SIMD in C++: auto-vectorization in a nutshell

Anthony Boulmier

Université de Genève

anthony.boulmier@unige.ch

October 20, 2020

Outline

1. Introduction

2. How to ?

3. Breaking the code!

4. Benchmarks

5. Conclusion

Motivation

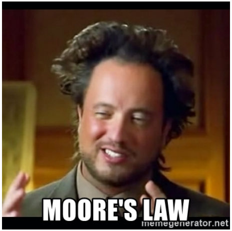
- CPU clock rate does not increase anymore

Motivation

- CPU clock rate does not increase anymore
- More cores → more transistors !

Motivation

- CPU clock rate does not increase anymore
- More cores → more transistors !
- Or ? Do multiple instructions at the same time!



What is SIMD ?

SIMD stands for: **S**ingle **I**nstruction **M**ultiple **D**ata

- Same operation on multiple values (think about element-wise vector operations)
  Example: You want to sum the forces (XYZ) for 100K particles. Do 4, 8, or 16 summations at the same time instead of 1.

What is SIMD ?

SIMD stands for: **S**ingle **I**nstruction **M**ultiple **D**ata

- Same operation on multiple values (think about element-wise vector operations)
  Example: You want to sum the forces (XYZ) for 100K particles. Do 4, 8, or 16 summations at the same time instead of 1.
- Recent CPUs have an instruction set that works on **vector registers** (*%xmm*, *%ymm*, and *%zmm* registers, note: xmm are also used for scalar operations)

What is SIMD ?

SIMD stands for: **S**ingle **I**nstruction **M**ultiple **D**ata

- Same operation on multiple values (think about element-wise vector operations)
  Example: You want to sum the forces (XYZ) for 100K particles. Do 4, 8, or 16 summations at the same time instead of 1.
- Recent CPUs have an instruction set that works on **vector registers** (*%xmm*, *%ymm*, and *%zmm* registers, note: xmm are also used for scalar operations)
- Up to 16x performance improvement and it is almost free lunch !
  Codes that benefit from vector instructions are called vectorized code.

Vector registers

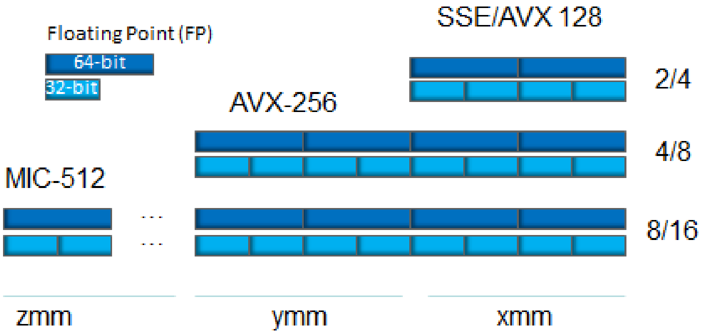
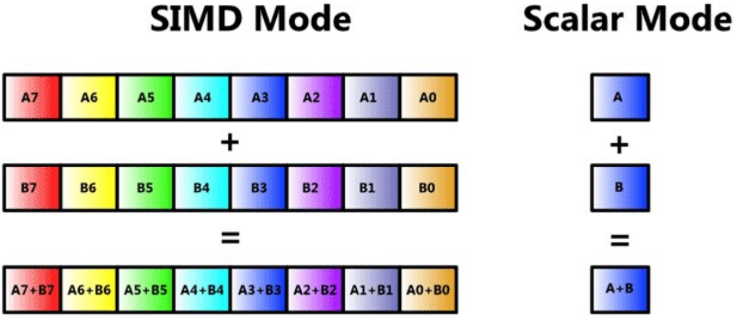


Figure: Vector registers, source: Cornell

What is SIMD ?


SIMD Mode
Scalar Mode
A7
A6
A5
A4
A3
A2
A1
A0
+
B7
B6
B5
B4
B3
B2
B1
B0
=
A7+B7
A6+B6
A5+B5
A4+B4
A3+B3
A2+B2
A1+B1
A0+B0
A
+
B
=
A+B

A LOT can be done with SIMD

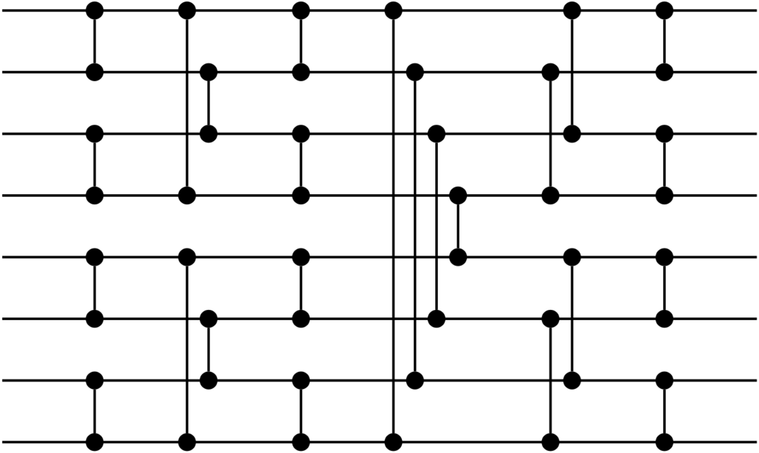


Figure: What is this?

How to write auto-vectorized code?

Some portion of your codes may already benefits from it!
Compilers “auto-vectorize” under the following constraints:

- Can only vectorize for-loop with a fixed (known at the beginning) number of iterations

How to write auto-vectorized code?

Some portion of your codes may already benefits from it!
Compilers “auto-vectorize” under the following constraints:

- Can only vectorize for-loop with a fixed (known at the beginning) number of iterations
- No dependency between array indices (iterations are independent)

How to write auto-vectorized code?

Some portion of your codes may already benefits from it!
Compilers “auto-vectorize” under the following constraints:

- Can only vectorize for-loop with a fixed (known at the beginning) number of iterations
- No dependency between array indices (iterations are independent)
- No conditional branching

How to write auto-vectorized code?

Some portion of your codes may already benefits from it!
Compilers “auto-vectorize” under the following constraints:

- Can only vectorize for-loop with a fixed (known at the beginning) number of iterations
- No dependency between array indices (iterations are independent)
- No conditional branching
- The compiler is sure that the performance will increase (use of cost function)

How to write auto-vectorized code?

Some portion of your codes may already benefits from it!
Compilers “auto-vectorize” under the following constraints:

- Can only vectorize for-loop with a fixed (known at the beginning) number of iterations
- No dependency between array indices (iterations are independent)
- No conditional branching
- The compiler is sure that the performance will increase (use of cost function)
- Only vectorize inner loops

Rule of thumb: if you can’t tell how to vectorize a code, neither can the compiler.

Write and debug vectorized code

1. Write auto-vectorization friendly codes
   Array of Structure (AoS) vs. Structure of Array (SoA)
2. Use the proper compilation line
   Vectorization is enabled by default at *-O3*
   gcc -O2 -ftree-vectorize -fopt-info-vec-{all,missed,optimized}
   clang -O2 -ftree-vectorize -Rpass-analysis=loop-vectorize -Rpass=loop-vectorize
3. Look at assembly code
   -S -o main.s

Auto-vectorization friendly code

Example

Listing 1: AoS

```
const auto N = 100000;
struct Body {
    std::array<float, 2> v, p /*, ... */;
};
std::array<Body, N> bodies;
```

Example

Listing 2: SoA

```cpp
const auto N = 100000;
struct Bodies {
    std::array<float, N> vx, px, vy, py /*, ... */;
};
Bodies bodies;
```

AoS

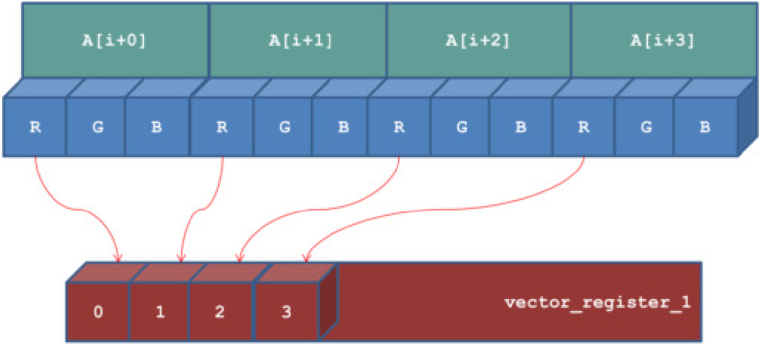


Figure: non-coalesced memory access, source: Jeffers et al. 2016

Simple code

Example

Listing 3: Vectorization example

```
/* ... */
float f(float * a, float * b){
    #pragma GCC ivdep
    for(int i=0;i < 10000000; ++i)
      a[i] = a[i] + b[i];
    return a[50000];
}
/* ... */
```

godbolt.org: online disassembler

Simple code

Behind the scene (non-vectorized)

Listing 4: Vectorization example

```
.L2:
      movss    xmm0, DWORD PTR [rdi+rax]
      addss    xmm0, DWORD PTR [rsi+rax]
      movss    DWORD PTR [rdi+rax], xmm0
      add      rax, 4
      cmp      rax, 40000000
      jne      .L2
```

In godbolt.org: gcc-10 -O2

Simple code

Behind the scene (vectorized)

Listing 5: Vectorization example

```
.L2:
    vmovups zmm1, ZMMWORD PTR [rsi+rax]
    vaddps  zmm0, zmm1, ZMMWORD PTR [rdi+rax]
    vmovups ZMMWORD PTR [rdi+rax], zmm0
    add     rax, 64
    cmp     rax, 40000000
    jne     .L2
```

In godbolt.org: gcc-10 -O2 -ftree-vectorize -mavx512f

Simple code

Behind the scene (vectorized)

Listing 6: Vectorization example

```
.L2:
    vmovups zmm1, ZMMWORD PTR [rsi+rax]
    vaddps  zmm0, zmm1, ZMMWORD PTR [rdi+rax]
    vmovups ZMMWORD PTR [rdi+rax], zmm0
    add     rax, 64
    cmp     rax, 40000000
    jne     .L2
```

In godbolt.org: gcc-10 -O2 -ftree-vectorize -mavx512f

What if the array size in byte is not divisible by the vector register size?

Simple code

Behind the scene (vectorized)

Listing 7: Vectorization example

```
.L2:
    vmovups zmm1, ZMMWORD PTR [rsi+rax]
    vaddps  zmm0, zmm1, ZMMWORD PTR [rdi+rax]
    vmovups ZMMWORD PTR [rdi+rax], zmm0
    add     rax, 64
    cmp     rax, 40000000
    jne     .L2
```

In godbolt.org: gcc-10 -O2 -ftree-vectorize -mavx512f

What if the array size in byte is not divisible by the vector register size?

The compiler executes the rest in scalar mode.

Note: unaligned calls are equivalent to aligned code since *Intel Sandy Bridge*: see, Agner Fog

Exercise

What does the compiler say?

Listing 8: Vectorization example

```cpp
void f(float* A, float* B, int N){
    for(int i = 0; i < N; ++i)
        A[i] = A[i] * B[i];
}
```

```
g++-10 main.cpp -O3 -mavx512f -fopt-info-vec-all -o main
```

Exercise

What does the compiler say?

Listing 9: Vectorization example

```
void f(float* A, float* B, int N){
    for(int i = 0; i < N; ++i)
        A[i] = A[i] * B[i];
}
```

g++-10 main.cpp -O3 -mavx512f -fopt-info-vec-all -o main

```
main.cpp:3:19: optimized: loop vectorized using 64 byte vectors
main.cpp:3:19: optimized:  loop versioned for vectorization
because of possible aliasing
```

The compiler built a vectorized version, but it had to add aliasing checks

Exercise

What does the compiler say?

Listing 10: Vectorization example

```cpp
void f(float* A, float* B, int N){
#pragma GCC ivdep
    for(int i = 0; i < N; ++i)
        A[i] = A[i] * B[i];
}
```

gcc-10 main.cpp -O3 -mavx512f -fopt-info-vec-all -o main

Exercise

What does the compiler say?

Listing 11: Vectorization example

```
void f(float* A, float* B, int N){
#pragma GCC ivdep
    for(int i = 0; i < N; ++i)
        A[i] = A[i] * B[i];
}
```

gcc-10 main.cpp -O3 -mavx512f -fopt-info-vec-all -o main

```
main.cpp:3:19: optimized: loop vectorized using 64 byte vectors
```

Compiled to avx512 instruction set !

Simple code

What does the compiler say?

Listing 12: Vectorization example

```cpp
float f(float* A, float* B, int N){
#pragma GCC ivdep
    for(int i = 1; i < N; ++i)
        A[i] = A[i-1] * B[i-1];
    return A[0];
}
```

gcc-10 main.cpp -O3 -mavx512f -fopt-info-vec-all -o main

Simple code

What does the compiler say?

Listing 13: Vectorization example

```
float f(float* A, float* B, int N){
#pragma GCC ivdep
    for(int i = 1; i < N; ++i)
        A[i] = A[i-1] * B[i-1];
    return A[0];
}
```

gcc-10 main.cpp -O3 -mavx512f -fopt-info-vec-all -o main

```
main.cpp:5:18: missed: could not vectorize loop
main.cpp:6:21: missed: not vectorized: no vectype
for stmt: _8=*_7
```

Simple code

What does the compiler say?

Listing 14: Vectorization example

```
float f(float* A, float* B, int N){
#pragma GCC ivdep
    for(int i = 1; i < N; ++i)
        if(A[i] < 3) A[i] = A[i] * B[i];
    return A[0];
}
```

gcc-10 main.cpp -O3 -mavx512f -fopt-info-vec-all -o main

Simple code

What does the compiler say?

Listing 15: Vectorization example

```
float f(float* A, float* B, int N){
#pragma GCC ivdep
    for(int i = 1; i < N; ++i)
        if(A[i] < 3) A[i] = A[i] * B[i];
    return A[0];
}
```

gcc-10 main.cpp -O3 -mavx512f -fopt-info-vec-all -o main

```
main.cpp:5:18: missed: could not vectorize loop
main.cpp:5:18: missed: not vectorized: control flow in loop.
```

Benchmark description

- Two arrays (x,y) containing $S = 10^7$ float (32 bits).
- Iterate over the arrays and execute a function that costs X flops. The function is generated and inlined at compile time using *constexpr*.
- Do this T times and compute min, max, and average computing time in serial mode and (auto)vectorized mode.

Benchmark description

- Two arrays (x,y) containing $S = 10^7$ float (32 bits).
- Iterate over the arrays and execute a function that costs X flops. The function is generated and inlined at compile time using *constexpr*.
- Do this T times and compute min, max, and average computing time in serial mode and (auto)vectorized mode.
- Function MULSUM:
  $f_{\text{MUL}}(X, x_i, y_i) = y_i * f_{\text{MUL}}(X - 1, y_i, x_i)$
  $f_{\text{MUL}}(0, x_i, y_i) = y_i$
  Example: $f(2, x_i, y_i) = y_i * x_i * y_i$

Benchmark description

- Two arrays (x,y) containing $S = 10^7$ float (32 bits).
- Iterate over the arrays and execute a function that costs X flops. The function is generated and inlined at compile time using *constexpr*.
- Do this T times and compute min, max, and average computing time in serial mode and (auto)vectorized mode.
- Function MULSUM:
  $f_{\mathrm{MUL}}(X, x_i, y_i) = y_i * f_{\mathrm{MUL}}(X - 1, y_i, x_i)$
  $f_{\mathrm{MUL}}(0, x_i, y_i) = y_i$
  Example: $f(2, x_i, y_i) = y_i * x_i * y_i$
- Function FMASUM:
  $f_{\mathrm{FMA}}(X, x_i, y_i) = y_i + \sum_1^X x_i y_i$
  Example: $f_{\mathrm{FMA}}(2, x_i, y_i) = y_i + x_i y_i + x_i y_i$
  Note: I consider *fused multiply add (FMA)* to cost 1 flop.

Benchmark

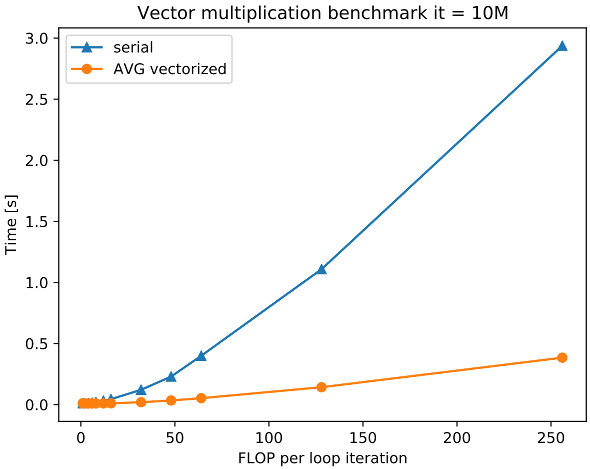


Figure: Time, gcc-10, Intel i7-10750H (Comet Lake): with avx2 (ymm 256 bits)

Benchmark

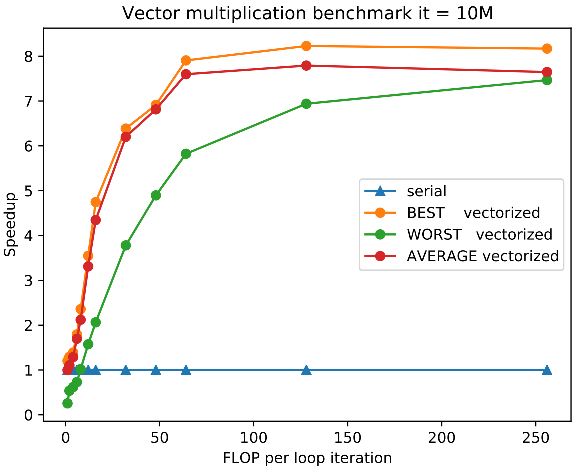


Figure: Speedup, gcc-10, Intel i7-10750H (Comet Lake): with avx2 (ymm 256 bits)

Benchmark

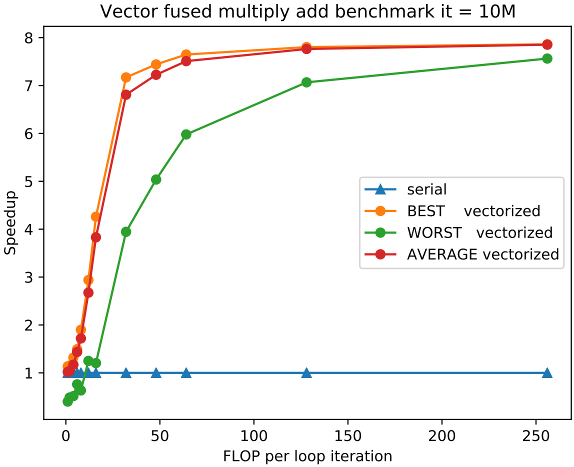


Figure: Speedup, gcc-10, Intel i7-10750H (Comet Lake): with avx2 (ymm 256 bits)

clang 10: huho

```
main.cpp:69:3: remark: the cost-model indicates that interleaving is not
beneficial [-Rpass-analysis=loop-vectorize]
```

Benchmark

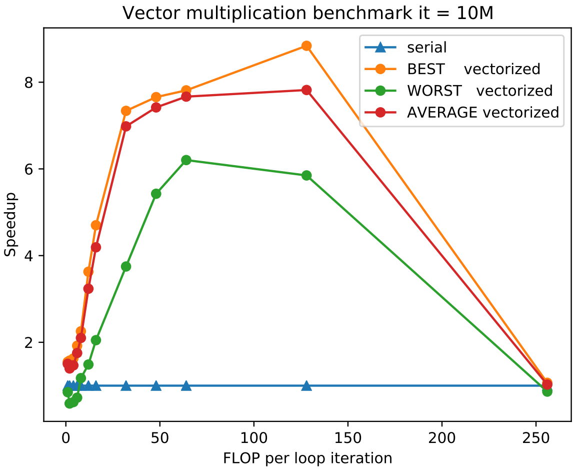


Figure: Speedup, clang-10, Intel i7-10750H (Comet Lake): with avx2 (ymm 256 bits)

Benchmark summary

1. Bad news 1: when there is a few FLOP per iteration, the computation is bounded by memory transfer. Push/Pull to/from vector registers can kill your perfs!

Benchmark summary

1. Bad news 1: when there is a few FLOP per iteration, the computation is bounded by memory transfer. Push/Pull to/from vector registers can kill your perfs!
2. Bad news 2: the compiler can mispredict performance and it may not vectorize, even straightforward code.

Benchmark summary

1. Bad news 1: when there is a few FLOP per iteration, the computation is bounded by memory transfer. Push/Pull to/from vector registers can kill your perfs!
2. Bad news 2: the compiler can mispredict performance and it may not vectorize, even straightforward code.
3. Good news: you can still sometimes be N times faster. $N = \frac{\text{register size}}{\text{type size}}$

Conclusion

- SIMD requires the proper data structure and fine-grained data parallelism
- Using vector registers and auto-vectorization, this can increase performance of data-parallel loop by a factor 4, 8, 16
- The compiler can auto-vectorize under *many* constraints, some code may be impossible to auto-vectorize
- The compiler can be wrong
- Debugging auto-vectorized code is **tedious**
- Be careful about the overhead for vectorization (push/pull to/from vector registers)

Next talk

If you want something done right, you have to do it yourself!