Practical vectorization

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

Thematic CERN School of Computing 2021

Outline

Goal of this course

- Make the theory explained by Andrzej concerning SIMD and vectorization more concrete
- Detail the impact of vectorization on your code
  - on your data model
  - on actual C^{++}code
- Give an idea of what to expect from vectorized code

SIMD - Single Instruction Multiple Data

Concept

- Run the same operation in parallel on multiple data
- Operation is as fast as in single data case
- The data leave in a "vector"

Practically

$$\boxed{A} + \boxed{B} = \boxed{R} \quad \Rightarrow \quad \begin{array}{|c|}\hline A^1 \\ \hline A^2 \\ \hline A^3 \\ \hline A^4 \\ \hline \end{array} + \begin{array}{|c|}\hline B^1 \\ \hline B^2 \\ \hline B^3 \\ \hline B^4 \\ \hline \end{array} = \begin{array}{|c|}\hline R^1 \\ \hline R^2 \\ \hline R^3 \\ \hline R^4 \\ \hline \end{array}$$

Promises of vectorization

Theoretical gains

- Computation speed up corresponding to vector width
- Note that it's dependant on the type of data
  - float vs double
  - shorts versus ints

Various units for various vector width

Name	Arch	nb bits	nb floats/int	nb doubles/long
SSE[1] 4	X86	128	4	2
AVX[2]	X86	256	8	4
AVX[2] 2 (FMA)	X86	256	8	4
AVX[2] 512	X86	512	16	8
SVE[3]	ARM	128-2048	4-64	2-32

[1] Streaming SIMD Extensions [2] Advanced Vector eXtension [3] Scalable Vector Extension

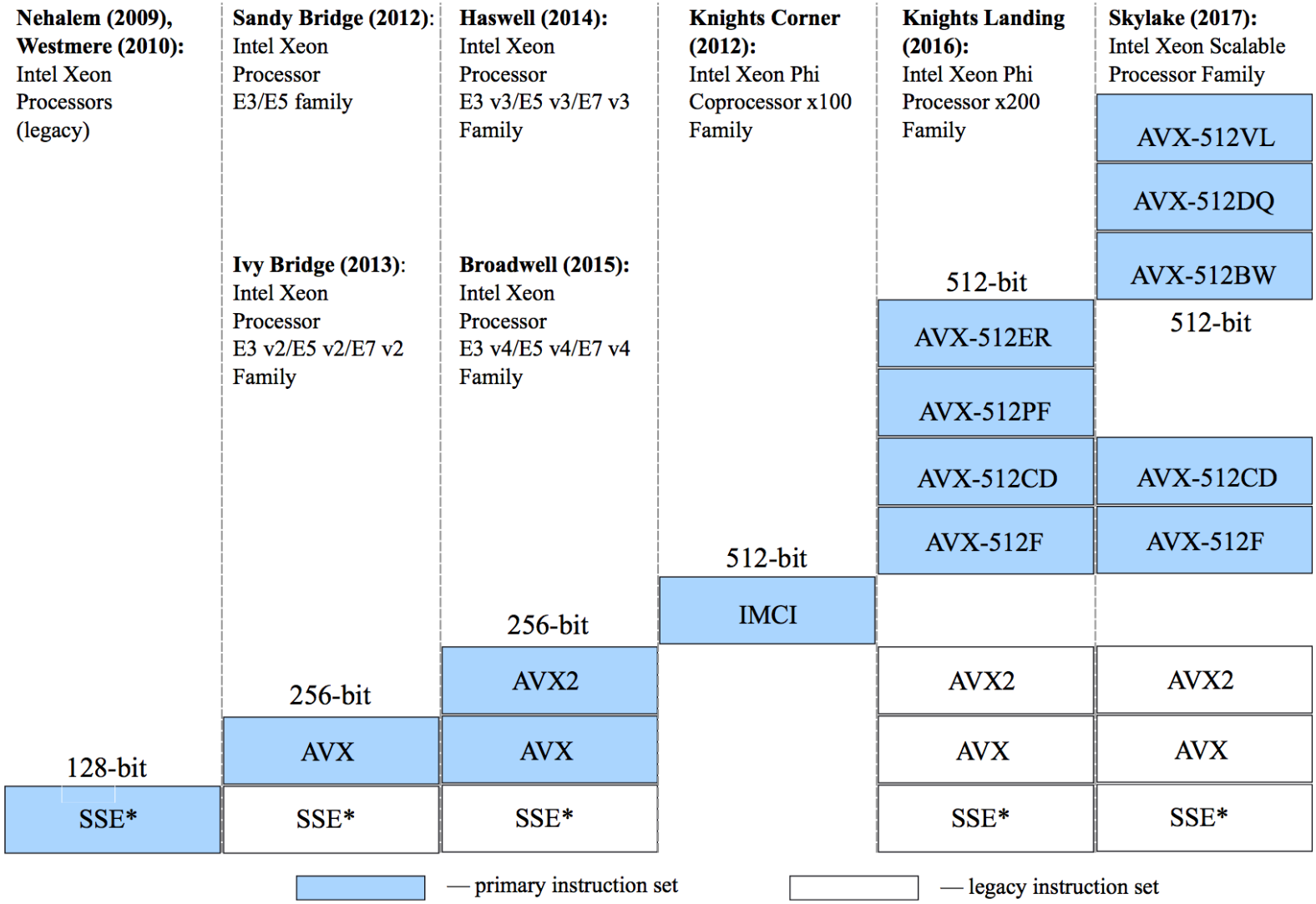
How to now what you can use

Manually

Look for sse, avx, etc in your processor flags

```
lscpu | egrep ``mmx|sse|avx''
Flags: fpu vme de pse tsc msr pae mce cx8 apic sep
       mtrr pge mca cmov pat pse36 clflush dts acpi
       mmx fxsr sse sse2 ss ht tm pbe syscall nx
       rdtscp lm constant_tsc arch_perfmon pebs bts
       rep_good nopl xtopology nonstop_tsc cpuid
       aperfmperf pni pclmulqdq dtes64 monitor ds_cpl
       vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid
       sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer
       aes xsave avx f16c rdrand lahf_lm cpuid_fault
       epb pti ibrs ibpb stibp tpr_shadow vnmi
       flexpriority ept vpid fsgsbase smep erms
       xsaveopt dtherm ida arat pln pts
```

Situation for Intel processors



Am I using vector registers ?

Yes you are

- As vector registers are used for scalar operations
- Remember Andrzej's picture

Am I efficiently using vector registers ?

- Here we have to look at the generated assembly code
- Looking for specific intructions
- Or for the use of specific names of registers

Side note : what to look at ?

What you should look at

- Specific, CPU intensive pieces of code
- The most time consuming functions
- Very small subset of your code (often $< 5\%$)

Where you should not waste your time

- Try to have an overall picture of vectorization in your application
- As most of the code won't use vectors anyway

Crash course in SIMD assembly

Register names

- SSE : xmm0 to xmm15 (128 bits)
- AVX2 : ymm0 to ymm15 (256 bits)
- AVX512 : zmm0 to zmm31 (512 bits)

In scalar mode, SSE registers are used

floating point instruction names

where `<op><simd or not><raw type>`

- `<op>` is something like `vmul`, `vadd`, `vmov` or `vfmadd`
- `<simd or not>` is either 's' for scalar or 'p' for packed (i.e. vector)
- `<raw type>` is either 's' for single precision or 'd' for double precision

Typically :

`vmulss`, `vmovaps`, `vaddpd`, `vfmaddpd`

Practical look at assembly

Extract assembly code

- Run objdump -d -C on your executable or library
- Search for your function name

Check for vectorization

- For avx2, look for ymm
- For avx512, look for zmm
- Otherwise look for instructions with ps or pd at the end
  - but ignore mov operations
  - only concentrate on arithmetic ones

Exercise 1

Code

```
d18:   c5 fc 59 d8       vmulps %ymm0,%ymm0,%ymm3
d1c:   c5 fc 58 c0       vaddps %ymm0,%ymm0,%ymm0
d20:   c5 e4 5c de       vsubps %ymm6,%ymm3,%ymm3
d24:   c4 c1 7c 59 c0    vmulps %ymm8,%ymm0,%ymm0
d29:   c4 c1 64 58 da    vaddps %ymm10,%ymm3,%ymm3
d2e:   c4 41 7c 58 c3    vaddps %ymm11,%ymm0,%ymm8
d33:   c5 e4 59 d3       vmulps %ymm3,%ymm3,%ymm2
d37:   c4 c1 3c 59 f0    vmulps %ymm8,%ymm8,%ymm6
d3c:   c5 ec 58 d6       vaddps %ymm6,%ymm2,%ymm2
```

Solution

- Presence of ymm
- Vectorized, AVX level

Exercise 2

Code

```
b97:   0f 28 e5          movaps %xmm5,%xmm4
b9a:   f3 0f 59 e5       mulss  %xmm5,%xmm4
b9e:   f3 0f 58 ed       addss  %xmm5,%xmm5
ba2:   f3 0f 59 ee       mulss  %xmm6,%xmm5
ba6:   f3 0f 5c e7       subss  %xmm7,%xmm4
baa:   0f 28 f5          movaps %xmm5,%xmm6
bad:   f3 41 0f 58 e0    addss  %xmm8,%xmm4
bb2:   f3 0f 58 f2       addss  %xmm2,%xmm6
bb6:   0f 28 ec          movaps %xmm4,%xmm5
```

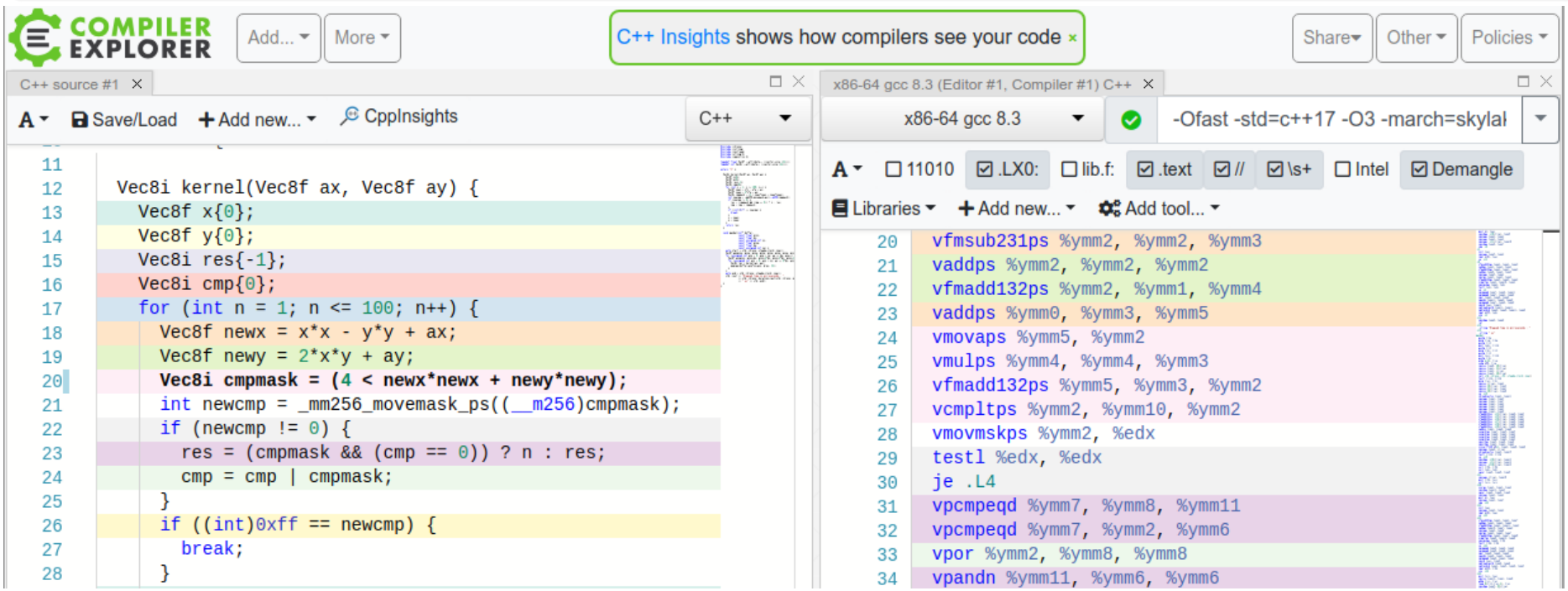
Solution

- Presence of xmm but ps only in mov
- Not vectorized

For small pieces of code : godbolt

what is it

- Online, on the fly compilation
- Annotated, colorized assembler
- Supports many platforms and compilers



Data format for vectorization

Some constraints

- Loading/storing a vector from/to non contiguous data is very inefficient
  - even worse than n data loads
- Converting data format is also expensive
  - and will ruin your vectorization gains
- So you need proper data format from scratch

Which data layout to choose ?

- Depends on your algorithm
- Also depends on your CPU !

First Vectorization Attempt

Simple, standard matrix times vector - Let's adopt a row first storage

$$\begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix} = \begin{bmatrix} T_{XX} & T_{XY} & T_{XZ} \\ T_{YX} & T_{YY} & T_{YZ} \\ T_{ZX} & T_{ZY} & T_{ZZ} \end{bmatrix} \cdot \begin{bmatrix} P_X \\ P_Y \\ P_Z \end{bmatrix} = \begin{bmatrix} T_{XX}.P_x + T_{XY}.P_Y + T_{XZ}.P_Z \\ T_{YX}.P_x + T_{YY}.P_Y + T_{YZ}.P_Z \\ T_{ZX}.P_x + T_{ZY}.P_Y + T_{ZZ}.P_Z \end{bmatrix}$$

Will actually be something like :

$$\begin{bmatrix} T_{XX} & T_{XY} & T_{XZ} \\ T_{YX} & T_{YY} & T_{YZ} \\ T_{ZX} & T_{ZY} & T_{ZZ} \end{bmatrix} \cdot \begin{bmatrix} P_X \\ P_Y \\ P_Z \end{bmatrix} = \begin{bmatrix} T_{XX}.P_x & T_{XY}.P_Y & T_{XZ}.P_Z \\ T_{yX}.P_x & T_{yY}.P_Y & T_{yZ}.P_Z \\ T_{zX}.P_x & T_{zY}.P_Y & T_{zZ}.P_Z \end{bmatrix} \Rightarrow \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}$$

3 mul $\sim$ 6 cycles

6 hadd, 3 mov $\sim$ 60 cycles

Scalar case : 9 mul, 6 adds $\sim$ 30 cycles

Second Vectorization Attempt

Let's adopt a column first storage and use Broadcast

$$\begin{bmatrix} T_{XX} & T_{XY} & T_{XZ} \\ T_{YX} & T_{YY} & T_{YZ} \\ T_{ZX} & T_{ZY} & T_{ZZ} \end{bmatrix} \cdot \begin{bmatrix} P_X \\ P_Y \\ P_Z \end{bmatrix} = \begin{bmatrix} T_{XX} \\ T_{YX} \\ T_{ZX} \end{bmatrix} \cdot \begin{bmatrix} P_X \\ P_X \\ P_X \end{bmatrix} + \begin{bmatrix} T_{XY} \\ T_{YY} \\ T_{ZY} \end{bmatrix} \cdot \begin{bmatrix} P_Y \\ P_Y \\ P_Y \end{bmatrix} + \begin{bmatrix} T_{XZ} \\ T_{YZ} \\ T_{ZZ} \end{bmatrix} \cdot \begin{bmatrix} P_Z \\ P_Z \\ P_Z \end{bmatrix}$$

Costs :

- 3 broadcasts $\sim$ 3 cycles
- 3 mul and 2 adds $\sim$ 10 cycles

Twice better than scalar case !
Wait a minute... only twice ?!?

Vertical vs Horizontal vectorization

Vertical vectorization

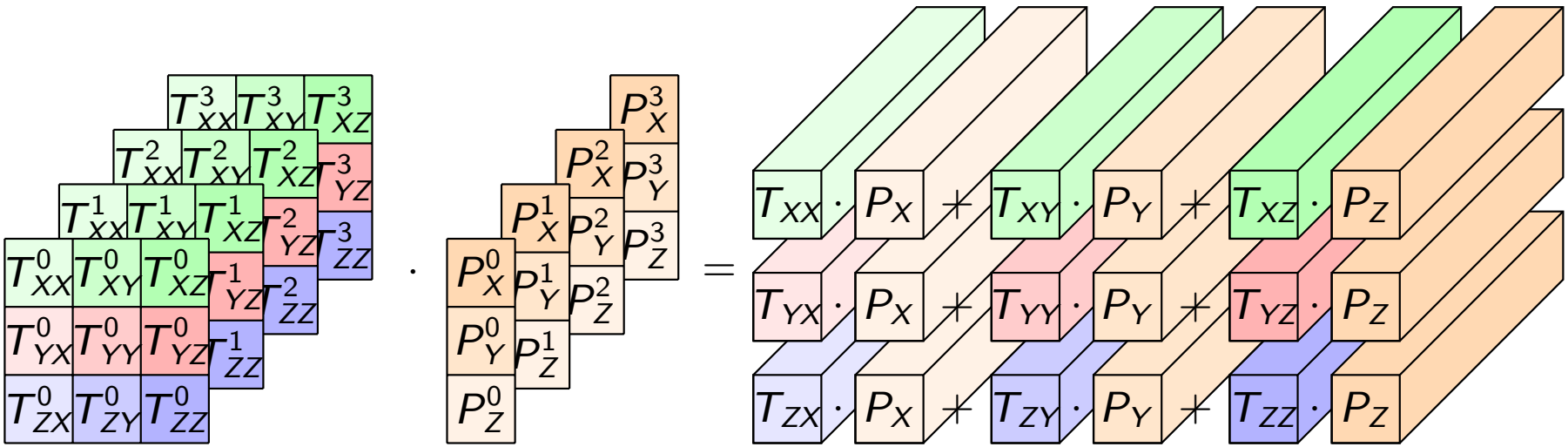
- Previous attempts are examples of vertical vectorization
- They use parallelism within the given data
- Speedup is limited by data size
  - was 3 numbers here, while our vector was 8 or 16 items long !

Horizontal vectorization

- Aims at using parallelism between independent (but similar) computations
- In our example several (many ?) products Matrix by Vector
- Allows to fully use the vector sizes

Horizontal vectorization example

Let's compute n products in one go (n = 4 on the picture)
Let's have vectors in the new dimension, one color = one vector



We compute as if we were scalar, using vectors to do n at a time
Cost :

- 9 mul + 6 add $\sim$ 30 cycles for n products
- n is typically 8 or 16 $\rightarrow$ 4 to 2 cycles per product
- Perfect speedup !

Vertical vectorization allows AoS

AoS = Array of Structures

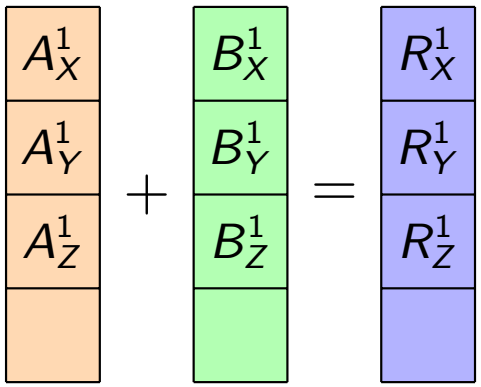
Basically you use standard structures

So you can write very natural code :

```
struct Vector { float x; float y; float z; };
using Matrix = std::array<float, 12>; // padded
std::array<Vector,N> Ps = ...;
std::array<Matrix,N> Ts = ...;
auto V0 = multiply(Ts[0], Ps[0]);
```

Drawback

- It does not scale with vector width
- It needs adaption of your math code
  - a dedicated, vectorized multiply method



Horizontal vectorization requires SoA

SoA = **S**tructures **o**f **A**rray

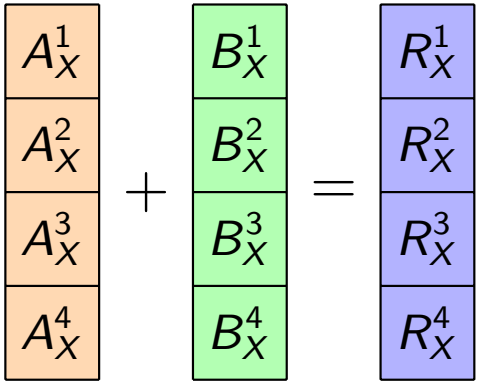
That is standard structures where each element became a vector

Thus you loose the concept of elements

```
using floats = std::array<float,N>;
struct Vectors { floats xs, ys, zs; };
using Matrices = std::array<floats, 9>;
Vectors Ps = ...;
Matrices Ts = ...;
auto Vs = multiply(Ts, Ps);
// no Ts[0] or Ps[0]
```

Advantages

- Scales perfectly with vector width
- Code similar (identical ?) to scalar one

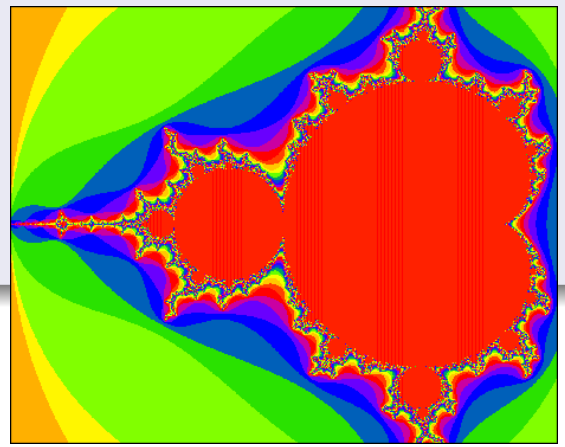


Example Code

Mandelbrot kernel

Given (ax, ay) a point in 2D space, compute n :

```
int kernel(float ax, float ay) {
  float x = 0; float y = 0;
  for (int n = 1; n <= 100; n++) {
    float newx = x*x - y*y + ax;
    float newy = 2*x*y + ay;
    if (4 < newx*newx + newy*newy) return n;
    x = newx; y = newy;
  }
  return -1;
}
```



Why is autovectorization not so easy ?

Summary of Andrzej's slides

Main issues with autovectorization :

- Aliasing, alignment, data dependencies, branching, ...
- In general lack of knowledge of the compiler

Ways to solve (some of) them

- `restrict`, align, ternary operator, ... aka give knowledge to compiler
- And proper data structures (SoA)

Still worth trying it

- It's (almost) a free lunch !
- 100% portable code
- No dependencies

How to autovectorize ?

Compiler flags

- Optimization ones
  - For gcc, clang : -O3 or -O2 -ftree-vectorize
  - For icc : -O2 or -O3. Use -no-vec to disable it
- Architecture ones
  - For avx2 : -mavx2 on gcc/clang, -axAVX2 -xAVX2 on icc
  - For avx512 on gcc/clang : -march=skylake-avx512
  - For avx512 on icc: -xCORE-AVX512
  - For optimal vectorization depending on your CPU : -march=native on gcc/clang, -xHOST on icc

How to debug autovectorization

Ask the compiler about its choices

For icc

- Use -vec-report=5 to "tell the vectorizer to report on non-vectorized loops and the reason why they were not vectorized"

For clang

- Use -Rpass-missed=loop-vectorize to "identify loops that failed to vectorize"
- Use -Rpass-analysis=loop-vectorize to "show the statements that caused the vectorization to fail"

For gcc

- Use -fopt-info-vec-missed to get "detailed info about loops not being vectorized"

Autovectorizing Mandelbrot

code

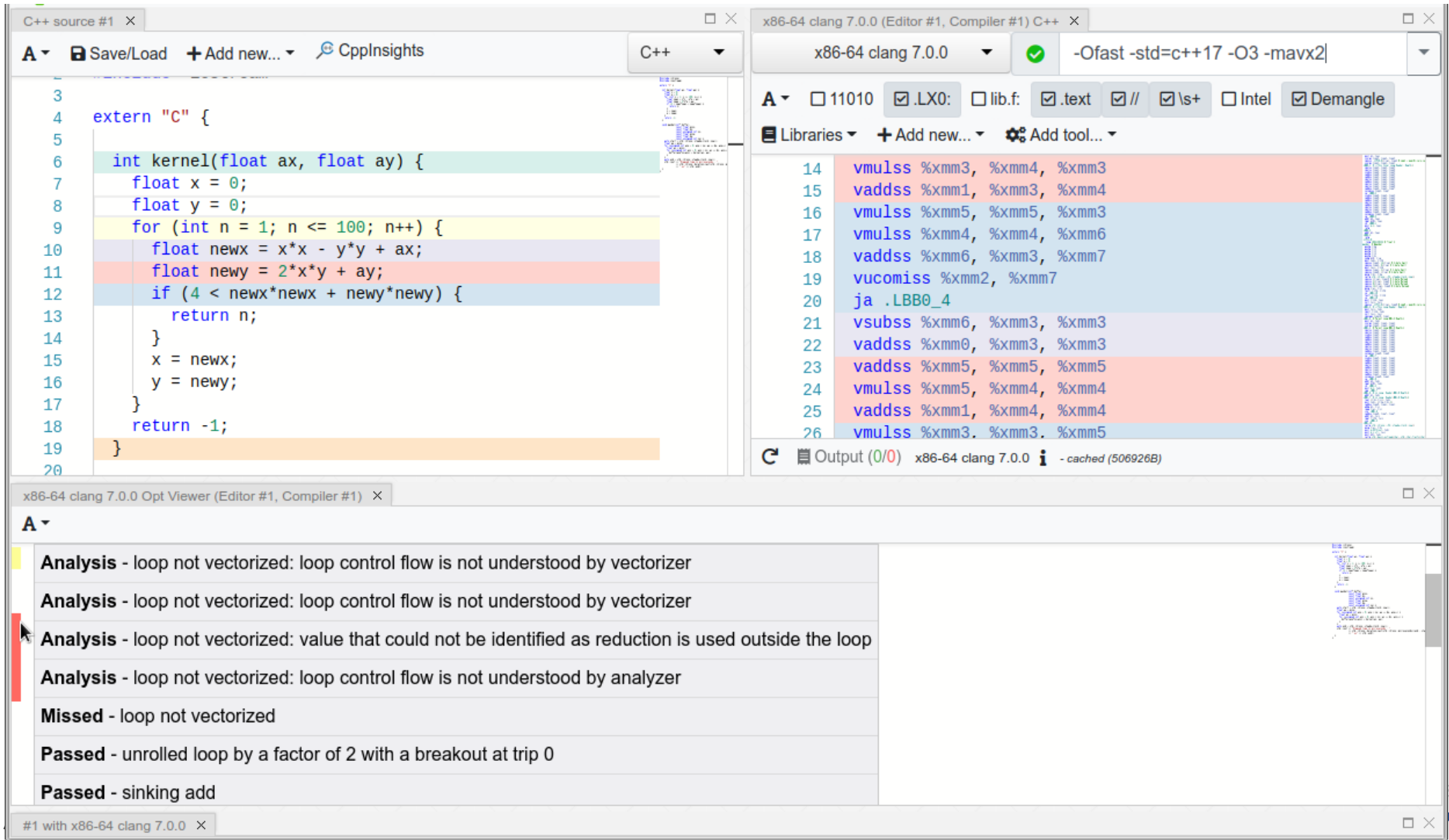
```
8  int kernel(float ax, float ay) {
9    float x = 0; float y = 0;
10   for (int n = 1; n <= 100; n++) {
11     float newx = x*x - y*y + ax;
12     float newy = 2*x*y + ay;
13     if (4 < newx*newx + newy*newy) {
```

compiler output (gcc)

```
...
mandel.cpp:10:21: note: not vectorized: control flow in loop.
mandel.cpp:10:21: note: bad loop form.
...
```

For small code, godbolt is again your friend

- Use clang, click "Add new..." in assembly pane
- Select "optimization output"
- Move mouse on the side of the interesting code



Autovectorization is still not enough

It's not fully mature

- Still very touchy despite improvements
- Only able to vectorize loops (or almost)
- Hardly able to handle branching via masks
- No abstract knowledge of the application (yet ?)

It will probably never be good enough

- As it cannot know as much as the developer
- Especially concerning input data such as
  - average number of tracks reconstructed
  - average energy in that data sample
- And the optimal code depends on this

So we need to vectorize by hand from time to time

Why inline assembly should not be used

- Hard to write/read
- Not portable at all
  - processor specific AND compiler specific
- Almost completely superseeded by intrinsics

So just don't do it

The intrinsics layer

Principles

- Intrinsics are functions that the compiler replaces with the proper assembly instructions
- It hides nasty assembly code but maps 1 to 1 to SIMD assembly instructions

Pros

- Easy to use
- Full power of SIMD can be achieved

Pros

- Very verbose, very low level
- Processor specific

Intrinsics crash course

naming convention :

```
_mm<S><mask>_<op>_<suffix>(data_type param1, ...)
```

where

- <S> is empty for SSE, 256 for AVX2 and 512 for AVX512
- <mask> is empty or _mask or _maskz (AVX512 only)
- <op> is the operator (add, mul, ...)
- <suffix> describes the data in the vector

Example :
`_mm256_mul_ps`, `_mm512_maskz_add_pd`

see `https://software.intel.com/sites/landingpage/IntrinsicsGuide`

intel Intrinsics Guide

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code. ×

Technologies

- [ ] MMX
- [ ] SSE
- [ ] SSE2
- [ ] SSE3
- [ ] SSSE3
- [ ] SSE4.1
- [ ] SSE4.2
- [ ] AVX
- [ ] AVX2
- [x] FMA
- [ ] AVX-512
- [ ] KNC
- [ ] SVML
- [ ] Other

Categories

- [ ] Application-Targeted
- [ ] Arithmetic
- [ ] Bit Manipulation
- [ ] Cast
- [ ] Compare
- [ ] Convert
- [ ] Cryptography
- [ ] Elementary Math

Functions

- [ ] General Support
- [ ] Load
- [ ] Logical
- [ ] Mask
- [ ] Miscellaneous
- [ ] Move

_mm_search ?

__m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c) vfmadd132pd, vfmadd213pd, vfmadd231pd

__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c) vfmadd132pd, vfmadd213pd, vfmadd231pd

__m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c) vfmadd132ps, vfmadd213ps, vfmadd231ps

__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c) vfmadd132ps, vfmadd213ps, vfmadd231ps

Synopsis

```
__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c)
#include <immintrin.h>
Instruction: vfmadd132ps ymm, ymm, ymm
             vfmadd213ps ymm, ymm, ymm
             vfmadd231ps ymm, ymm, ymm
CPUID Flags: FMA
```

Description

Multiply packed single-precision (32-bit) floating-point elements in a and b, add the intermediate result to packed elements in c, 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]) + c[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Knights Landing	6	0.5
Skylake	4	0.5
Broadwell	5	0.5
Haswell	5	0.5

Practically for Mandelbrot

Code

```
__m256i kernel(__m256 ax, __m256 ay) {
  __m256 x = _mm256_setzero_ps();
  __m256 y = _mm256_setzero_ps();
  for (int n = 1; n <= 100; n++) {
    __m256 newx = _mm256_add_ps
    (_mm256_sub_ps(_mm256_mul_ps(x,x), _mm256_mul_ps(y,y)), ax);
    __m256 newy = _mm256_add_ps
    (_mm256_mul_ps(two,_mm256_mul_ps(x,y)), ay);
    __m256 norm = _mm256_add_ps(_mm256_mul_ps(newx, newx),
    _mm256_mul_ps(newy, newy));
    __m256 cmpmask = _mm256_cmp_ps(four, norm, _CMP_LT_OS);
```

- A bit too verbose to my taste !
- Hard to understand what's going on

Vector compiler extensions

Principle

- Compiler extended syntax to write SIMD code
- Compiler specific, mostly (clang and gcc are close)
- Allows to use vector types naturally

Pros

- Easy to use
- (Almost) independent of processor

Cons

- Limited instruction set
- Compiler specific

Practically for Mandelbrot

Code

```
typedef float Vec8f __attribute__ ((vector_size (32)));
typedef int Vec8i __attribute__ ((vector_size (32)));
Vec8i kernel(Vec8f ax, Vec8f ay) {
  Vec8f x{0};
  Vec8f y{0};
  for (int n = 1; n <= 100; n++) {
    Vec8f newx = x*x - y*y + ax;
    Vec8f newy = 2*x*y + ay;
    Vec8i cmpmask = (4 < newx*newx + newy*newy);
```

- Syntax very close to scalar case
- Only change : the comparison is returning a mask rather than a boolean

The library way

Expectations

- Write compiler agnostic code
- With natural syntax, a la compiler extensions
- Evolve with technologies without modifying the code

Many available libraries

VC, **xSIMD**, **VCL**, **UME::SIMD**, **VecCore**, ...

- **VC** has made a proposal to C^{++}standard comitee
  - we may have vector support in the standard one day
- **VCL** is header only library, so easy to use
- **VecCore** is an attempt to rule them all
  - basically a common wrapper on top on the rest

Not all of them support AVX 512

VCL - Practically for Mandelbrot

Code

```
#include <vectorclass.h>
Vec8i kernel(Vec8f ax, Vec8f ay) {
  Vec8f x{0};
  Vec8f y{0};
  for (int n = 1; n <= 100; n++) {
    Vec8f newx = x*x - y*y + ax;
    Vec8f newy = 2*x*y + ay;
    Vec8fb newcmp = (4 < newx*newx + newy*newy);
```

- Code very close to vector extensions' one, but compiler agnostic
- Still using mask obviously

VC - Practically for Mandelbrot

Code

```
#include <Vc/vector.h>
Vc::int_v kernel(Vc::float_v ax, Vc::float_v ay) {
  Vc::float_v x(0);
  Vc::float_v y(0);
  for (int n = 1; n <= 100; n++) {
    Vc::float_v newx = x*x - y*y + ax;
    Vc::float_v newy = 2*x*y + ay;
    auto newcmp = (4 < newx*newx + newy*newy);
```

- Note that the code is vector width agnostic this time !

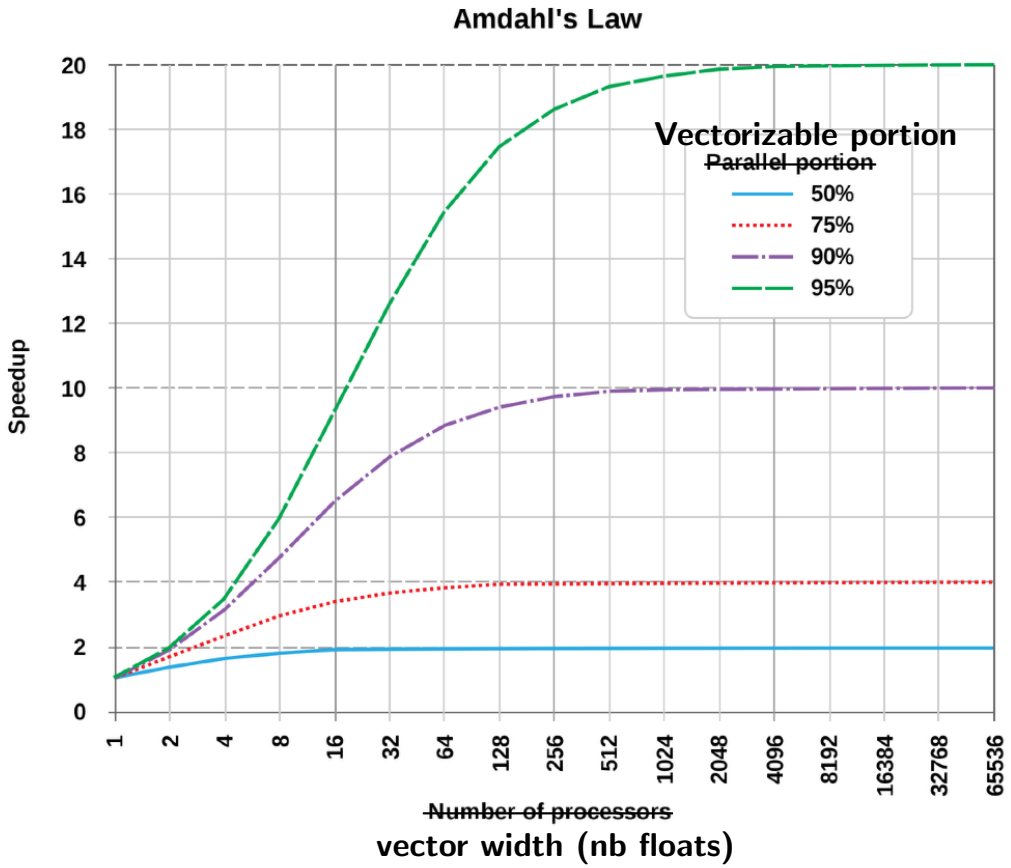
Amdahl strikes back

Remember the talk of Danilo ? I revisited it slightly

P : vectorizable portion
N : vector width (floats)
Δt_0 : scalar execution time

$$\Delta t = \Delta t_0.[(1-P) + \frac{P}{N}]$$

$$Speedup = \frac{\Delta t_0}{\Delta t} = \frac{1}{(1-P)+\frac{P}{N}}$$



"... the effort expended on achieving high speed through vectorization is wasted unless it is accompanied by achievements in scalar processing rates of very nearly the same magnitude." - me, 2019

When does vectorization bring speed ?

Vectorization will bring speed if

- The code in computing intensive
- You have enough parallelism (think horizontal vectorization)
- The code has few branches
- The data have proper format (SoA)

Vectorization won't necessarily work

- If you do not have SoA and conversion is too costly
  - you lose back what you won (or more ?)
- For specific algorithms
  - typically standard sorting algorithm (std::sort)
  - for that case SoA is even to be avoided

A matter of testing and experience. You'll be surprised for sure

A word on Vectorization and IO

The Problem

- When you optimize one piece, you put more pressure on the others
- Speeding up CPU may lead to memory bandwidth issues

Typical scenario

- Suppose you get x16 speedup on your matrix - vector code
- So you'll use 16x more input data than you used to
- Do you have the memory bandwidth for that ?

Roofline Model

Definition

Let's define for a given piece of code (aka kernel) :

W work, number of operations performed

Q memory traffic, number of bytes of memory transfers

$I = \frac{W}{Q}$ the arithmetic intensity

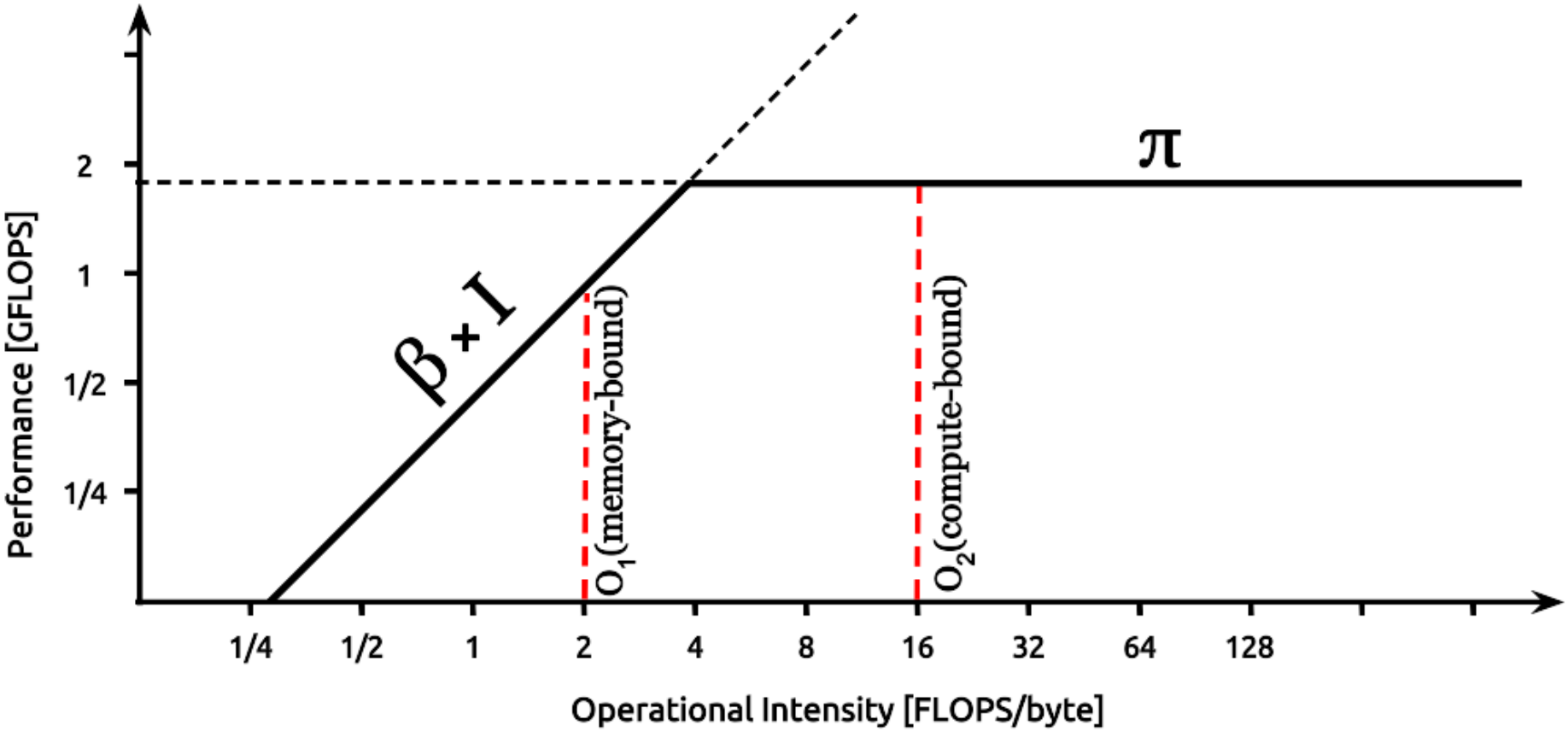
And let's define for a given hardware :

β the peak bandwidth in bytes/s, usually obtained via benchmarks

π the peak performance in flops/s, derived from the architecture

All this is plotted in a log-log graph of flops/s versus arithmetic intensity

Roofline plot

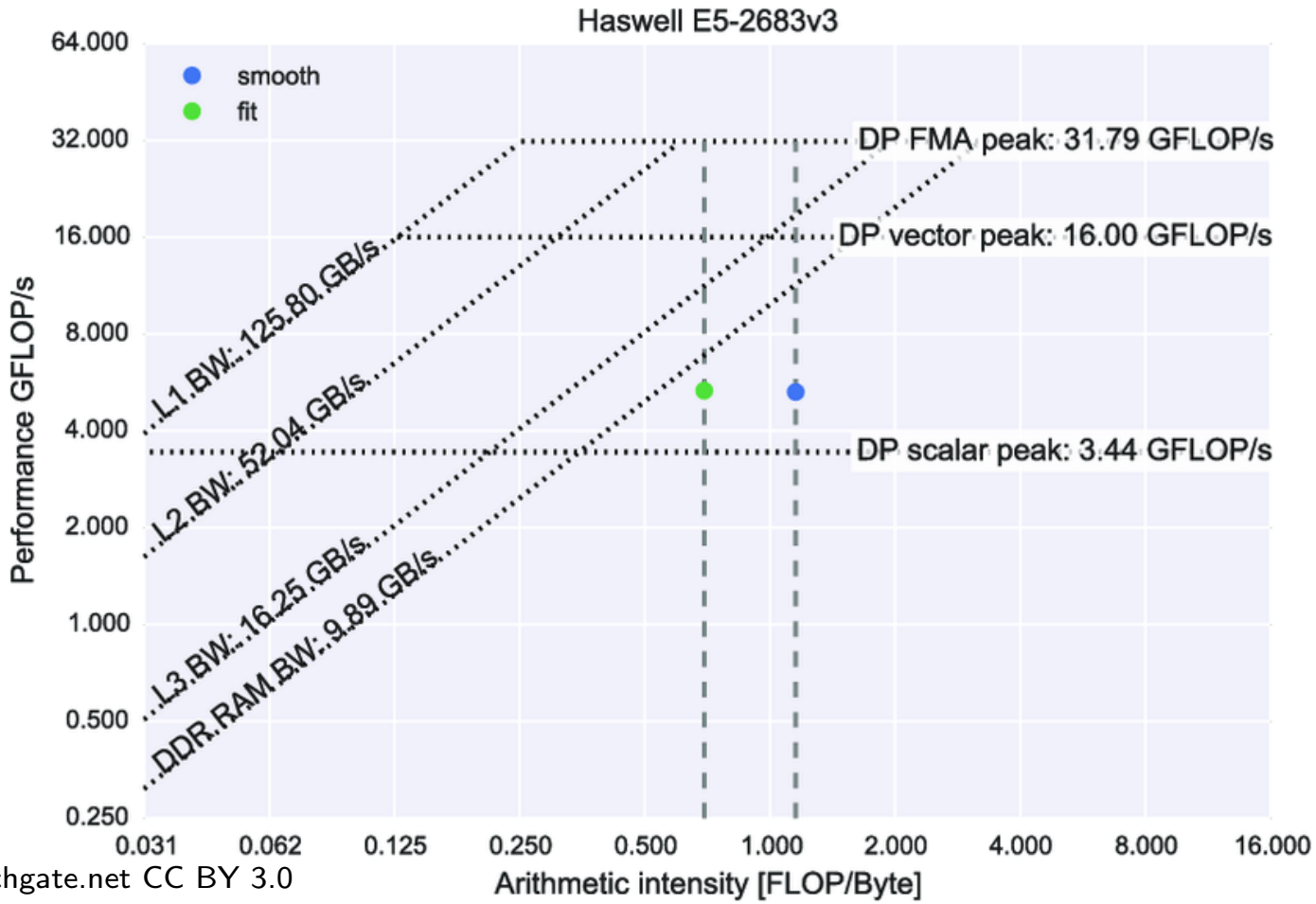


- Shows what you can expect for a given arithmetic intensity
- And whether you are ultimately CPU or I/O bound

Source : Giu.natale CC BY-SA 4.0, via Wikimedia Commons

Realistic Roofline plot

- Multiple lines for different levels of caches
- Multiple lines for vectorization and FMA



Source : researchgate.net CC BY 3.0

Conclusion

Key messages of the day

- Vectorization requires a suitable data model, typically SoA
  - And always prefer horizontal vectorization when you can
- There are several ways to vectorize
  - Check whether autovectorization works for you
  - Otherwise choose between intrinsics, compiler extensions and libraries
- Do not build wrong expectations on the overall speedup
  - Amdahl's law is really stubborn
  - And you may hit other limitations, like I/O