Some elements on Vectorization

SCIENTIFIC & DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F.

Advanced HPC 2024-2025 @ Università di Trieste

Some elements on Vectorization

SCIENTIFIC & DATA-INTENSIVE COMPUTING

Università di Trieste
Advanced HPC 2024-2025

Luca Tornatore
I.N.A.F.

Outline

- Introduction
- Checking the flags
- Vectorization by compiler's wow effect
- Vectortization by vector types
- Vectorization by intrinsics
- Vectorization by OpenMP SIMD

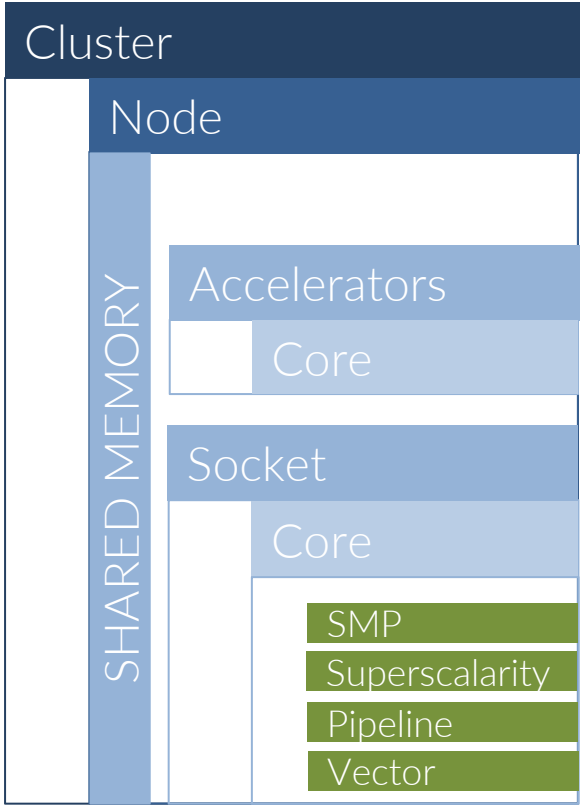
Main sources

- Intel®'s Intrinsics Guide
- Cornell's Virtual Workshop on Vectorization *(cited as VWV)*
- SIMD at Algorithmica.org
- High Performance Parallelism Pearls, Volume 1 & 2
- Materials and examples uploaded in the git

[0]

Introduction

Recap: the levels of parallelism



Group of nodes connected via a fast network

Set of CPUs and Accelerators connected via fast lanes
Memory is shared among CPUs; may be shared also with accelerators on new emerging architectures

Special devices specialized for high throughput
massive parallelism of many simple cores

Set of cores that share levels of cache memory

Group of funtional units that communicate / operate via registers and L1 cache

Group of thread contexts that share functional units

Group of instructions sharing functional units thanks to out-of-order capability

Multiple instructions sharing functional units

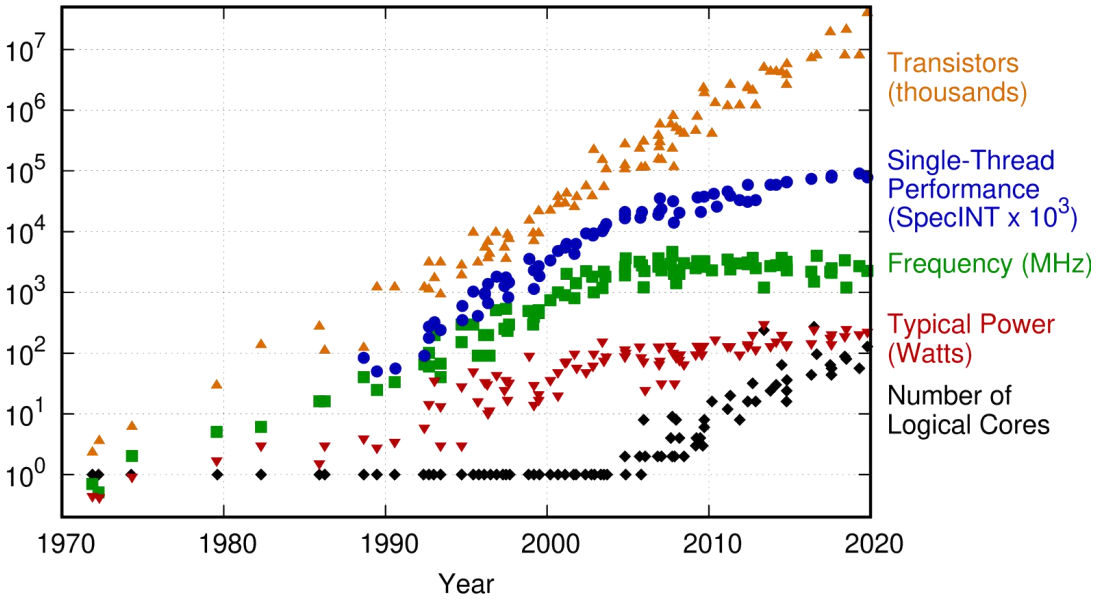
Single instructions on Multiple Data

Recap: the levels of parallelism

Cluster	Group of nodes connected via a fast network
Node	Set of CPUs and Accelerators connected via fast lanes Memory is shared among CPUs; may be shared also with accelerators on new emerging architectures
SHARED MEMORY	
Accelerators	Special devices specialized for high throughput massive parallelism of many simple cores
Core	
Socket	Set of cores that share levels of cache memory
Core	Group of funtional units that communicate / operate via registers and L1 cache
SMP	Group of thread contexts that share functional units
Superscalarity	Group of instructions sharing functional units thanks to out-of-order capability
Pipeline	Multiple instructions sharing functional units
Vector	Single instructions on Multiple Data

Recap: why vectorization ?

Dennard et al. (1974) showed that if MOSFET transistors in a semiconductor device shrink by a factor k in each dimension, then the voltage V and current I required by each circuit element can also be made to decrease by k.
This reduces each element's power consumption (VI) by **k²**. But the number of these elements per unit area increases by **k²**, so the power dissipated per unit area stays the same.
This means the power and cooling constraints of the overall device remain unaltered. Yet these scaled-down elements can operate at higher frequency, because the delay time TRC per circuit element is reduced by k. (Basic physics: per element, the resistance R=V/I is unchanged, while the capacitance C depends on area/distance and shrinks by k.) Unfortunately, starting around 2005, this favorable Dennard scaling began to run into trouble due to leakage currents that develop at super-small dimensions. As a result, even though transistors are still shrinking in size, the frequency cannot be made to go higher.



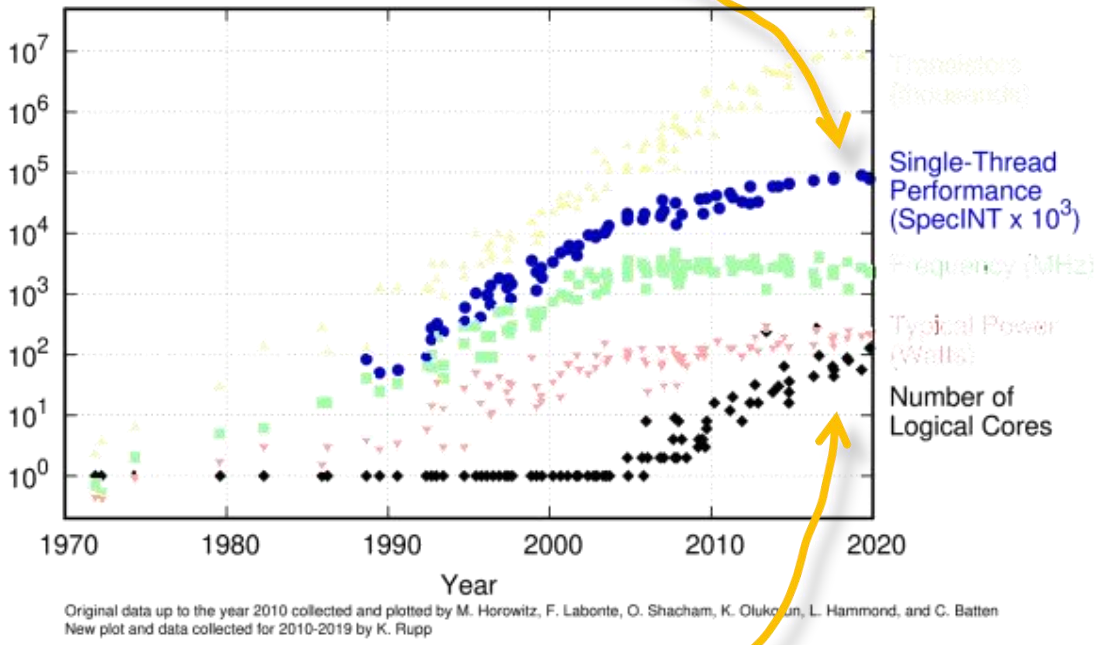
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Picture and text quoted from the Cornell Virtual Workshop

Recap: why vectorization ?

Dennard et al. (1974) showed that if MOSFET transistors in a semiconductor device shrink by a factor k in each dimension, then the voltage V and current I required by each circuit element can also be made to decrease by k.
This reduces each element's power consumption (VI) by **k²**. But the number of these elements per unit area increases by **k²**, so the power dissipated per unit area stays the same.
This means the power and cooling constraints of the overall device remain unaltered. Yet these scaled-down elements can operate at higher frequency, because the delay time TRC per circuit element is reduced by k. (Basic physics: per element, the resistance R=V/I is unchanged, while the capacitance C depends on area/distance and shrinks by k.) Unfortunately, starting around 2005, this favorable Dennard scaling began to run into trouble due to leakage currents that develop at super-small dimensions. As a result, even though transistors are still shrinking in size, the frequency cannot be made to go higher.

performance increase partially due to vectorization



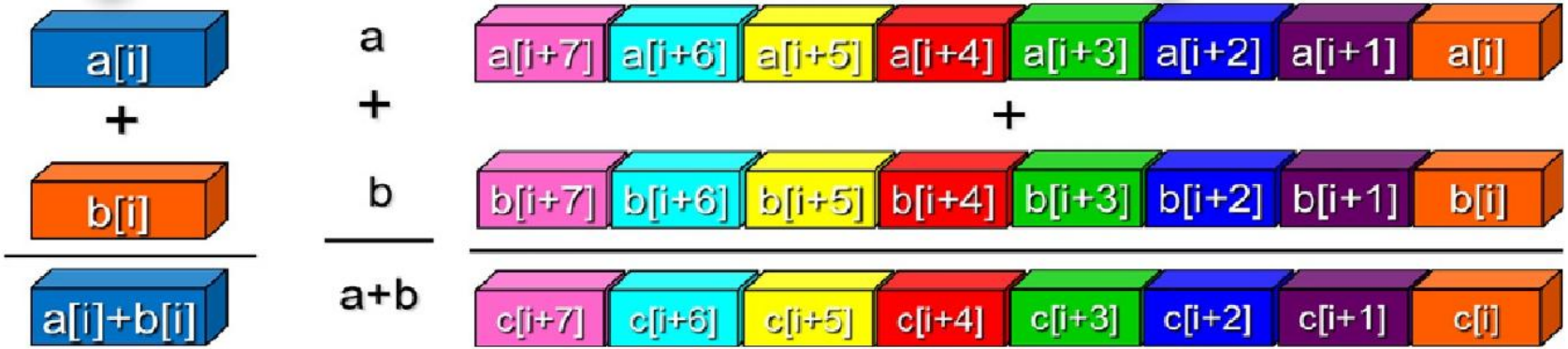
More cores to increase parallelism since the frequency has hit the power wall

Vector instructions

What is the nature of vector operations ?

- **Scalar mode**
  - One instruction produces one result (SISD)
- **SIMD processing**
  - One instruction can produce multiple results (SIMD)
    - using AVX VADDPS instruction

```
for (i=0; i<=MAX; i++)
    c[i] = a[i] + b[i];
```

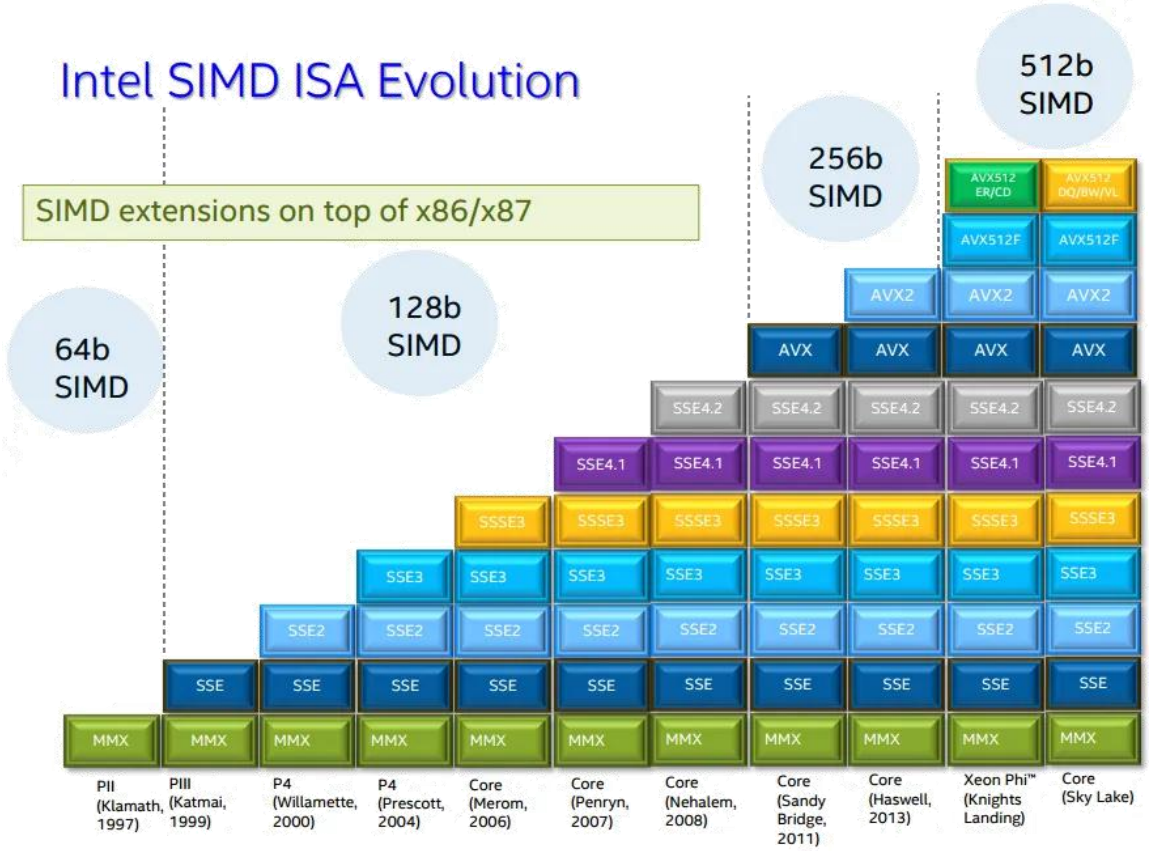


taken from "High Performance Parallelism Pearls, vol 2" , ch. 22

Vector instructions, evolution

Example for Intel® ISA.
AMD® (3dNow®),
ARM® (SVE®, Neon®),
IBM® (AltiVec®),
RISC-V (V), etc,
have their own.

Different ISA share several similar features while having some peculiar ones.



taken from the net, several instances of this figure exist without credits

Vector instructions, evolution

1999+ **SSE (128bit)**

128 bit vector extension. Four 32bit units can be processed in parallel

2006+ **SSE4 (128bit)**

Extension to SSE, SSE2, SSE3, contains additional instructions for vector processing

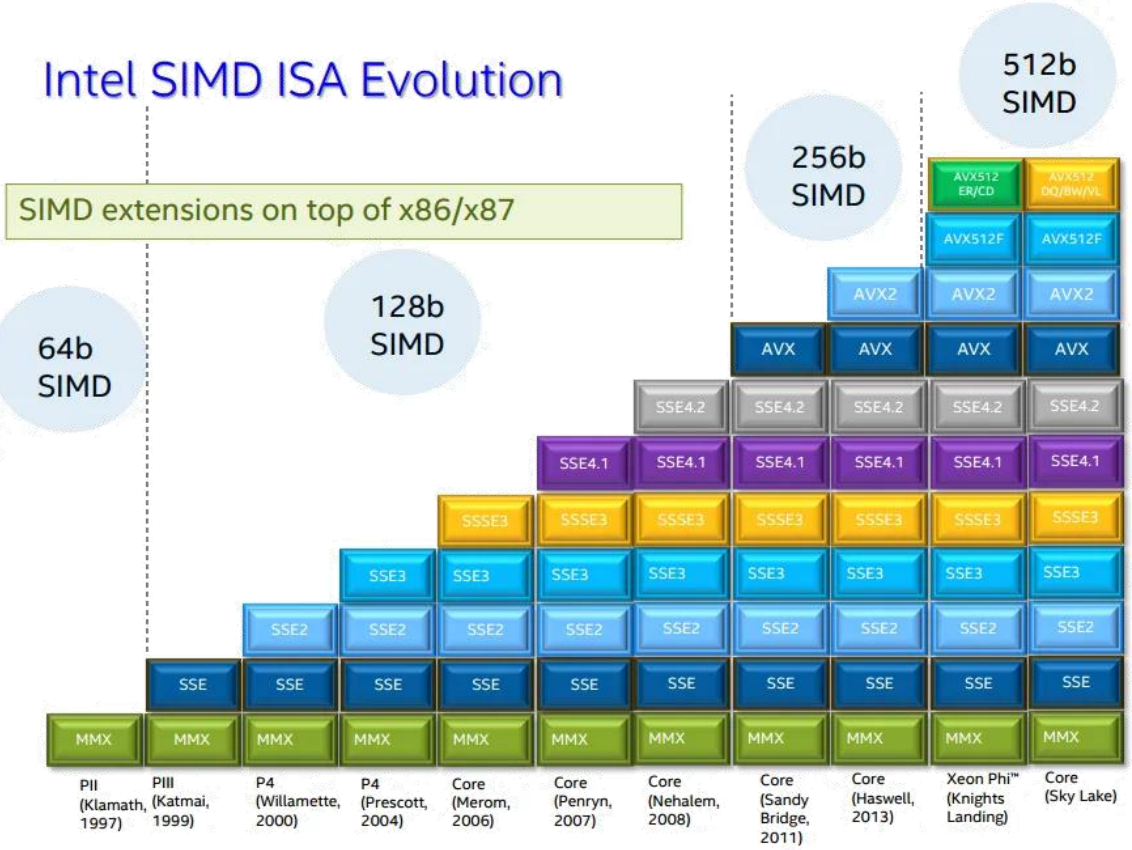
2013+ **AVX, AVX2 (256bit)**

256 bit vector extensions, registers would be twice as large as SSE and have twice the throughput. Eight 32bit units can be processed in parallel

16 - 2018
2022+ **AVX-512 (512bit)**

Launched in Intel Knight's landing CPU in 2016, discontinued in 2018. AMD launched AVX512 support in Zen 4 CPUs from 2022

Intel Xeon CPU in this timeline may support AVX-512



taken from the net, several instances of this figure exist without credits

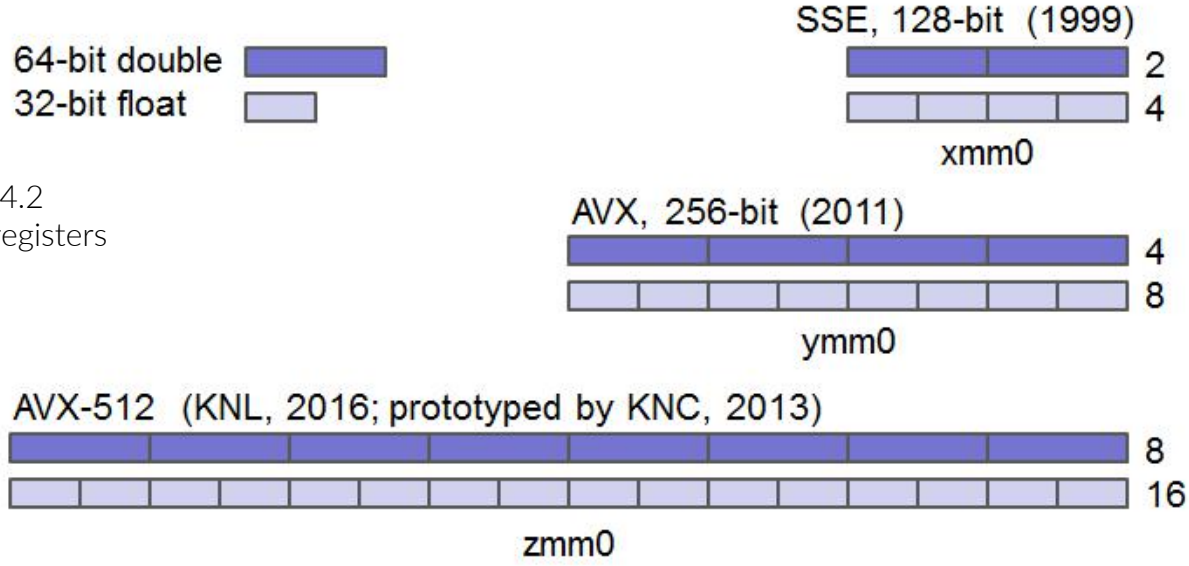
Vector instructions, registers size

511 … 256	255 … 128	127 … 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

From the Wikipedia page on AVX-512

X86 AVX / AVX2
YMM[0-15] registers

X86 SSE-SSE4.2
XMM[0-15] registers



From the Cornell Virtual Workshop on Vectorization https://cvw.cac.cornell.edu/vector

Strategies for vectorization

How is it possible to achieve the vectorization ?

More Convenience

- *Higher Level*
- *More abstraction*
- *No HW dependence*

- *Less abstraction*
- *HW specific*
- *Lower level*

More Control

OpenMP Vectorization
Compiler Auto-Vectorization

C++ classes

Compiler-dependent vector types

Intrinsics

Assembler

Strategies for vectorization

What we'll see

More Convenience

- *Higher Level*
- *More abstraction*
- *No HW dependence*

- *Less abstraction*
- *HW specific*
- *Lower level*

More Control

✓ **OpenMP Vectorization**

✓ **Compiler Auto-Vectorization**

C++ classes

✓ **Compiler-dependent vector types**

✓ **Intrinsics**

Assembler

[1]

How to check for SIMD capablities

Checking for the right flags

1) statically get the value for your cpu, for instance by
   `grep` the output of either **`lscpu`** or **`/proc/cpuinfo`**

```
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 pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
         cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr
         pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
         3dnowprefetch cpuid_fault epb ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad
         fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb intel_pt sha_ni xsaveopt
         xsavec xgetbv1 xsaves split_lock_detect user_shstk avx_vnni dtherm ida arat pln pts hwp hwp_not ify
         hwp_act_window hwp_epp hwp_pkg_req hfi vnmi umip pku ospke waitpkg gfni vaes vpclmulqdq rdpid movdiri movdir64b
         fsrm md_clear serialize arch_lbr ibt flush_l1d arch_capabilities
```

2) discover from the source code at both compile-time and run-time

Checking for the right flags

How can we check what capabilities has the cpu on which the code runs?

include the relevant header that defines low-level routines

get the cpu data via the cpuid instruction

parse the cpu's flags looking for precise tags

```
#include <cpuid.h>

int main ( void )
{
  ...
  __builtin_cpu_init();

  if ( __builtin_cpu_supports("avx512f") )
    printf("CPU supports AVX512f\n");
  if ( __builtin_cpu_supports("avx2") )
    printf("CPU supports AVX2\n");
  if ( __builtin_cpu_supports("avx") )
    printf("CPU supports AVX\n");
  if ( __builtin_cpu_supports("sse4.2") )
    printf("CPU supports SSE4.2\n");
  if ( __builtin_cpu_supports("sse4.1") )
    printf("CPU supports SSE4.1\n");
  if ( __builtin_cpu_supports("sse3") )
    printf("CPU supports SSE3\n");
```

Checking for the right flags

How can we check what capabilities has the cpu on which the code runs?

It is possible to parse the compile-time macros defined by the compiler to check what flags are active on the current cpu.

... however, let's try to run the code `support.c`

to understand that the compiler needs some directives too.

```
#include <immintrin.h>

#ifdef __AVX512__

#define VD_SIZE ( sizeof( __m512d ) / sizeof(double) )

#elif defined ( __AVX__ ) || defined ( __AVX2__ )

#define VD_SIZE ( sizeof( __m256d ) / sizeof(double) )

#elif defined ( __SSE4__ ) || defined ( __SSE3__ )

#define VD_SIZE ( sizeof( __m128d ) / sizeof(double) )

#else

#define VD_SIZE 1

#endif
```

Checking for the right flags

How can we check what capabilities has the cpu on which the code runs?

It is possible to parse the compile-time macros defined by the compiler to check what flags are active on the current cpu.

... however, let's try to run the code `support.c`

to understand that the compiler needs some directives too.

note: including **`immintrin.h`** is needed to have all the basic intrinsics (for instance, the types __mm512d, __mm256d, ..)

```
#include <immintrin.h>

#ifdef __AVX512__

#define VD_SIZE ( sizeof( __m512d ) / sizeof(double) )

#elif defined ( __AVX__ ) || defined ( __AVX2__ )

#define VD_SIZE ( sizeof( __m256d ) / sizeof(double) )

#elif defined ( __SSE4__ ) || defined ( __SSE3__ )

#define VD_SIZE ( sizeof( __m128d ) / sizeof(double) )

#else

#define VD_SIZE 1

#endif
```

Checking for the right flags

How can we check what capabilities has the cpu on which the code runs?

```
:> $ gcc -o support support.c
:> $ ./support

CPU supports AVX / AVX2
CPU supports SSE4.2
CPU supports SSE4.1
CPU supports SSE3

The double vector size is : 1
```

Why the compiler is able to correctly recognize the CPU's capabilities but actually gets a wrong vector size ?

Also compiling with **-O3** does not make it to behave appropriately

Exercise: compile & run support.c

Checking for the right flags

How can we check what capabilities has the cpu on which the code runs?

```
:> $ gcc -march=native -o support support.c
:> $ ./support

CPU supports AVX / AVX2
CPU supports SSE4.2
CPU supports SSE4.1
CPU supports SSE3

The double vector size is : 4
```

We need to instruct the compiler to set the current architecture as a target, instead of a generic **x86_64**

Checking for the right flags

How can we check what capabilities has the cpu on which the code runs?

```
:> $ gcc -msse4.2 -o support support.c
:> $ ./support

CPU supports AVX / AVX2
CPU supports SSE4.2
CPU supports SSE4.1
CPU supports SSE3

The double vector size is : 2
```

We may ask to support a specific SIMD extension (Intel's and AMD's)

```
-msse
-msse2
...
-msse4.2
-mavx
-mavx2
-mavx512f
```

ARM, POWER, and others have their own specific targets

Exercise: compile with icx and clang

[1] Check capabilities

Checking for the right flags

How can we check what capabilities has the cpu on which the code runs?

Add the directive

```
#pragma GCC target("avx2")
```

at the top of the code
However this is a technique that is strongly compiler-dependent

```
#pragma GCC target("avx2")

#include <immintrin.h>
#include <cpuid.h>

#ifdef __AVX512__

#define VD_SIZE ( sizeof( __m512d ) / sizeof(double) )

#elif defined ( __AVX__ ) || defined ( __AVX2__ )

#define VD_SIZE ( sizeof( __m256d ) / sizeof(double) )

...
```

Exercise: compile with and run with gcc

[2]

Vector types (via intrinsics)

The vector intrinsics types

Once we include the <immintrin.h> header, we get access to the intrinsincs routines and types

INTEGER types

	types name	int 8bits	int 16bits	int 32bits	int 64bits
64bits	__m64	8x	4x	2x	1x
128bits	__m128i	16x	8x	4x	2x
256bits	__m256i	32x	16x	8x	4x
512bits	__m512i	64x	32x	16x	8x

The vector intrinsics types

Once we include the <immintrin.h> header, we get access to the intrinsincs routines and types

FP types

	types name	single precision	double precision
128bits	__m128	4x	-
	__m128d	-	2x
256bits	__m256	8x	-
	__m256d	-	4x
512bits	__m512	16x	-
	__m512d	-	8x

Defining vector types via intrinsics

If we want to make explicit use of vector types, and to write a code that adapts to the machine it compiles on, we need to define our own types:

the types

- `dvector_t`
- `fvector_t`
- `ivector_t`

will exist in our code with the correct sizes. As well, we may define our own wrappers for intrinsics operations

see `Vectorization/support.declare_vector_types.c`

```
// ...............................................
#ifdef __AVX512__
typedef __m512d dvector_t;
typedef __m512  fvector_t;
typedef __m512i ivector_t;
// ...............................................
#elif defined ( __AVX__ ) || defined ( __AVX2__ )
typedef __m256d dvector_t;
typedef __m256  fvector_t;
typedef __m256i ivector_t;
// ...............................................
#elif defined ( __SSE4__ ) || defined ( __SSE3__ )
typedef __m128d dvector_t;
typedef __m128  fvector_t;
typedef __m128i ivector_t;
// ...............................................
#else
typedef double dvector_t;
typedef double fvector_t;
typedef double ivector_t;
#endif
// ...............................................

#define DV_ELEMENT_SIZE (sizeof( dvector_t ) / sizeof(double) )
#define DV_BIT_SIZE (sizeof( dvector_t ) * 8 )

#define FV_ELEMENT_SIZE (sizeof( fvector_t ) / sizeof(float) )
#define FV_BIT_SIZE (sizeof( fvector_t ) * 8 )

#define IV_ELEMENT_SIZE (sizeof( ivector_t ) / sizeof(int) )
#define IV_BIT_SIZE (sizeof( ivector_t ) * 8 )
```

[3]

Auto-vectorization

Auto vectorizing a simple loop

Auto-vectorization capabilities of the compilers is a very valuable tools. However, some general concepts must be understood and analysed because several factors may hinder the vectoriation

- loop-carried data depencies
- control dependencies and branches
- unaligned memory
- not suited loop structure
- strided memory access
- function calls
- non-vectorizable math functions
- not supported data types
- …

Auto vectorizing a simple loop

Let's start with a very simple loop.

Here we perform a simple reduction of an array, with datatype **`dtype`** that is determined at compile time (either **`int`** or **`float`**).

```
dtype sum_loop ( dtype *array, uint N )
{
  dtype sum = 0;
  for ( uint32_t i = 0; i < N; i++ )
     sum += array[i];
  return sum;
}
```

Let's compile the code `sum_loop.c` for both integere and float data, asking for the vectorization report and generating the assembler:

```
gcc -S -fverbose-asm -masm=intel
-O3 -march=native -mtune=native -ftree-vectorize
-fopt-info-vec-optimized -fopt-info-vec-missed
-DDTYPE=[INTEGER|FPSP] -o sum_loop.[int|float].s sum_loop.c -masm=intel
```

- options to get the generated assembler in intel syntax
- options for optimization and vectorization
- options to get an optimization report on vectorization

Auto vectorizing a simple loop

Let's inspect what the compiler issues for **`int`** type

```
.L4:
# sum_loop.c:51:     sum += array[i];
        vpaddd  ymm0, ymm0, YMMWORD PTR [rax]   # vect_sum_11.16, vect_sum_11.16, MEM <vector(8) unsigned int> [(uint32_t *)_69]
        add     rax, 32                         # ivtmp.22,
        cmp     rdx, rax                        # _53, ivtmp.22
        jne     .L4     #,
        vmovdqa xmm1, xmm0                      # tmp142, vect_sum_11.16
        vextracti128    xmm0, ymm0, 0x1         # tmp143, vect_sum_11.16
        mov     edx, ecx                        # niters_vector_mult_vf.10, N
        vpaddd  xmm0, xmm1, xmm0                # _41, tmp142, tmp143
        and     edx, -8                         # niters_vector_mult_vf.10,
        test    cl, 7                           # N,
        vpsrldq xmm1, xmm0, 8                   # tmp145, _41,
        vpaddd  xmm0, xmm0, xmm1                # _43, _41, tmp145
        vpsrldq xmm1, xmm0, 4                   # tmp147, _43,
        vpaddd  xmm0, xmm0, xmm1                # tmp148, _43, tmp147
        vmovd   eax, xmm0                       # <retval>, tmp148
        je      .L15                            #,
        vzeroupper
```

the actual loop

```
vpaddd  ymm0, ymm0, YMMWORD PTR [rax]
add     rax, 32
cmp     rdx, rax
jne     .L4
```

Auto vectorizing a simple loop

Let's inspect what the compiler issues for **`int`** type

`vpadd target, opA, opB` → `target = opA+opB`

where

- `target, opA` and `opB` are `YMMWORD`, i.e. 512bits-wide
- are considered as vectors of `32bits integers`

```
vpaddd   ymm0, ymm0, YMMWORD PTR [rax]
add      rax, 32
cmp      rdx, rax
jne      .L4
```

Addressing mode:
" take 512bits from the address held in `reg rax` "
i.e. the sq brackets [ ] mean "consider what is inside as an address, i.e. a pointer

`rax` works also as a loop counter; compare if it is equal to the final address `rdx`; if not, jump to the begin of the loop body

increment `rax` by `32`, i.e. the address it points to by 32bytes (=256bits)

Auto vectorizing a simple loop

Let's inspect what the compiler issues for **`int`** type

```
.L4:
# sum_loop.c:51:     sum += array[i];
        vpaddd  ymm0, ymm0, YMMWORD PTR [rax]   # vect_sum_11.16, vect_sum_11.16, MEM <vector(8) unsigned int> [(uint32_t *)_69]
        add     rax, 32                         # ivtmp.22,
        cmp     rdx, rax                        # _53, ivtmp.22
        jne     .L4     #,
        vmovdqa xmm1, xmm0                      # tmp142, vect_sum_11.16
        vextracti128    xmm0, ymm0, 0x1         # tmp143, vect_sum_11.16
        mov     edx, ecx                        # niters_vector_mult_vf.10, N
        vpaddd  xmm0, xmm1, xmm0                # _41, tmp142, tmp143
        and     edx, -8                         # niters_vector_mult_vf.10,
        test    cl, 7                           # N,
        vpsrldq xmm1, xmm0, 8                   # tmp145, _41,
        vpaddd  xmm0, xmm0, xmm1                # _43, _41, tmp145
        vpsrldq xmm1, xmm0, 4                   # tmp147, _43,
        vpaddd  xmm0, xmm0, xmm1                # tmp148, _43, tmp147
        vmovd   eax, xmm0                       # <retval>, tmp148
        je      .L15                            #,
        vzeroupper
```

This section get the final single value into **`eax`**.
In fact, at the end of the loop **`ymm0`** contains 8 partial results.
The 7 summations are done via 3 **`vpaddd`** on reshuffled registers

if the iteration space was not exactly divisible by 8, we need to account for the remainder iterations;
otherwise, we jump to a subsequent label

HINT: a nice repository to explore assembler instructions is
`https://www.felixcloutier.com/x86/`

Auto vectorizing a simple loop

Let's inspect what the compiler issues for `float` type

```
.L4:
        vaddss  xmm0, xmm0, DWORD PTR [rax]
        add     rax, 32
        vaddss  xmm0, xmm0, DWORD PTR -28[rax]
        vaddss  xmm0, xmm0, DWORD PTR -24[rax]
        vaddss  xmm0, xmm0, DWORD PTR -20[rax]
        vaddss  xmm0, xmm0, DWORD PTR -16[rax]
        vaddss  xmm0, xmm0, DWORD PTR -12[rax]

        vaddss  xmm0, xmm0, DWORD PTR -8[rax]
        vaddss  xmm0, xmm0, DWORD PTR -4[rax]
        cmp     rax, rcx
        jne     .L4
```

8 `vaddss` calls on xmm0 to itself after having loaded subsequent memory addresses

The loop is then not truly vectorized !

`vaddss target, opA, opB`

add the low SP FP from `opB` to `opA` and stores the result in low SP FP of `target`

Auto vectorizing a simple loop

To further test the result of the compilation, let's profile the codes using `perf`
find the details in `Vectorization/.c`

For both int and float versions we'll check the events

`cycles:u , instructions:u`

Whereas we'll check che specific events

`int_vec_retired.add_256`

and

`fp_arith_inst_retired.256b_packed_single:u , fp_arith_inst_retired.scalar_single:u`

for the int and float version respectively

Auto vectorizing a simple loop

To further test the result of the compilation, let's profile the codes using `perf`

	integer	float
not vectorized	0 int_vec.add_256	0 vector 1,000,000 scalar
vectorized	250,000 int_vec.add_256	250,000 vector 1,000,000 scalar

Auto vectorizing a simple loop

To further test the result of the compilation, let's profile the codes using `perf`

	integer	float
not vectorized	0 int_vec.add_256	0 vector 1,000,000 scalar
vectorized	250,000 int_vec.add_256	250,000 vector 1,000,000 scalar

125k are vector ops to initialize the loop.

125k are vec addition in the summation loop

these are the vector ops to initialize the loop.
125k to convert vectors of int into floats, 125k to mov regs in memory

these are the scalar in the summation loop, that are not vectorized at all

Auto vectorizing a simple loop

Hence, autovectorization works nicely for the integers but not at all for the floats.
Of course that descends from the compiler *not* being entitled to reshuffle operations in the summation loop.
For as we have written it

$$s \mathrel{+}= a_i$$

there is a clear critical path on `s`, that must be respected by the compiler.
Let's try by explicitly relaxing the safe math assumption(*).

	vectorized	vectorized unsafe-math
metrics	250,000 vector 1,000,000 scalar	375,000 vector 0 scalar

(*) *Note: Intel's compiler by default assumes the relaxed IEEE math*

Auto vectorization of loops

In the following we will examine what is needed to let the compiler vectorize loops for us and how to cure the most common issues that hinder the vectorization.

As we have just seen, to express the inherent loop data- and instruction-parallelism is of primary importance authorizing the compiler to re-shuffle floating-point operations.

references:
- *A guide to Vectorization with Intel® C++ compiler*
- *Requirements for vectorizable loops (Intel)*
- *Loop Independence, Compiler Vectorization and Threading of Loops, Intel®*
- *Vectorization with compilers, G. Zitzlsberger @ IT4I*
- *Cornell Virtual Workshop on Vectorization*

Auto vectorization of loops

Ask the compiler to vectorize

Tipically `-O2` or even `-O3` are required to vectorize, in addition to explicit compiler-dependent options.

GCC/clang

```
-ftree-vectorize        enables the vectorization of loops
-funroll-loops          enables the loop unrolling (may or may not
                        issue a faster code)
-march=native           specifies the target architecture
-mtune=native           specifies the architecture for code tuning
```

Intel

```
-xHost                  enables the maximum vectorization available on the
                        target cpu
-axFLAG1 -axFLAG2 ...   enables multiple targets. FLAGS are AVX,SSE4.2,
                        SSE4.1, ..
```

Auto vectorization of loops

Ask the compiler to report on optimizations and vectorization

Not surprinsingly, the output of the compiler may be full of insights on your code

GCC/clang

```
-fopt-info-vec-optimized   reports on the successful vectorizations
-fopt-info-vec-missed      reports on the reasons for unsuccessful
                           vectorizations
```

Intel

```
-qopt-report=<n>           enables the output of diagnosis
-qopt-report-file=name     n = 0 silent
check also                 n = 1 loops succesfully vectorized
-opt-report-phase          n = 2 loops not vectorized, and the reasons for
                           n = 3 dependency informations
                           n = 4 reports vectorization issues only
                           n = 5 reports vect. issues with dependency infos
```

Auto vectorization of loops

Basic requirements for vectorization

- **`countable loops, with no data-dependent control flow`**
  The iteration space must be known at run-time. The end of the loop can not depend on data (i.e.: loops without break instructions)

example of non-vectorizable loops (from Intel's guide) with data-dependent exit point

```
int i = 0.;
while (i < 100)
{
   a[i] = b[i] * c[i];
   if (a[i] < 0.0)
      break;
   ++i;
}
```

```
for ( int i = 0; i < size_of_data(); i++ )
{
   ....
}
```

example of non-vectorizable loops due to varying iteration space

Auto vectorization of loops

Basic requirements for vectorization

- `countable loops, with no data-dependent control flow`

- `loops without branching`
  since the SIMD instructions are meant to perform exactly the same operation on multiple data, different iterations can not have different control flows.
  if statements are admitted if they can result in a mask - like implementation (hile instructions are executed for all the elements, the results are propagated only for those whose mask entry is true)

example of vectorizable loop

```
for ( int i = 0; i < N; i++ )
{
   float tmp = a[i]*b[i] - a[i]/b[i];
   if ( tmp > 1 ) {
      c[i] = tmp }
   else { c[i] = sqrt(tmp) - exp(tmp); }
}
```

Auto vectorization of loops

Basic requirements for vectorization

- `countable loops, with no data-dependent control flow`

- `loops without branching`
  since the SIMD instructions are meant to perform exactly the same operation on multiple data, different iterations can not have different control flows.
  if statements are admitted if they can result in a mask - like implementation (hile instructions are executed for all the elements, the results are propagated only for those whose mask entry is true)

- `no function calls`
  for the same reason mentioned above.
  Vectorizable functions are an exception.

Auto vectorization of loops

What could go wrong ?

- `non-contiguous memory access`
  loops that access data with a stride, or even worse with an irregular non-contiguous pattern, are rarely vectorized.
  That is because while consecutive data can be loaded in a single cache line, or in a register, with a single memory operation, sparse memory access need separate loads and data gathering; if the compiler estimates that this overhead is larger than the benefits, the loop is not vectorized

  → `Vectorization/non_contiguous_access.c`

- `Data dependencies`
  fundamental fact : a loop is vectorizable if there are no cyclic dependencies chains among different iterations within the vector length.
  For practical purposes, any dependency beyond the vector length does not hinder vectorization.

  note: AVX512-CD implements Conflict-Detection

Auto vectorization of loops

What could go wrong ? Data Dependencies

- **`Read-After-Write (flow-dependency)`**
  A variable is written in an interation and read on a subsequent one

```
for ( int i = 1; i < N; i++ )
   a[i]= a[i-1] + c[i];
```

This loop can NOT be vectorized.

Auto vectorization of loops

What could go wrong ? Data Dependencies

- **`Read-After-Write (flow-dependency)`**
  A variable is written in an interation and read on a subsequent one

- **`Write-After-Read (anti-dependency)`**
  A variable is read in one iteration and written in a subsequent one.
  Unsafe for parallelism (no strict order among iterations assigned to tasks), normally safe for vectorization

```
for ( int i = 1; i < N; i++ )
   a[i-1]= a[i] + c[i];
```

This loop can be vectorized

```
for ( int i = 1; i < N; i++ ){
   a[i-1]= a[i] + c[i];
   sum += a[i]; }
```

This loop can NOT be vectorized because it's unclear if `a[i]` may be overwritten before the summation to `sum`

```
#pragma ivdep
 for ( int i = 1; i < N; i++ ){
   a[i-1]= a[i] + c[i];
   sum += a[i+4]; }
```

This loop can be vectorized

Auto vectorization of loops

What could go wrong ? Data Dependencies

- `Read-After-Write (flow-dependency)`
  A variable is written in an interation and read on a subsequent one

- `Write-After-Read (anti-dependency)`
  A variable is read in one iteration and written in a subsequent one.
  Unsafe for parallelism (no strict order among iterations assigned to tasks), normally safe for vectorization

- `Write-After-Write (output dependecy)`
  The same variable is written in more than one iteration.
  Generally unsafe both in parallelization and in vectorization.
  Tyipical example: a critical path

```
double sum = 0;
for ( int i = 1; i < N; i++ )
   sum += a[i] + c[i];
```

Auto vectorization of loops

What could go wrong ? Data Dependencies

- `Read-After-Write (flow-dependency)`
  A variable is written in an interation and read on a subsequent one

- `Write-After-Read (anti-dependency)`
  A variable is read in one iteration and written in a subsequent one.
  Unsafe for parallelism (no strict order among iterations assigned to tasks), normally safe for vectorization

- `Write-After-Write (output dependecy)`
  The same variable is written in more than one iteration.
  Generally unsafe both in parallelization and in vectorization.

- `Unclear dependencies`
  Tipically due to memory aliasing

Auto vectorization of loops

What could go wrong ? Memory disambiguation

- **`Unclear dependencies`**
  Tipically due to memory aliasing

```
void function ( int *a, int *b, int *c, int N )
 {
   for ( int i = 0; i < N; i++ )
     c[i]= a[i] + b[i];
 }
```

It may be impossible for the compiler to decide whether, for some value of ***`i`****,* ***`a[i]`****,* ***`b[i]`*** *and* ***`c[i]`*** *will somehow overlap.*

If that is the case, it will not issue a vector code.

When it is possible, the compiler will issue more than one version of the loop, and a code that performs an overlap test among the arrays.
Then the execution will be routed to the correct version at run-time

Auto vectorization of loops

What could go wrong ? Memory disambiguation

- **`Unclear dependencies`**
  Tipically due to memory aliasing

```
void function ( int *a, int *b, int *c, int N )
 {
   for ( int i = 0; i < N; i++ )
     c[i]= a[i] + b[i];
 }
```

you can easily **disambiguate**:

```
void function ( const int * restrict a, const int * restrict b, int * restrict c, const int N )
 {
   for ( int i = 0; i < N; i++ )
     c[i]= a[i] + b[i];
 }
```

The `restrict` keyword instructs the compiler that every restrict pointer will never overlap with any other pointer

The `const` qualifier adds important hints about what to optimize and how. Also add clarity for the programmers.

Note: a pointer to constant data is different than a costant pointer

Auto vectorization of loops

What could go wrong ? Unaligned memory

Auto vectorization of loops

What could go wrong ? SoA over AoS

Addressing modes in x64 assembly

That`s all folks, have fun