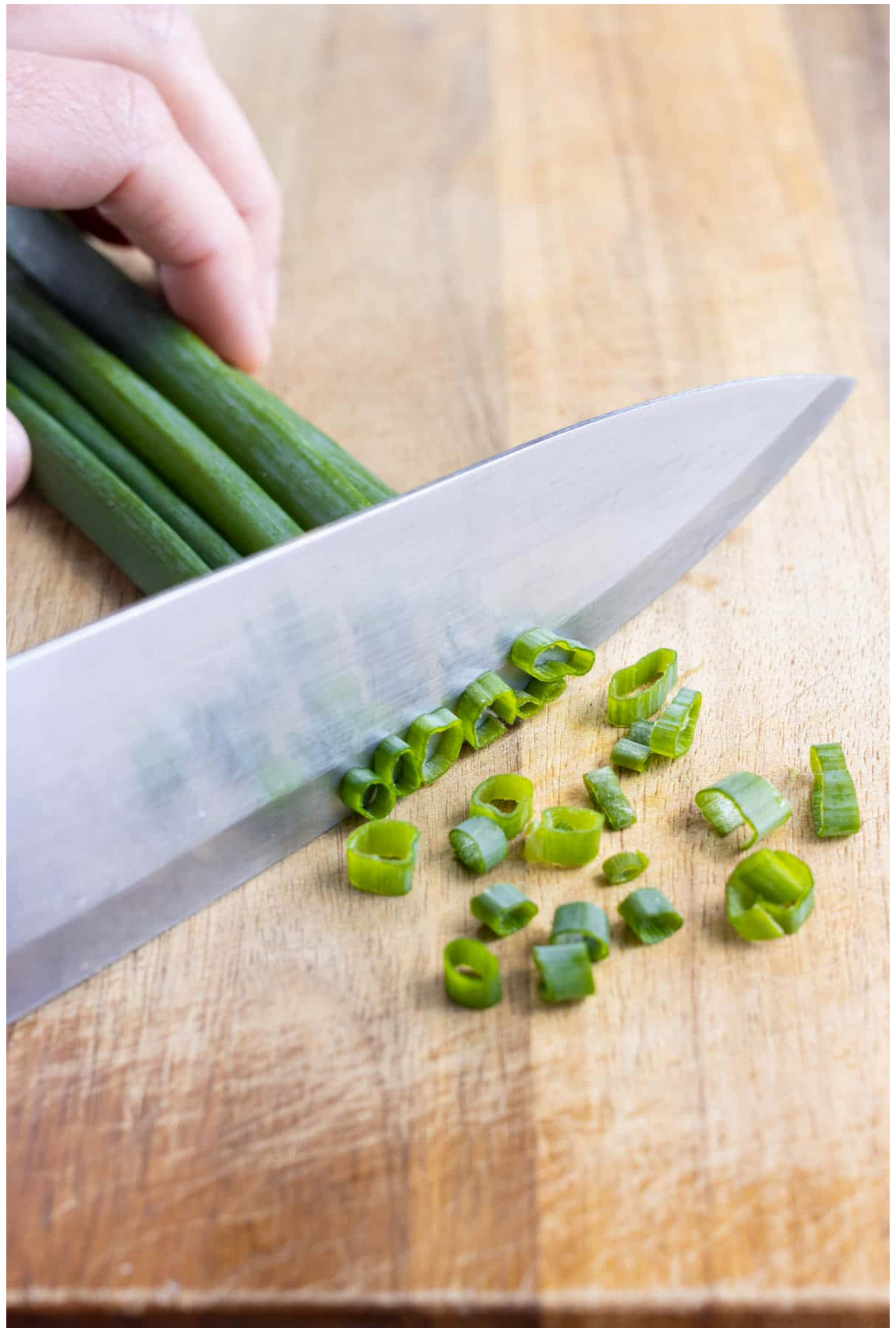
- **S**ingle **I**nstruction **M**ultiple **D**ata
- {128, 256, 512} bit registers
- SSE, AVX, NEON instructions



This was the first image result for SIMD?

Parallelism Hierarchy

Low Level: SIMD



- **S**ingle **I**nstruction **M**ultiple **D**ata
- {128, 256, 512} bit registers
- SSE, AVX, NEON instructions

Why should I care?

1.
2.
3.
4.

Why should I care?

1. I want to make my code faster
2.
3.
4.

Why should I care?

1. I want to make my code faster
2. I want to make my code faster
3.
4.

Why should I care?

1. I want to make my code faster
2. I want to make my code faster
3. I want to make my code faster
4.

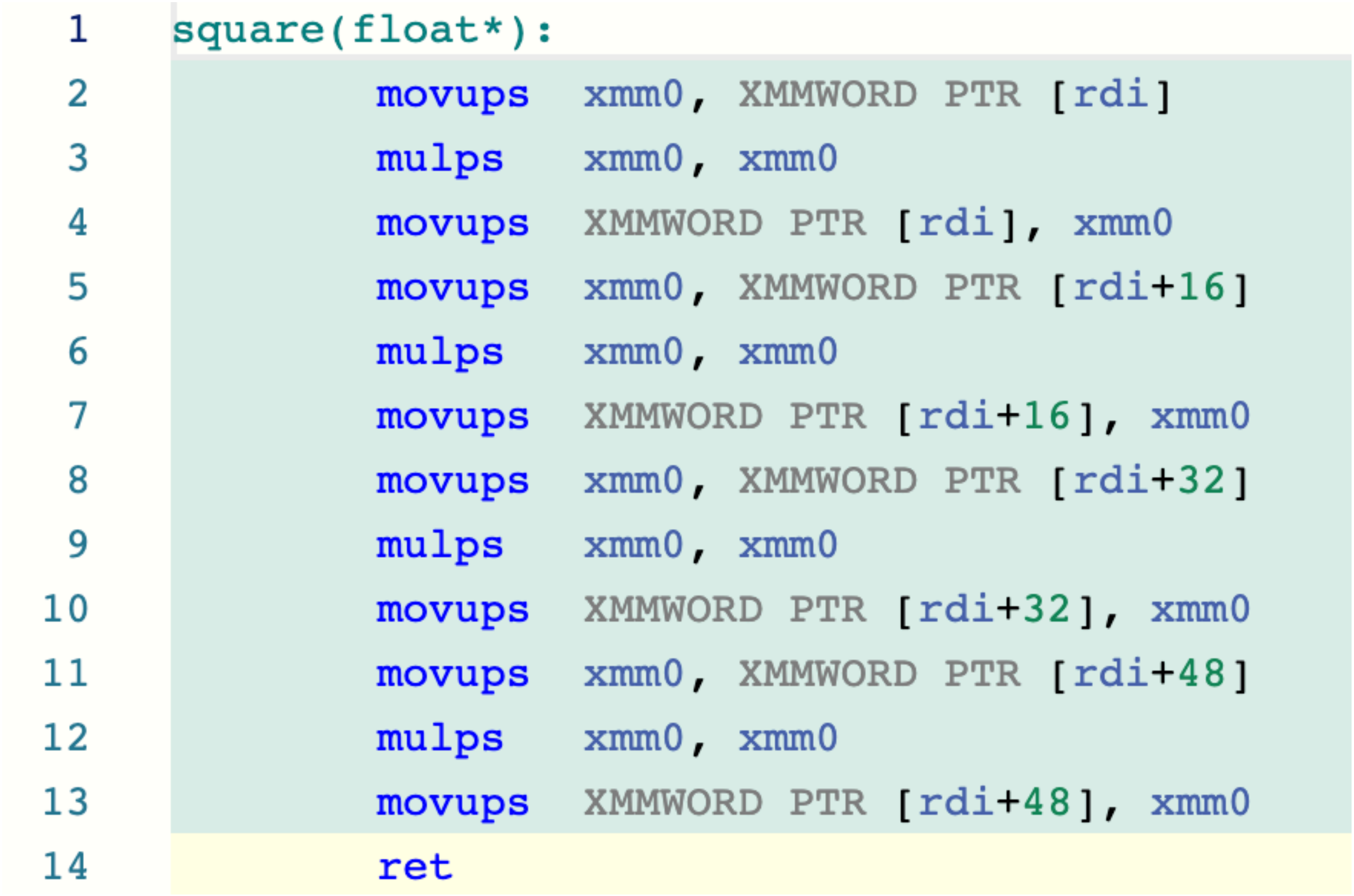
Why should I care?

1. I want to make my code faster
2. I want to make my code faster
3. I want to make my code faster
4. I want to make my code faster

SIMD Strategy: Auto vectorization

```c
void square(float values[16]) {
  for (int i = 0; i < 16; i++) {
    values[i] = values[i] * values[i];
  }
}
```

compiled with -O3



```
square(float*):
        movups  xmm0, XMMWORD PTR [rdi]
        mulps   xmm0, xmm0
        movups  XMMWORD PTR [rdi], xmm0
        movups  xmm0, XMMWORD PTR [rdi+16]
        mulps   xmm0, xmm0
        movups  XMMWORD PTR [rdi+16], xmm0
        movups  xmm0, XMMWORD PTR [rdi+32]
        mulps   xmm0, xmm0
        movups  XMMWORD PTR [rdi+32], xmm0
        movups  xmm0, XMMWORD PTR [rdi+48]
        mulps   xmm0, xmm0
        movups  XMMWORD PTR [rdi+48], xmm0
        ret
```

xmm: 128-bit

ymm: 256-bit

zmm: 512-bit

SIMD Strategy: Auto vectorization

wait, if the compiler already vectorizes my code automatically,

why would I ever vectorize my code manually?

In practice, (at the time of writing this):

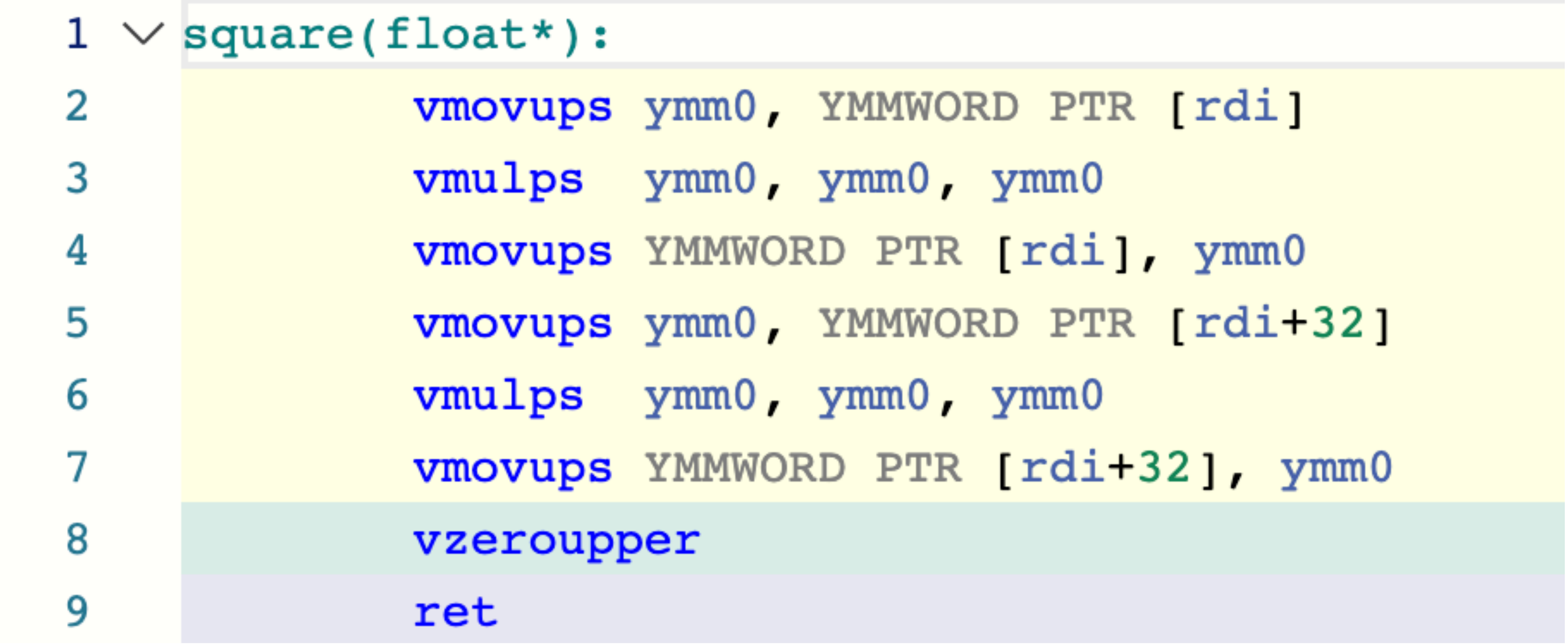
- automatic vectorization doesn't work reliably
  - no warnings / explanations if things don't vectorize
- automatic vectorization doesn't generate optimal code
  - e.g. emits 128-bit or 256-bit instructions* for AVX512 machine

SIMD Strategy: Auto vectorization

gcc13, compiled with -O3

-ffast-math -march=skylake-avx512 -ftree-vectorize

```c
void square(float values[16]) {
  for (int i = 0; i < 16; i++) {
    values[i] = values[i] * values[i];
  }
}
```



```asm
square(float*):
        vmovups ymm0, YMMWORD PTR [rdi]
        vmulps  ymm0, ymm0, ymm0
        vmovups YMMWORD PTR [rdi], ymm0
        vmovups ymm0, YMMWORD PTR [rdi+32]
        vmulps  ymm0, ymm0, ymm0
        vmovups YMMWORD PTR [rdi+32], ymm0
        vzeroupper
        ret
```

xmm: 128-bit

ymm: 256-bit

zmm: 512-bit

SIMD Strategy: Auto vectorization

Upsides:

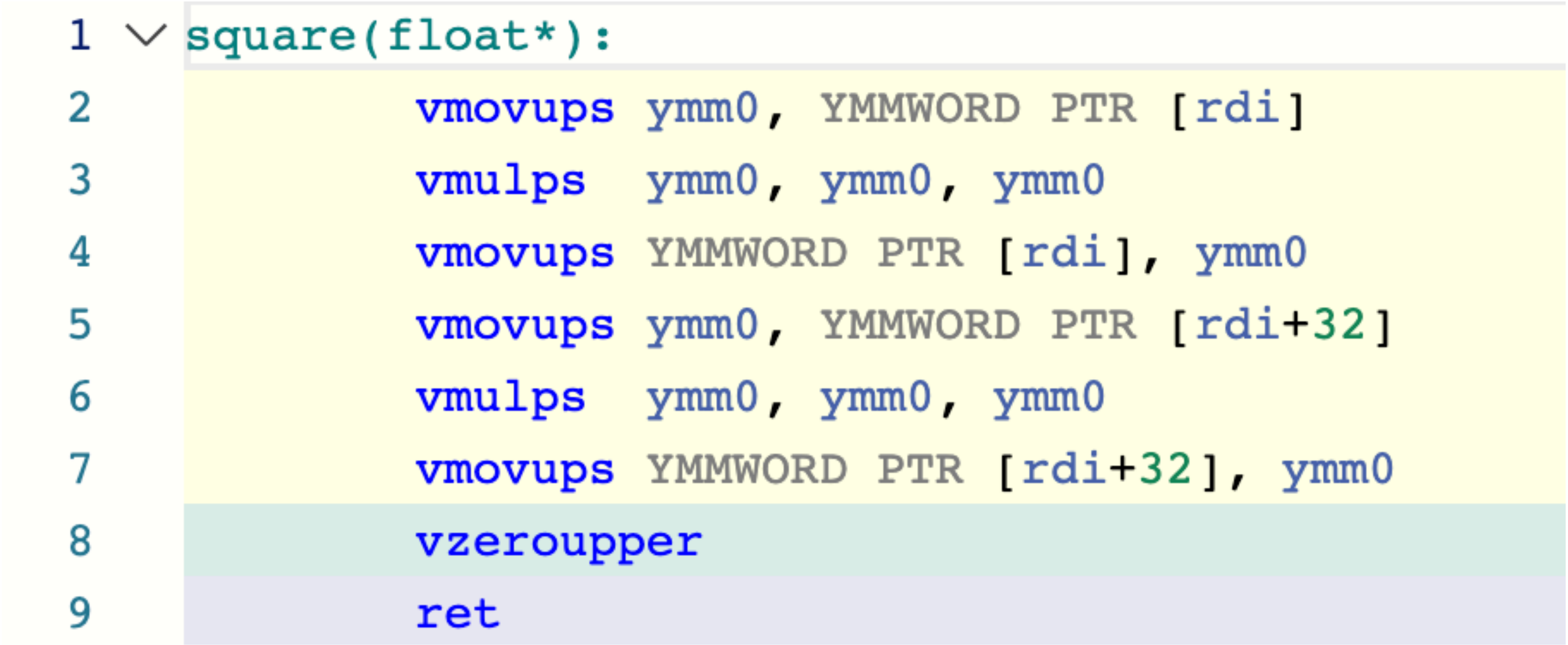
- Almost zero effort
- Okay performance

Downsides:

- Unreliable
- Okay performance
- Unclear why it doesn't work

SIMD Strategy: OpenMP #pragma

```c
void square(float values[16]) {
  #pragma omp simd
  for (int i = 0; i < 16; i++) {
    values[i] = values[i] * values[i];
  }
}
```



```
square(float*):
        vmovups ymm0, YMMWORD PTR [rdi]
        vmulps  ymm0, ymm0, ymm0
        vmovups YMMWORD PTR [rdi], ymm0
        vmovups ymm0, YMMWORD PTR [rdi+32]
        vmulps  ymm0, ymm0, ymm0
        vmovups YMMWORD PTR [rdi+32], ymm0
        vzeroupper
        ret
```

same assembly as automatic vectorization

SIMD Strategy: OpenMP #pragma

In my tests, I never saw any performance improvement from

`#pragma omp simd`

```
void square(float values[16]) {
  #pragma omp simd
  for (int i = 0; i < 16; i++) {
    values[i] = values[i] * values[i];
  }
}
```

```
square(float*):
        vmovups ymm0, YMMWORD PTR [rdi]
        vmulps  ymm0, ymm0, ymm0
        vmovups YMMWORD PTR [rdi], ymm0
        vmovups ymm0, YMMWORD PTR [rdi+32]
        vmulps  ymm0, ymm0, ymm0
        vmovups YMMWORD PTR [rdi+32], ymm0
        vzeroupper
        ret
```

same assembly as automatic vectorization

SIMD Strategy: Intrinsics

Conceptually (for AVX):

- #include <immintrin.h>
- Replace calculations by their associated intrinsics:
    - __m128 _mm_add_ps(__m128 a, __m128 b)
    - __m128 _mm_mul_ps(__m128 a, __m128 b)

SIMD Strategy: Intrinsics

Conceptually (for AVX):

- #include <immintrin.h>
- Replace calculations by their associated intrinsics:

which kind of operation

- `__m128 _mm_add_ps(__m128 a, __m128 b)`
- `__m128 _mm_mul_ps(__m128 a, __m128 b)`

single precision

128-bit vector of floats

SIMD Strategy: Intrinsics

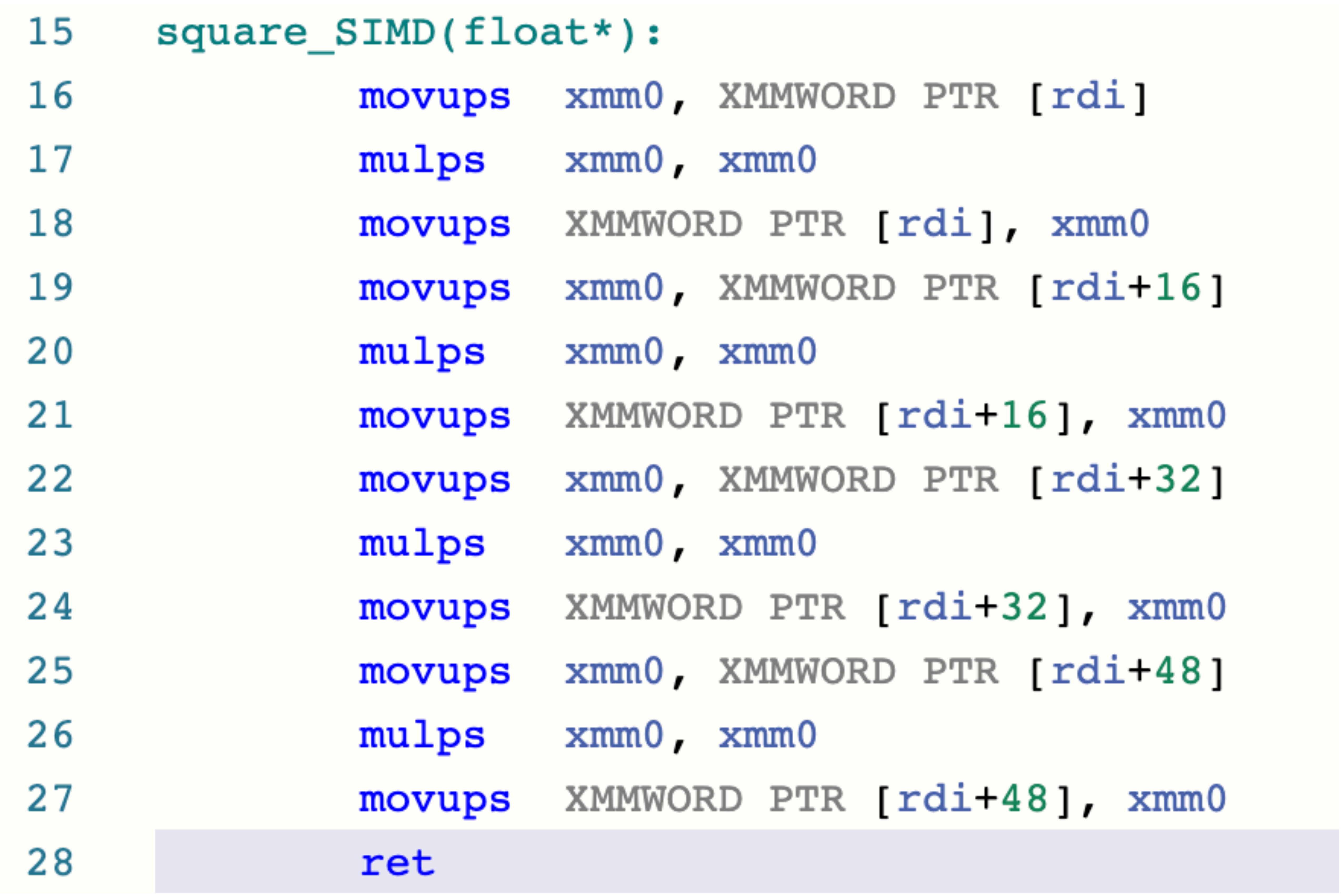
```
void square(float values[16]) {
  for (int i = 0; i < 16; i++) {
    values[i] = values[i] * values[i];
  }
}
```

```
void square_SIMD(float values[16]) {
  for (int i = 0; i < 4; i++) {
    __m128 f = _mm_loadu_ps(values + 4 * i);
    f = _mm_mul_ps(f, f);
    _mm_storeu_ps(values + 4 * i, f);
  }
}
```

https://godbolt.org/z/58nxdodT1

SIMD Strategy: Intrinsics

```c
void square_SIMD(float values[16]) {
  for (int i = 0; i < 4; i++) {
    __m128 f = _mm_loadu_ps(values + 4 * i);
    f = _mm_mul_ps(f, f);
    _mm_storeu_ps(values + 4 * i, f);
  }
}
```



```asm
square_SIMD(float*):
        movups  xmm0, XMMWORD PTR [rdi]
        mulps   xmm0, xmm0
        movups  XMMWORD PTR [rdi], xmm0
        movups  xmm0, XMMWORD PTR [rdi+16]
        mulps   xmm0, xmm0
        movups  XMMWORD PTR [rdi+16], xmm0
        movups  xmm0, XMMWORD PTR [rdi+32]
        mulps   xmm0, xmm0
        movups  XMMWORD PTR [rdi+32], xmm0
        movups  xmm0, XMMWORD PTR [rdi+48]
        mulps   xmm0, xmm0
        movups  XMMWORD PTR [rdi+48], xmm0
        ret
```

xmm: 128-bit

ymm: 256-bit

zmm: 512-bit

SIMD Strategy: Intrinsics

Upsides:

- Control / Performance
- No additional dependencies

Downsides:

- Not portable (AVX, SSE, NEON, ...)
- Unreadable
- Invasive refactoring
- Really low-level
- Conditional Expressions

SIMD Strategy: Intrinsics

Upsides:

- Control / Performance
- No additional dependencies

Downsides:

- **Not portable (AVX, SSE, NEON, ...)**
- **Unreadable**
- **Invasive** refactoring
- **Really low-level**
- Conditional Expressions

These are addressable by
an appropriate abstraction

SIMD Strategy: Use a library

Many options:

- p12tic/libsimdpp
- ermig1979/Simd
- google/highway
- mitsuba-renderer/enoki
- std::experimental::simd (parallelism TS v2)

Most are portable, expose operator overloads, ...

Look for one with the features your project needs

SIMD Strategy: Use a library

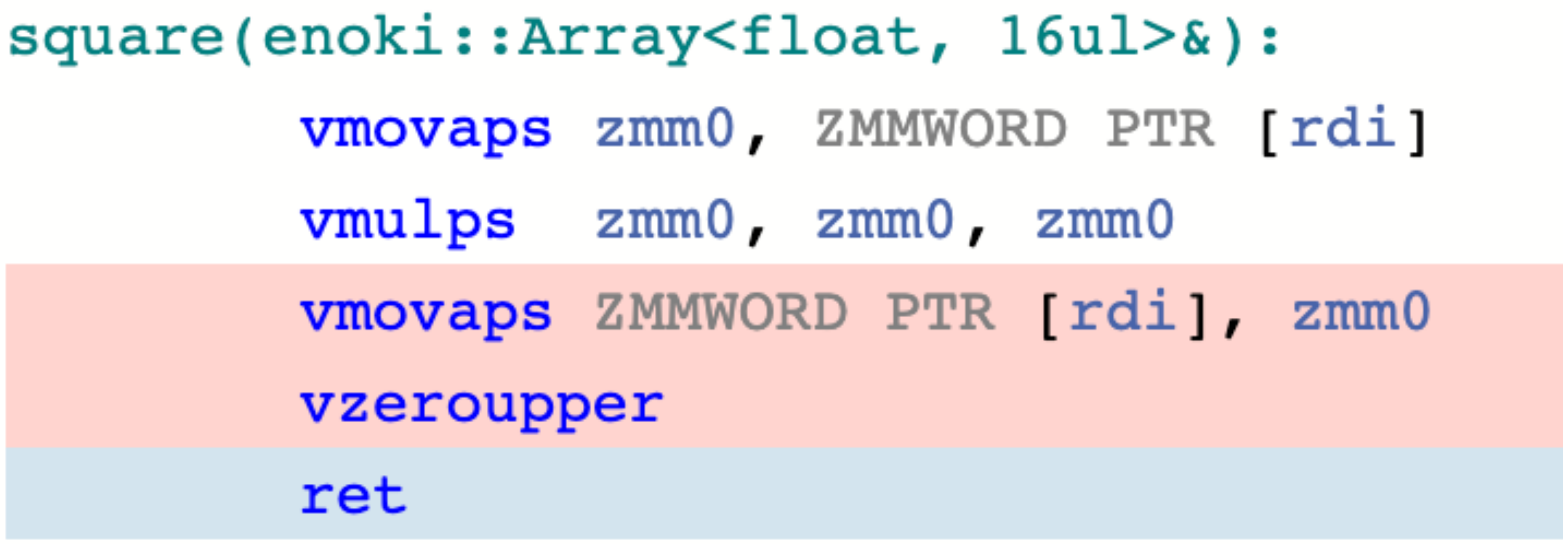
https://github.com/mitsuba-renderer/enoki

was the best-looking option I tried:

- easy to use
- performant implementations of math functions
- wonderful documentation

```cpp
#include "enoki/array.h"

void square(enoki::Array<float, 16> & values) {
  values = values * values;
}
```



```
square(enoki::Array<float, 16ul>&):
        vmovaps zmm0, ZMMWORD PTR [rdi]
        vmulps  zmm0, zmm0, zmm0
        vmovaps ZMMWORD PTR [rdi], zmm0
        vzeroupper
        ret
```

SIMD Strategy: Use a library

A more realistic example of a SIMD refactor with enoki:

https://github.com/samuelpmish/material_benchmarks/blob/main/src/J2_plasticity.cpp

SIMD Strategy: Use a library

A more realistic example of a SIMD refactor with enoki:

https://github.com/samuelpmish/material_benchmarks/blob/main/src/J2_plasticity.cpp

```
cd /path/to/material_benchmarks/assembly
$ ./x86_simd_report.sh J2_plasticity_scalar_x86.s
128-bit instructions: 451
256-bit instructions: 29
512-bit instructions: 0
$ ./x86_simd_report.sh J2_plasticity_simd_x86.s
128-bit instructions: 1
256-bit instructions: 0
512-bit instructions: 467
```

Conditional Expressions

Adapting straight-line code to use a SIMD library is doable,
but what about conditional branching?

```
float z = sin(x) * y;
if (z > 2.0) {
   z -= 1.0;
}
```

Conditional Expressions

Adapting straight-line code to use a SIMD library is doable,
but what about conditional branching?

```
float z = sin(x) * y;
if (z > 2.0) {
   z -= 1.0;
}
```

How can we stop all the lanes from
evaluating the conditional statements?

Conditional Expressions

Adapting straight-line code to use a SIMD library is doable,
but what about conditional branching?

```
enoki::Array<float,8> z = sin(x) * y;
z[z > 2.0] -= 1.0;
```

masked operations

Conditional Expressions

Adapting straight-line code to use a SIMD library is doable,
but what about conditional branching?

```
enoki::Array<float,8> z = sin(x) * y;
z[z > 2.0] -= 1.0;
```

masked operations are *okay*, but still require code modification

Conditional Expressions

Is there a cleaner way to handle conditionals in a SIMD context?

Unfortunately, I believe the answer is "no",
due to fundamental limitations of C++

Conditional Expressions

However, there is a "new" LLVM-based tool that extends
the C++ language to allow for a simpler way to write SIMD code

- Flexible SIMD width
- Supports different underlying hardware
- Conditional masking is handled automatically
- Many calculations are compatible with existing C++

Conditional Expressions

However, there is a "new" LLVM-based tool that extends
the C++ language to allow for a simpler way to write SIMD code

- Flexible SIMD width (warp size)
- Supports different underlying hardware (PTX)
- Conditional masking is handled automatically (vectorization at runtime)
- Many calculations are compatible with existing C++ (SPMD)

GLSL, released in 2001

CUDA C++, released in 2007

Some Performance Numbers

Speedup, relative to original implementation

Calculation	Skylake-X (512-bit)	Raptor-Lake (256-bit)	M1 (128-bit)
axpy	no data	1.1x	1.01x
Neohookean	2.4x	1.6x	1.0x
J2 Plasticity	2.5x	1.9x	1.9x

Some Performance Numbers

Speedup, relative to original implementation

Calculation	Skylake-X (512-bit)	Raptor-Lake (256-bit)	M1 (128-bit)
axpy	no data	1.1x ?!	1.01x ?!
Neohookean	2.4x	1.6x	1.0x ?!
J2 Plasticity	2.5x	1.9x	1.9x

Some Performance Numbers

Arithmetic intensity: (# floating point ops) / (# of bytes moved)

compute-bound vs. memory-bound

Some Performance Numbers

Arithmetic intensity: (# floating point ops) / (# of bytes moved)

compute-bound vs. memory-bound

SIMD isn't helpful for memory-bound kernels!

Is it worth the effort?

In most cases, I'd say: no

unless

1. you know your code has a compute-heavy bottleneck
2. that bottleneck is a calculation running on the CPU
3. application performance is absolutely critical
4. you're prepared to pay the costs to write / maintain the SIMD parts

Is it worth the effort?

In most cases, I'd say: no

unless

1. you know your code has a compute-heavy bottleneck
2. that bottleneck is a calculation running on the CPU
3. application performance is absolutely critical
4. you're prepared to pay the costs to write / maintain the SIMD parts

A good SIMD library helps mitigate the costs in (4),
but there is still a significant amount of work to refactor/maintain.

Summary

- Working with intrinsics directly is awful
- Auto vectorization is helpful, but insufficient
- There are a lot of good SIMD libraries
  - BUT, they still require some refactoring (esp. conditionals)
  - When does std::simd arrive, if ever?
- SIMD is only useful for compute-bound kernels!
- The return on investment with SIMD isn't very high (at most 2-4x)
  - more bang/buck with other optimizations (algorithm, threading, etc)
  - Only pursue SIMD optimizations last

Thanks!