Speed for Free

Current state of auto-vectorization in GCC and Clang

Stefan Fuhrmann

The Hidden Dimension

- Software gets faster by hardware getting faster
  - ✔ Core counts go up
  - ✔ Clocks go up
  - ✔ IPC goes up

- Throughput = **Cores x Clocks x IPC**

Introduction

The Hidden Dimension

- Software gets faster by hardware getting faster
  - ✔ Core counts go up
  - ✔ Clocks go up
  - ✔ IPC goes up
- In 1997 (!) MMX started a silent revolution
  - ✘ 1 op/instruction ⇒ 16 ops/instruction (today)
  - ✘ Not covered natively by C/C++ type system
  - ✘ 90% of performance potentially left on the table by C++
- Throughput = **Cores x Clocks x IPC**

Introduction

The Hidden Dimension

- Software gets faster by hardware getting faster
  - ✔ Core counts go up
  - ✔ Clocks go up
  - ✔ IPC goes up
- In 1997 (!) MMX started a silent revolution
  - ✘ 1 op/instruction ⇒ 16 ops/instruction (today)
  - ✘ Not covered natively by C/C++ type system
  - ✘ 90% of performance potentially left on the table by C++
- Throughput = **Cores x Clocks x IPC x Ops/instruction**

Introduction

My Angle

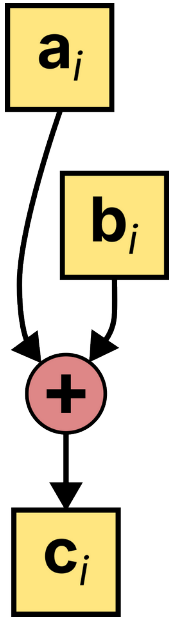
- Motivation
  - I deeply care about performance; it's also my job
  - Hand-vectorization too expensive for many applications
  - Masking in AVX512 and SVE2 simplifies auto-vectorization
  - GCC 15 and Clang 20 enabled auto-vectorization in –O2

- Questions
  - Will the compiler do all the work for me?
  - Gains in simple, STL-like loops
  - How much potential is there at all?

- GCC and Clang are awesome!

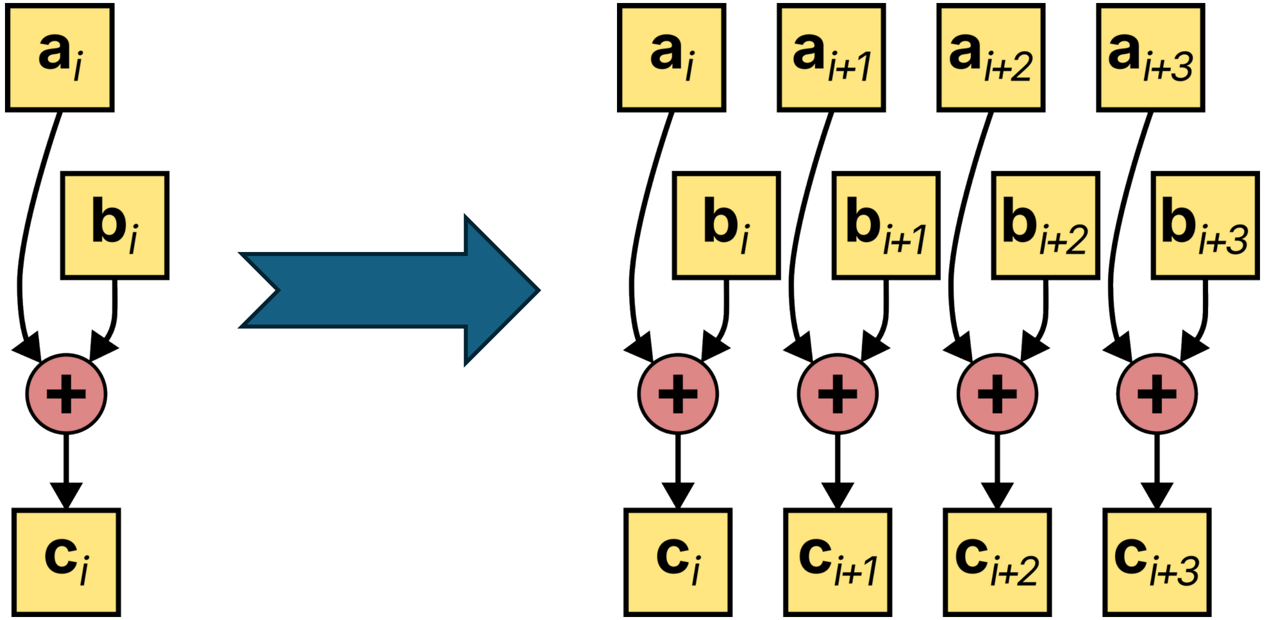
Recap: Vectorization = SIMD

Basic idea akin to loop unrolling:



Recap: Vectorization = SIMD

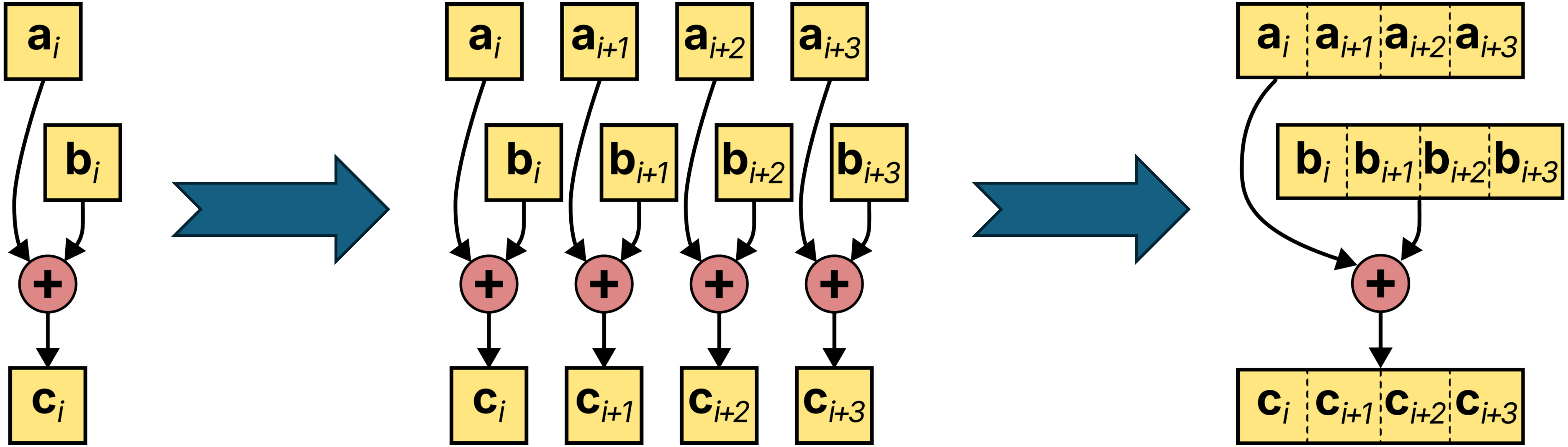
Basic idea akin to loop unrolling:



- Fewer iterations (loop overhead)
- More code

Recap: Vectorization = SIMD

Basic idea akin to loop unrolling:



- Fewer iterations (loop overhead)
- More code

- More work done by each instruction
- Code size similar to scalar

The Repo

https://github.com/stefeff/auto-vectorization

Linear Algebra

- Well-known, regular structures
  - High gains, moderate effort
  - FP operations always use vector units
  - HPC & benchmark relevant

Easy Mode

Linear Algebra

- Well-known, regular structures
  - High gains, moderate effort
  - FP operations always use vector units
  - HPC & benchmark relevant

- 16x16 matrix multiplication
  - Use specialized libs for larger problem sizes!

```
struct alignas(64) Matrix
{
  float data[16][16];
};

Matrix mult(const Matrix& lhs, const Matrix& rhs)
{
  Matrix result;
  for (int i = 0; i < 16; ++i) {
    for (int k = 0; k < 16; ++k) {
      float sum = 0;
      for (int n = 0; n < 16; ++n) {
        sum += lhs.data[i][n] * rhs.data[n][k];
      }
      result.data[i][k] = sum;
    }
  }
  return result;
}
```

Easy Mode

Linear Algebra

- Well-known, regular structures
  - High gains, moderate effort
  - FP operations always use vector units
  - HPC & benchmark relevant
- 16x16 matrix multiplication
  - Use specialized libs for larger problem sizes!
- Expectation
  - Close to 32 Flops/cycle HW limit

```
struct alignas(64) Matrix
{
  float data[16][16];
};

Matrix mult(const Matrix& lhs, const Matrix& rhs)
{
  Matrix result;
  for (int i = 0; i < 16; ++i) {
    for (int k = 0; k < 16; ++k) {
      float sum = 0;
      for (int n = 0; n < 16; ++n) {
        sum += lhs.data[i][n] * rhs.data[n][k];
      }
      result.data[i][k] = sum;
    }
  }
  return result;
}
```

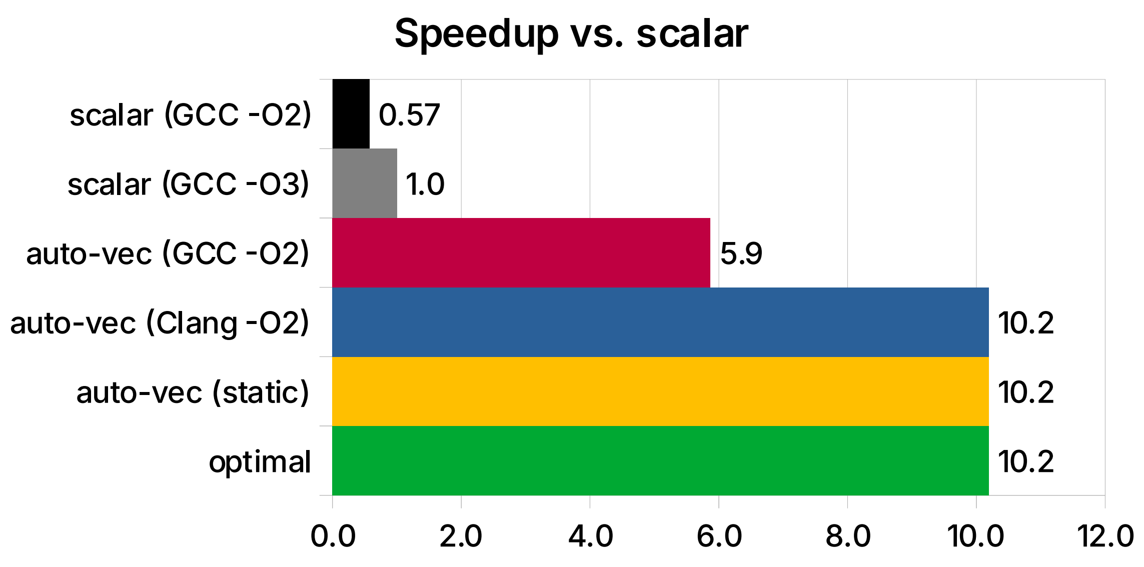
Easy Mode

Generated Code

https://godbolt.org/z/e737edGdY

Easy Mode

Results



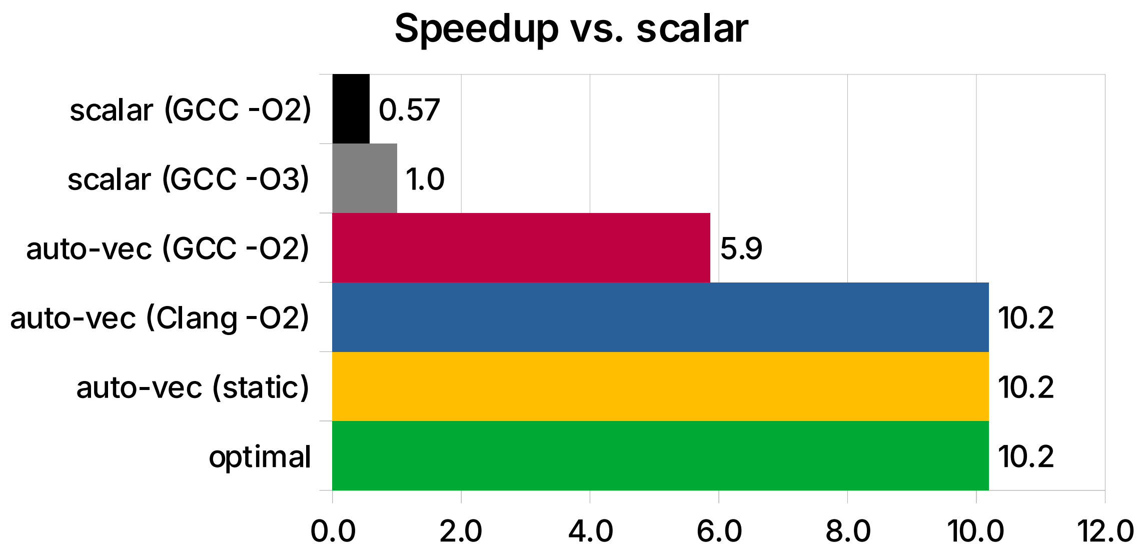
Flags for auto-vectorization:
`-DNDEBUG -march=native -O3 -march=znver4`

Flags for scalar:
```
-DNDEBUG -march=native -O3 -march=znver4 \
-fno-tree-vectorize
```

Easy Mode

Results

- GCC and Clang output near optimal
- GCC may fail to perform pre-load optimization
- Clang -O2 better than GCC -O2



Flags for auto-vectorization:

```
-DNDEBUG -march=native -O3 -march=znver4
```

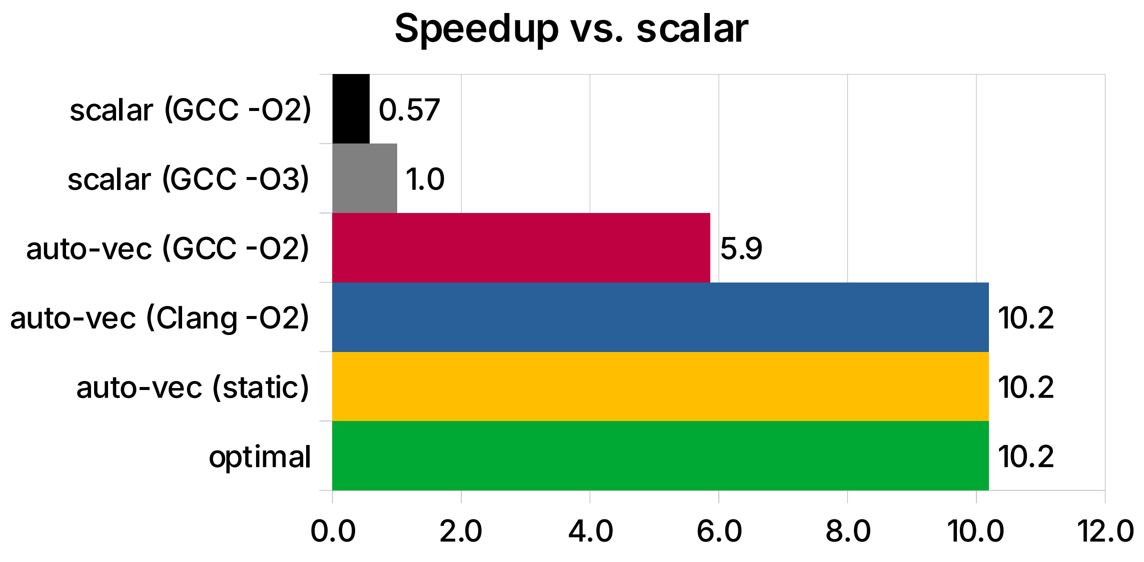
Flags for scalar:

```
-DNDEBUG -march=native -O3 -march=znver4 \
-fno-tree-vectorize
```

Easy Mode

Results

- GCC and Clang output near optimal
- GCC may fail to perform pre-load optimization
- Clang -O2 better than GCC -O2



- Rating:
  75% (log speedup : log optimal)
- Under optimal conditions:
  100%

Flags for auto-vectorization:
```
-DNDEBUG -march=native -O3 -march=znver4
```

Flags for scalar:
```
-DNDEBUG -march=native -O3 -march=znver4 \
-fno-tree-vectorize
```

Easy Mode

Unpack and Transform

- Extract data from odd-sized struct
  - Still simple 1D operation
  - Data not nicely aligned to SIMD chunk
  - Tail handling required

Moderate

Unpack and Transform

- Extract data from odd-sized struct
  - Still simple 1D operation
  - Data not nicely aligned to SIMD chunk
  - Tail handling required
- RGB-to-grayscale conversion
  - Temporary expansion to FP32 changes vector capacity

```
struct RGB
{
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

void rgb2gray(uint8_t* gray,
              const RGB* rgb,
              size_t n)
{
  for (size_t i = 0; i < n; ++i) {
    auto& pixel = rgb[i];
    float gr = 0.299f * pixel.red
             + 0.587f * pixel.green
             + 0.114f * pixel.blue;
    gray[i] = static_cast<uint8_t>(gr + 0.5f);
  }
}
```

Moderate

Unpack and Transform

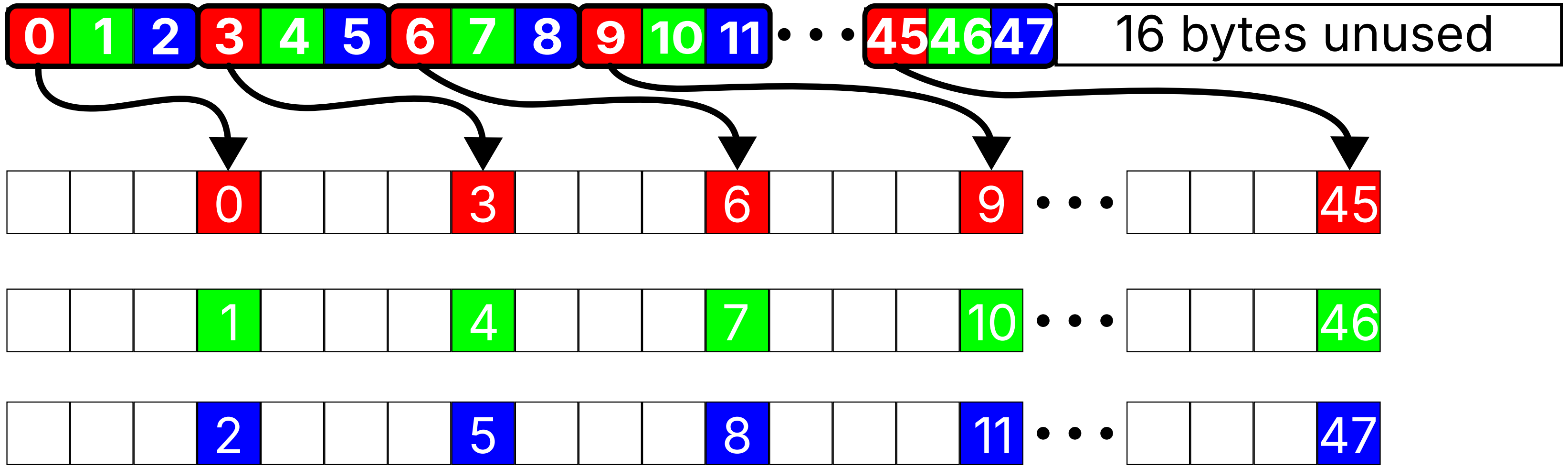
- Extract data from odd-sized struct
  - Still simple 1D operation
  - Data not nicely aligned to SIMD chunk
  - Tail handling required
- RGB-to-grayscale conversion
  - Temporary expansion to FP32 changes vector capacity
- Expectation
  - Optimal: 6 cycles / iteration (16 pixels)
  - Compiler will issue intermediate permutation instructions

```
struct RGB
{
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

void rgb2gray(uint8_t* gray,
              const RGB* rgb,
              size_t n)
{
  for (size_t i = 0; i < n; ++i) {
    auto& pixel = rgb[i];
    float gr = 0.299f * pixel.red
             + 0.587f * pixel.green
             + 0.114f * pixel.blue;
    gray[i] = static_cast<uint8_t>(gr + 0.5f);
  }
}
```

Moderate

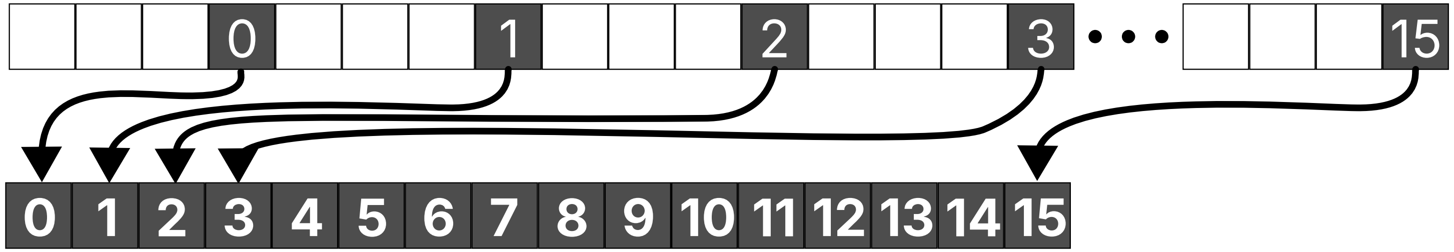
What the Compiler Should Produce



- Masking to load only 16 x 3 bytes
- Tail iteration may require "shorter" mask

- "permute" to pick every 3rd put into every 4th
- Masking to fill leading bytes with 0

convert uint32 → float32
calculate gray value
convert float32 → uint32



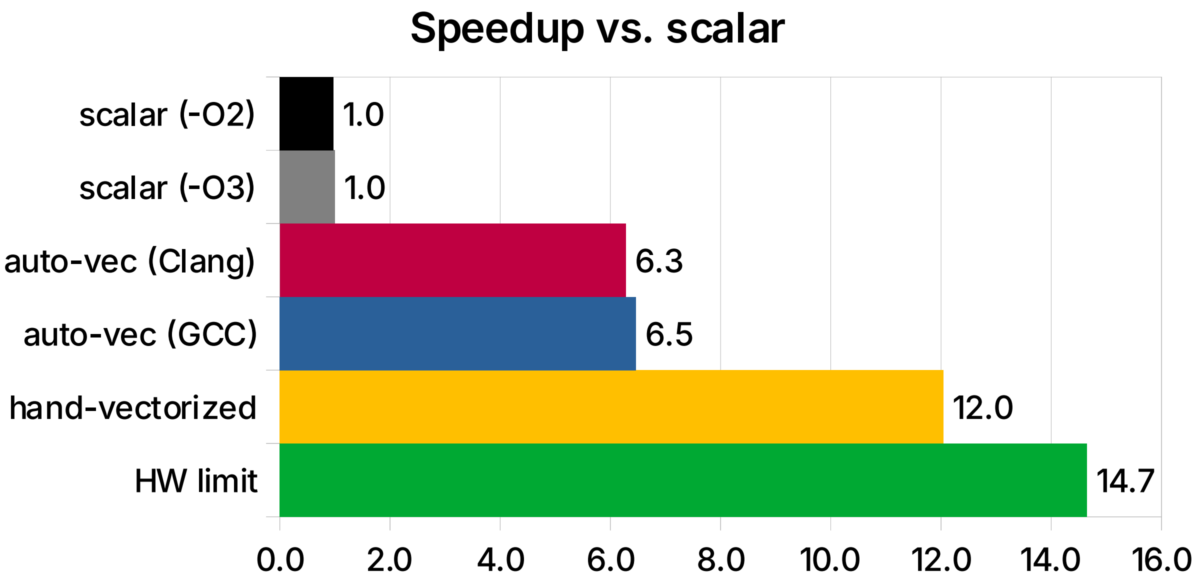
Moderate

Generated Code

https://godbolt.org/z/c5YeaG96s

Moderate

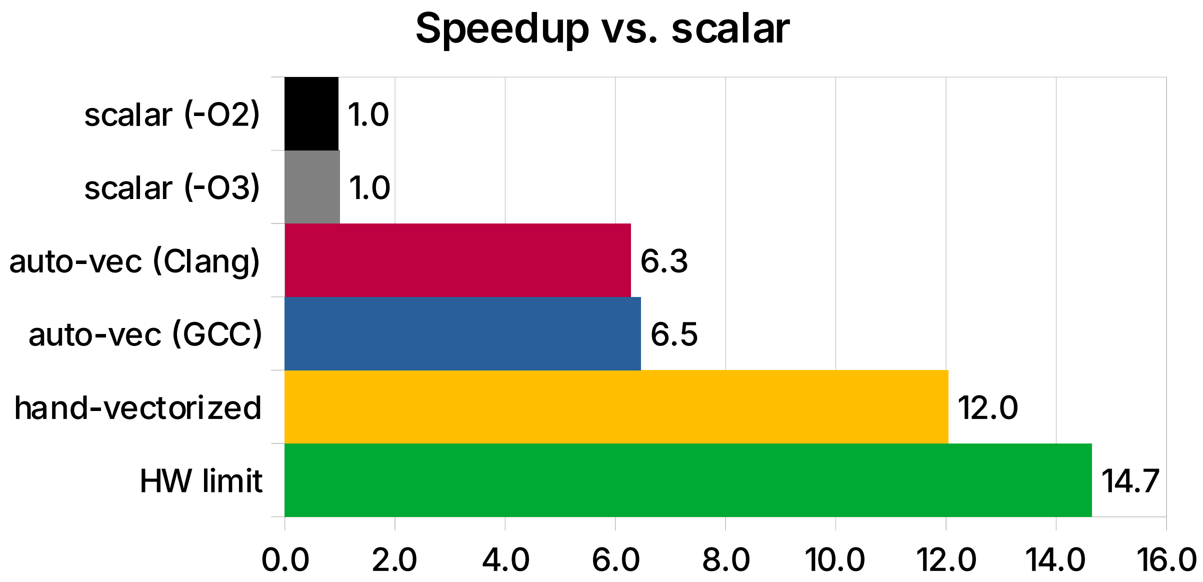
Results



Moderate

Results

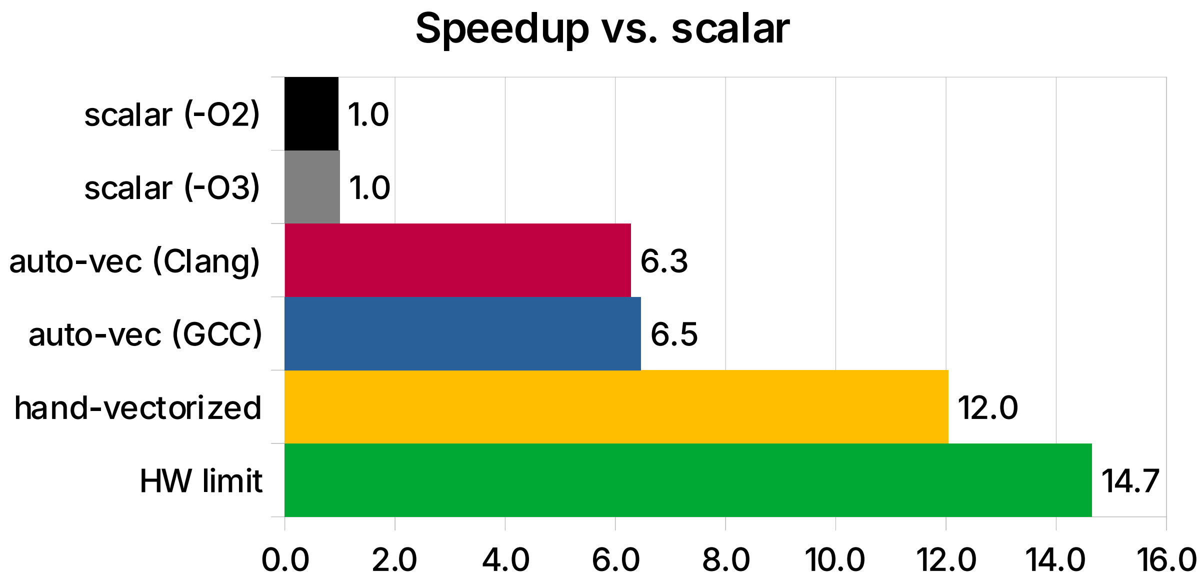
- Only minor differences between GCC and Clang, -O2 and -O3
- Loads 3 chunks instead of just 1 (input vector larger than processing capacity)
- No masked tail handling



Moderate

Results

- Only minor differences between GCC and Clang, -O2 and -O3
- Loads 3 chunks instead of just 1 (input vector larger than processing capacity)
- No masked tail handling



Moderate

- Rating:
  70% of HW limit

2D Operation `std::find_first_of`

- Innermost of nested loops
  - Usually the most costly
  - May be too short for vectorization

Challenging

2D Operation `std::find_first_of`

- Innermost of nested loops
  - Usually the most costly
  - May be too short for vectorization

- Operations on strings
  - Often, much faster solutions exist
  - LibC should cover more of those

```
auto findFirstOf(
  const std::string& s,
  const std::string& to_find)
{
  auto first1 = s.begin();
  auto last1 = s.end();
  auto first2 = to_find.begin();
  auto last2 = to_find.end();

  // logic from glibc std_algo.h's implementation
  // of std::find_first_of()
  for (; first1 != last1; ++first1)
    for (auto it = first2; it != last2; ++it)
      if (*first1 == *iter)
        return first1;

  return last1;
}
```

Challenging

2D Operation `std::find_first_of`

- Innermost of nested loops
  - Usually the most costly
  - May be too short for vectorization
- Operations on strings
  - Often, much faster solutions exist
  - LibC should cover more of those
- Expectation
  - Approach HW limit for long `to_find`
  - HW limit is 64 comparisons/cycle

```
auto findFirstOf(
  const std::string& s,
  const std::string& to_find)
{
  auto first1 = s.begin();
  auto last1 = s.end();
  auto first2 = to_find.begin();
  auto last2 = to_find.end();

  // logic from glibc std_algo.h's implementation
  // of std::find_first_of()
  for (; first1 != last1; ++first1)
    for (auto it = first2; it != last2; ++it)
      if (*first1 == *iter)
        return first1;

  return last1;
}
```

Challenging

Generated Code

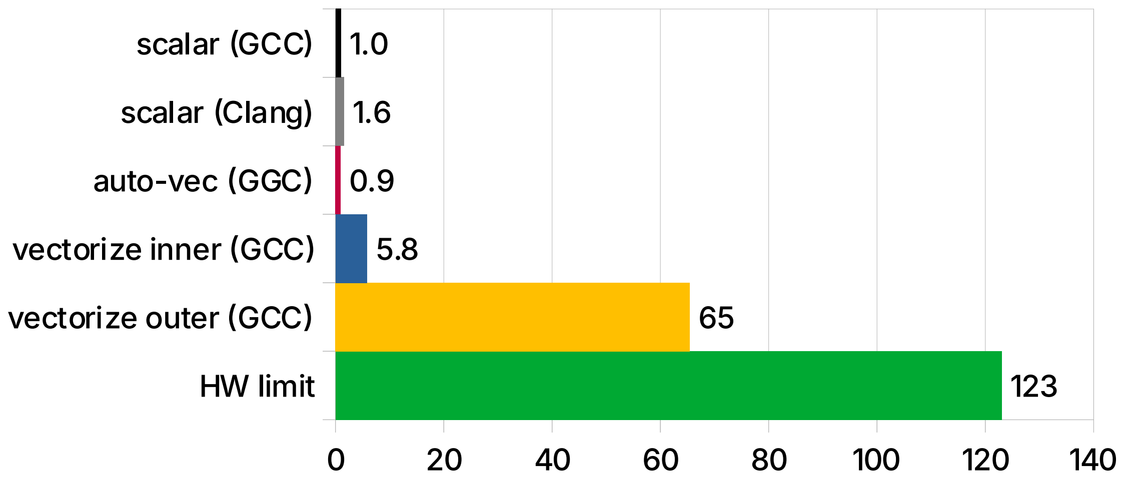
COMPILER EXPLORER

https://godbolt.org/z/344En8nzT

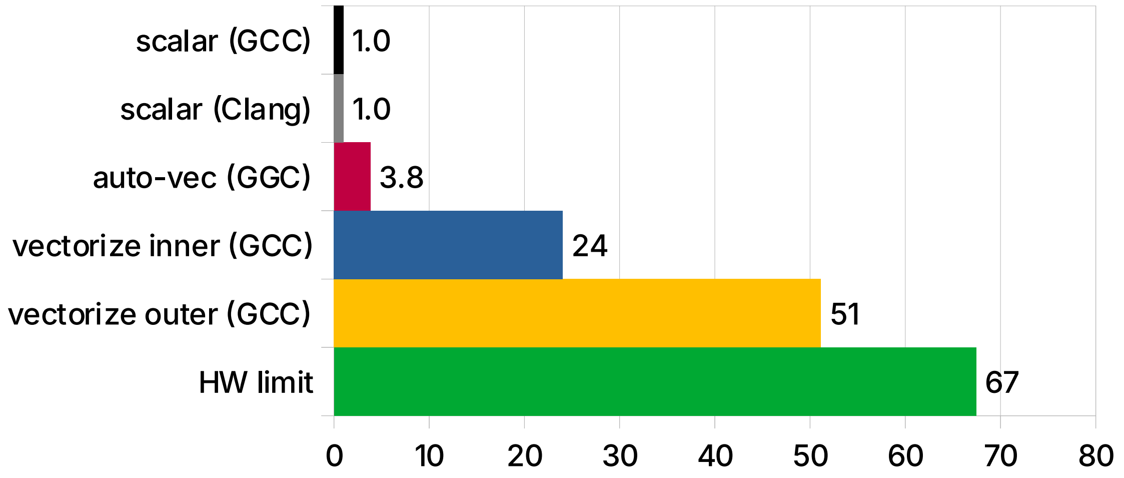
Challenging

Results

Speedup vs. scalar (to_find.len=5)



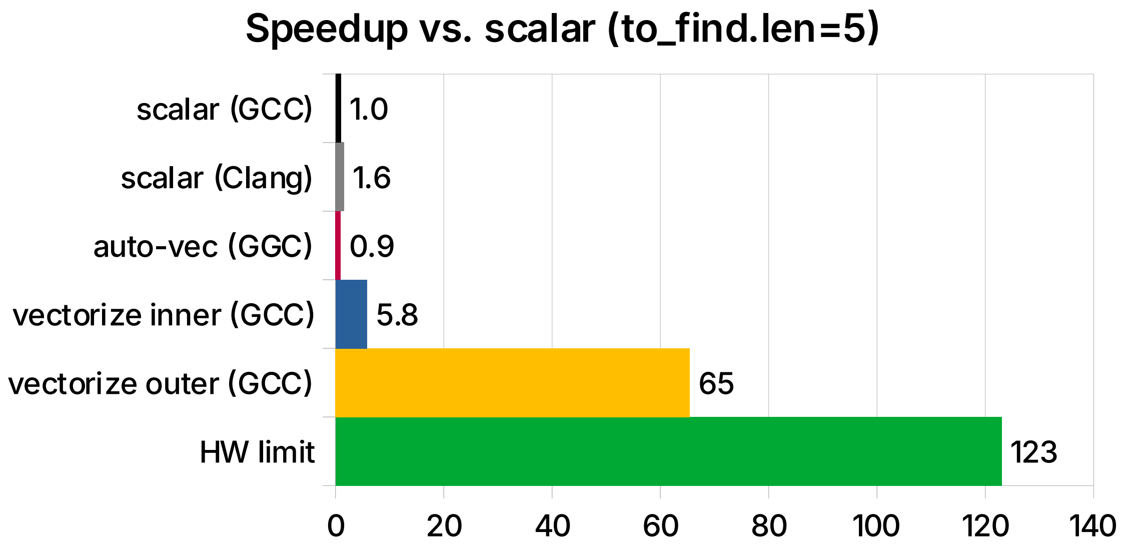
Speedup vs. scalar (to_find.len=160)

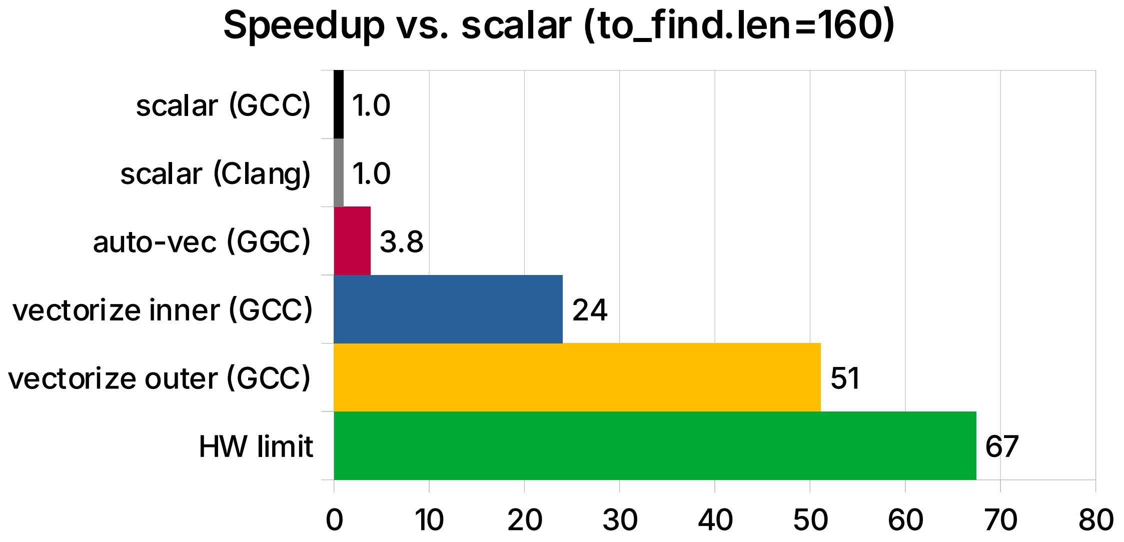


Challenging

Results

- Clang will often not vectorize
- Regression for short,
  modest gains for large inputs
- Effective inner loop vectorization requires masking
- Vectorizing outer loop much more efficient, but also uses masks



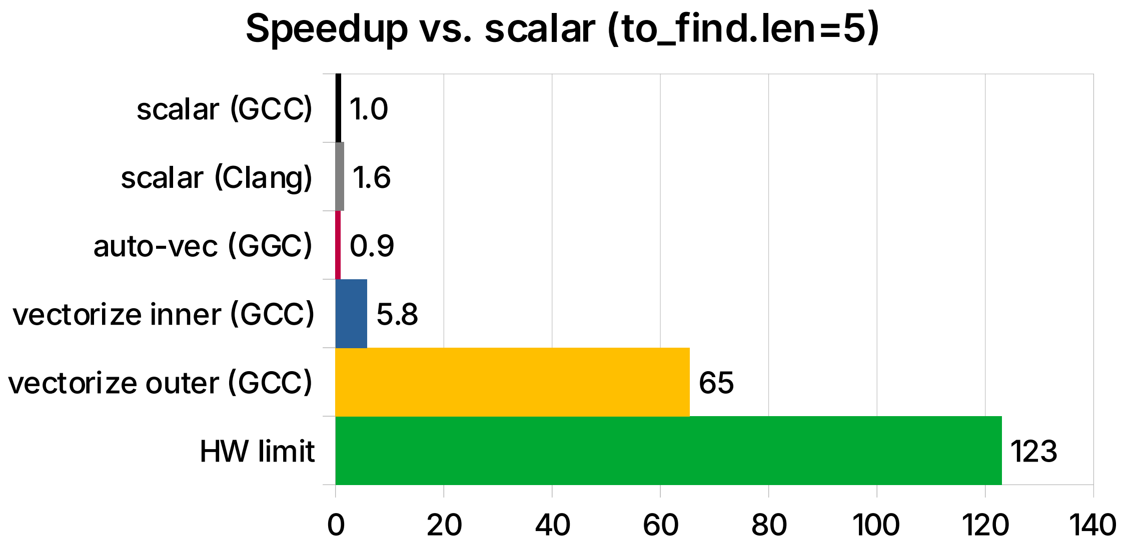


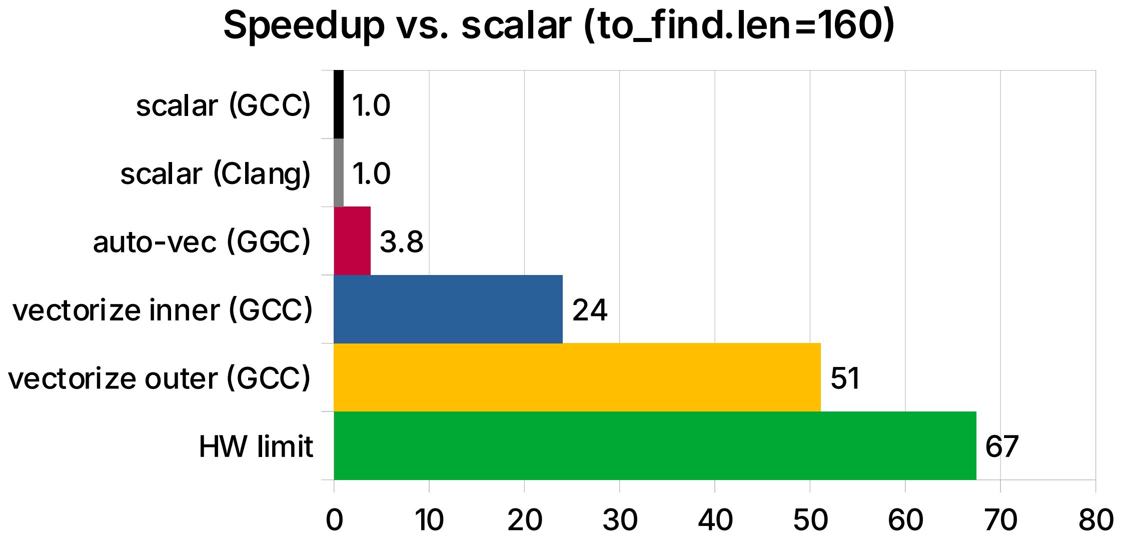
Challenging

Results

- Clang will often not vectorize
- Regression for short,
  modest gains for large inputs
- Effective inner loop vectorization requires masking
- Vectorizing outer loop much more efficient, but also uses masks

- Rating:
  -2% .. 32% of HW limit
- Compared to LibC-like solution:
  -1% .. 17%





Challenging

Compacting Vectors `std::remove_copy_if`

- Frequent in vectorized code
  - Filter, split or reformat data
  - Output shall be compact to process it further with full vector efficiency
  - More efficient than gather / scatter

Hard

Compacting Vectors `std::remove_copy_if`

- Frequent in vectorized code
  - Filter, split or reformat data
  - Output shall be compact to process it further with full vector efficiency
  - More efficient than gather / scatter
- Breaks vector lane mapping
  - More difficult than masking

```
// tell the compiler that input and output
// buffers don't overlap
float* sanitize(float* __restrict__ out,
                const float* __restrict__ in,
                size_t count)
{
  return std::remove_copy_if(
             in,
             in + count,
             out,
             [](auto v) {
                 return v <= 0.f;
             });
}
```

Hard

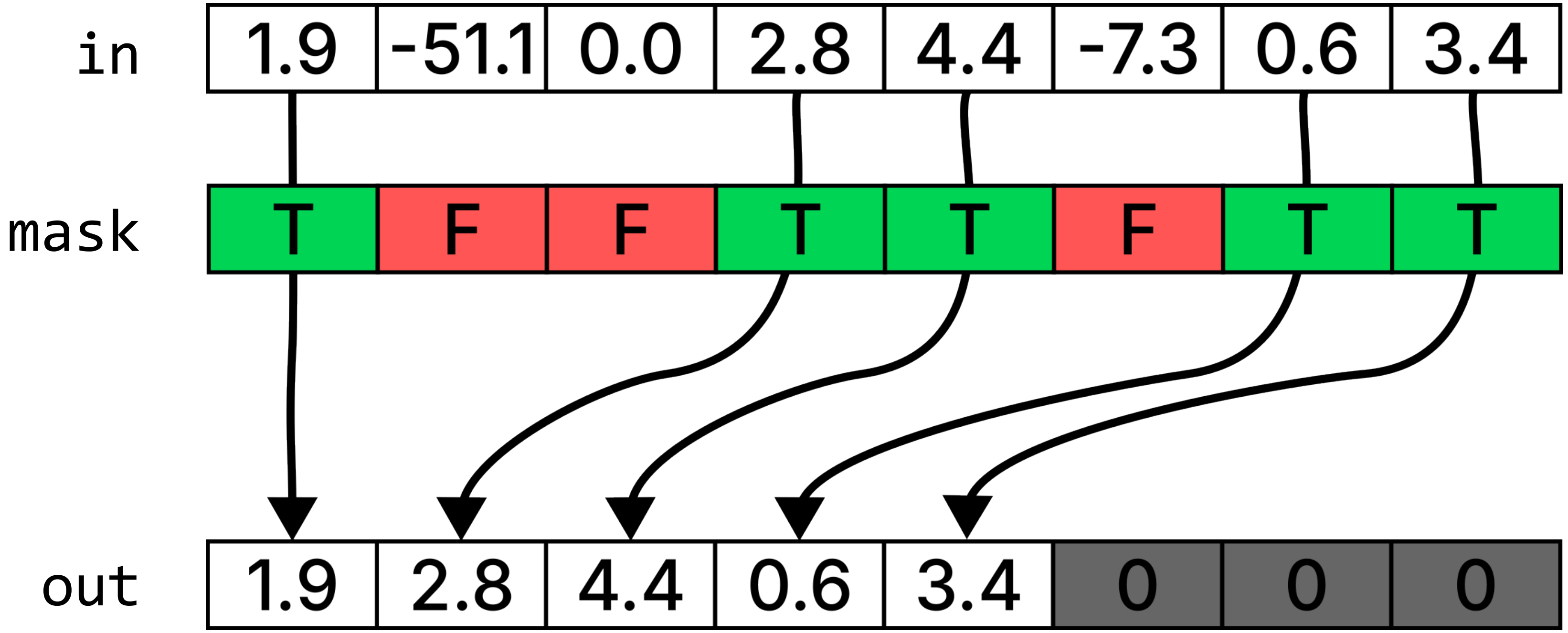
Compacting Vectors `std::remove_copy_if`

- Frequent in vectorized code
  - Filter, split or reformat data
  - Output shall be compact to process it further with full vector efficiency
  - More efficient than gather / scatter
- Breaks vector lane mapping
  - More difficult than masking
- Expectation
  - No auto-vectorization
  - Unaligned store limit: ~3 cycles / 64B

```
// tell the compiler that input and output
// buffers don't overlap
float* sanitize(float* __restrict__ out,
                const float* __restrict__ in,
                size_t count)
{
  return std::remove_copy_if(
              in,
              in + count,
              out,
              [](auto v) {
                  return v <= 0.f;
              });
}
```

Hard

Illustration




```
out = _mm256_maskz_compress_ps(mask, in)
```

Hard

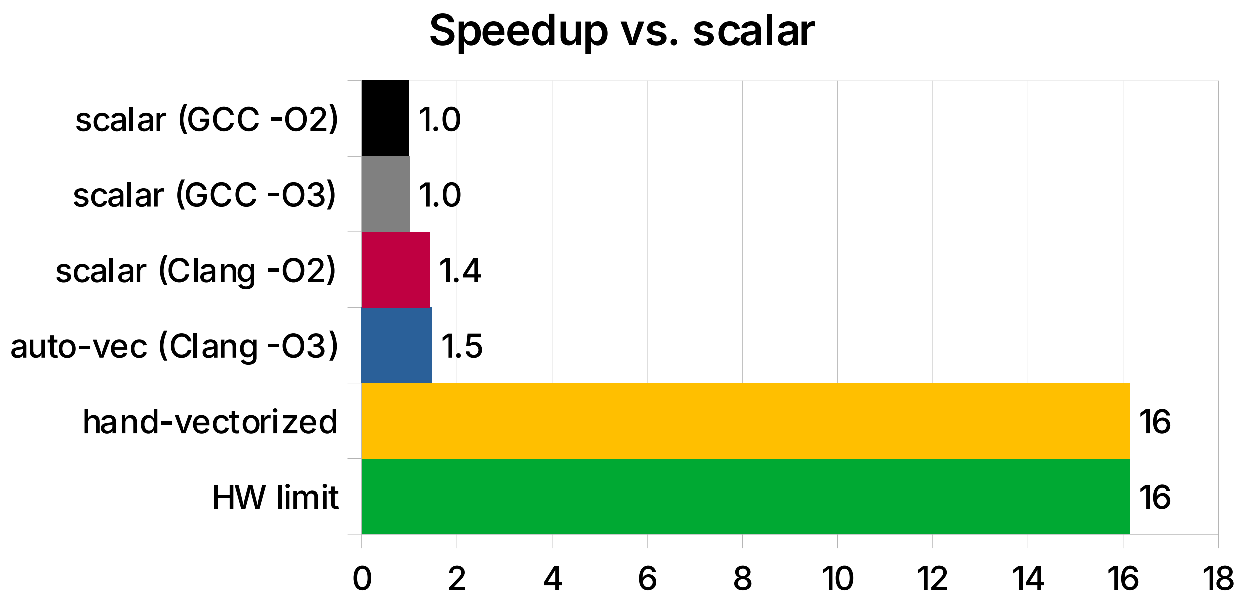
Generated Code

COMPILER EXPLORER

https://godbolt.org/z/xj53fG3nP

Hard

Results

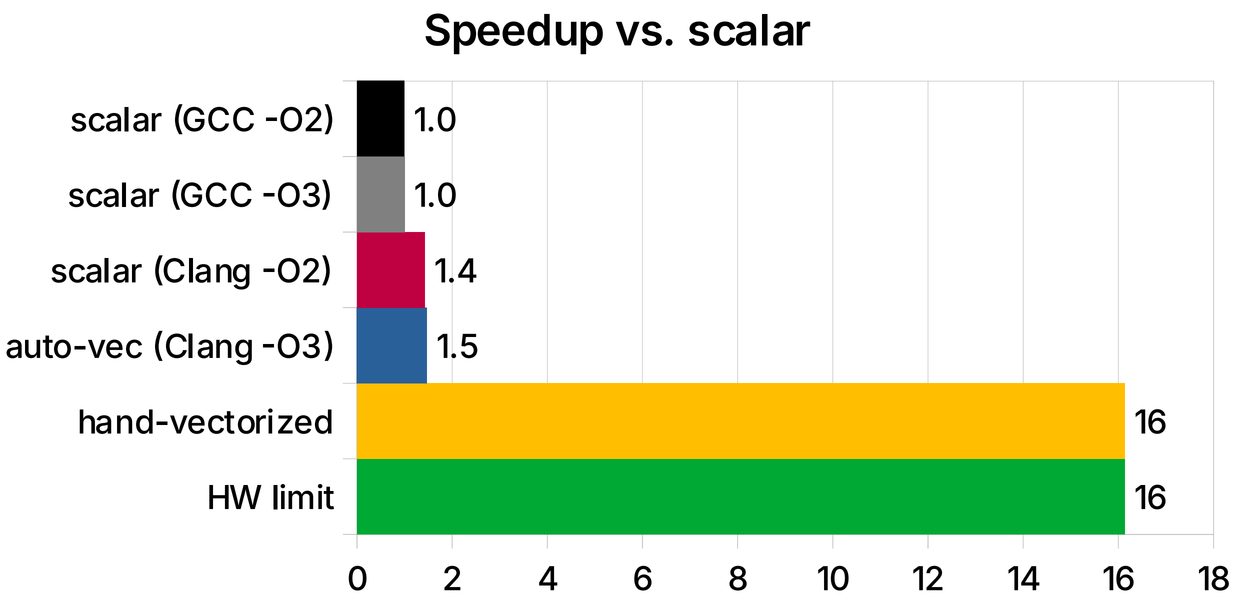


Hard

Results

- Same performance -O2 & -O3
- Clang faster due to unrolling

- No auto-vectorization



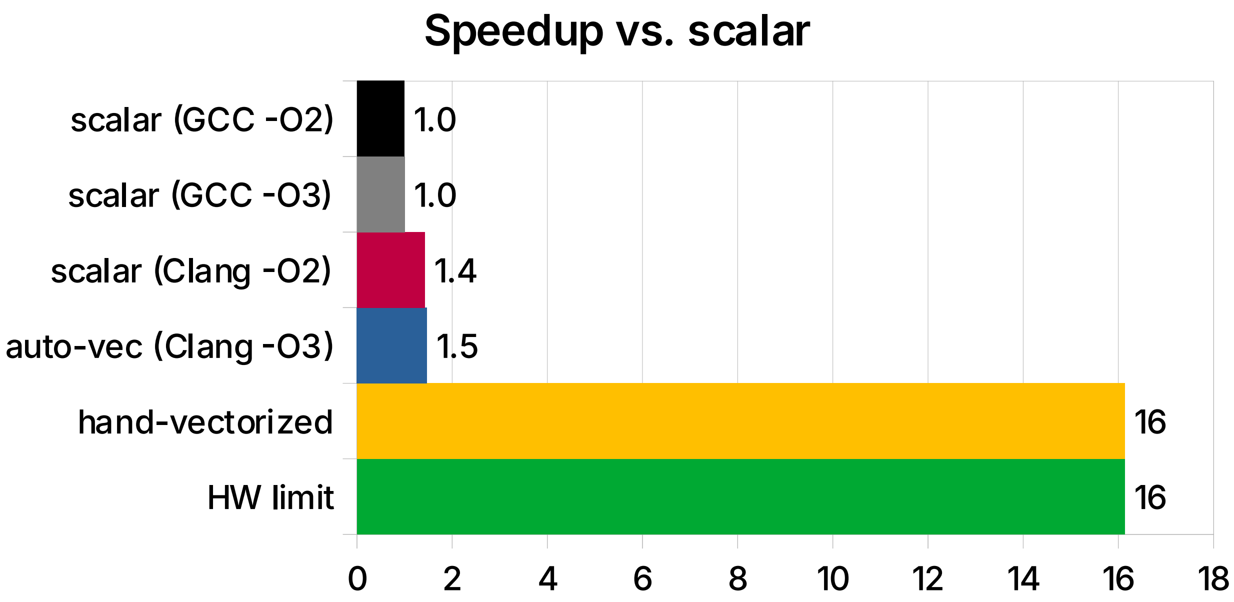
Hard

Results

- Same performance -O2 & -O3
- Clang faster due to unrolling

- No auto-vectorization

- Rating:
  0% of HW limit



Hard

Throughput Relative to GCC -O2

Could it be worth going from GCC -O2 to -O3 or Clang?

		-O2		-O3	
		GCC 15.2.0	Clang 21.1.3	GCC 15.2.0	Clang 21.1.3
scalar	matmul	100%	189%	174%	188%
	rgb2gray	100%	100%	100%	100%
	find /5	100%	123%	85%	135%
	find /160	100%	103%	100%	103%
	compact	100%	143%	100%	140%
matmul	auto-vec	100%	177%	107%	177%
	auto-vec static	100%	176%	174%	176%
	hand-vec	100%	100%	100%	100%
rgb2gray	auto-vec	100%	96%	99%	96%
	hand-vec	100%	96%	99%	96%
find	auto-vec /5	100%	132%	82%	146%
	auto-vec /160	100%	103%	382%	103%
	hand-vec inner /5	100%	104%	100%	104%
	hand-vec inner /160	100%	91%	88%	91%
	hand-vec outer /5	100%	93%	100%	93%
	hand-vec outer /160	100%	66%	100%	66%
	LibC-style /5	100%	101%	100%	102%
compact	auto-vec	100%	150%	100%	147%
	hand-vec	100%	102%	110%	102%

Summary

Conclusion

- Clang has edge over GCC on scalar code and -O2
- Simple, linear loops get meaningful speedups
- Linear algebra saw biggest performance increase (small matrices)

Conclusion

- Clang has edge over GCC on scalar code and -O2
- Simple, linear loops get meaningful speedups
- Linear algebra saw biggest performance increase (small matrices)

- GCC -O3 may still be required for auto-vectorization
- GCC somewhat better at AVX512 code (auto- and hand-vectorized)

Conclusion

- Clang has edge over GCC on scalar code and -O2
- Simple, linear loops get meaningful speedups
- Linear algebra saw biggest performance increase (small matrices)

- GCC -O3 may still be required for auto-vectorization
- GCC somewhat better at AVX512 code (auto- and hand-vectorized)

- Meaningful next step: generating masked loads and stores
- Masked operations in general
- Inner / outer loop trade-offs