Some elements on Vectorization

SCIENTIFIC & DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F.

Advanced HPC 2024-2025 @ Università di Trieste

Outline

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

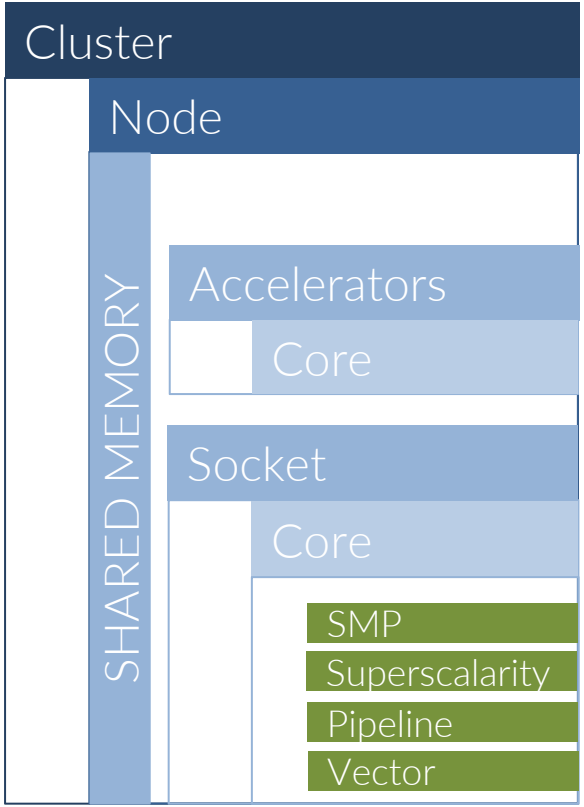
Main sources

- 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

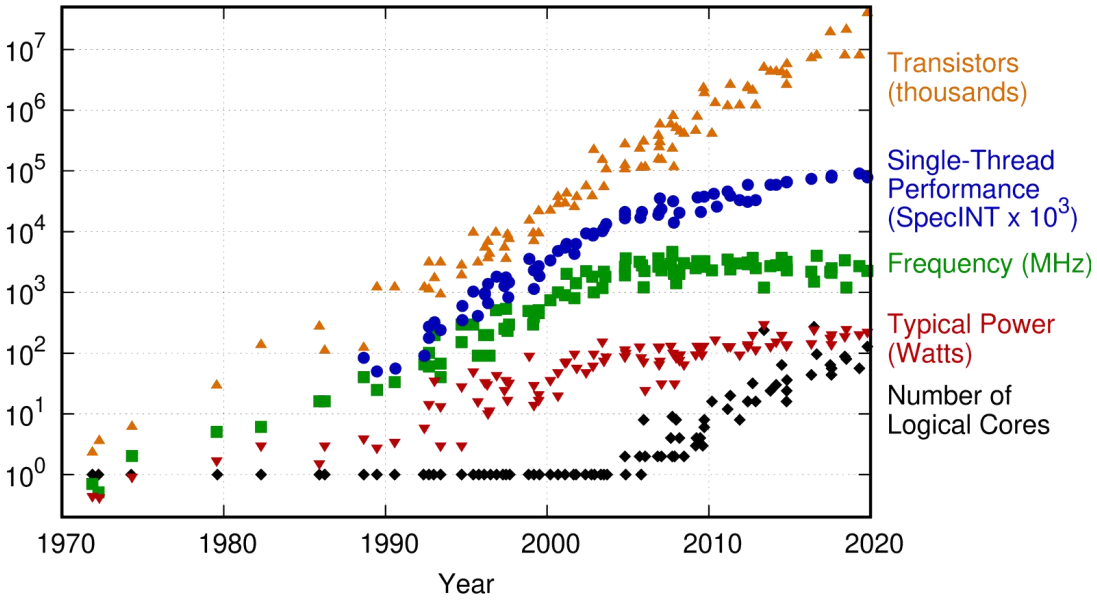
Single instructions on Multiple Data

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.



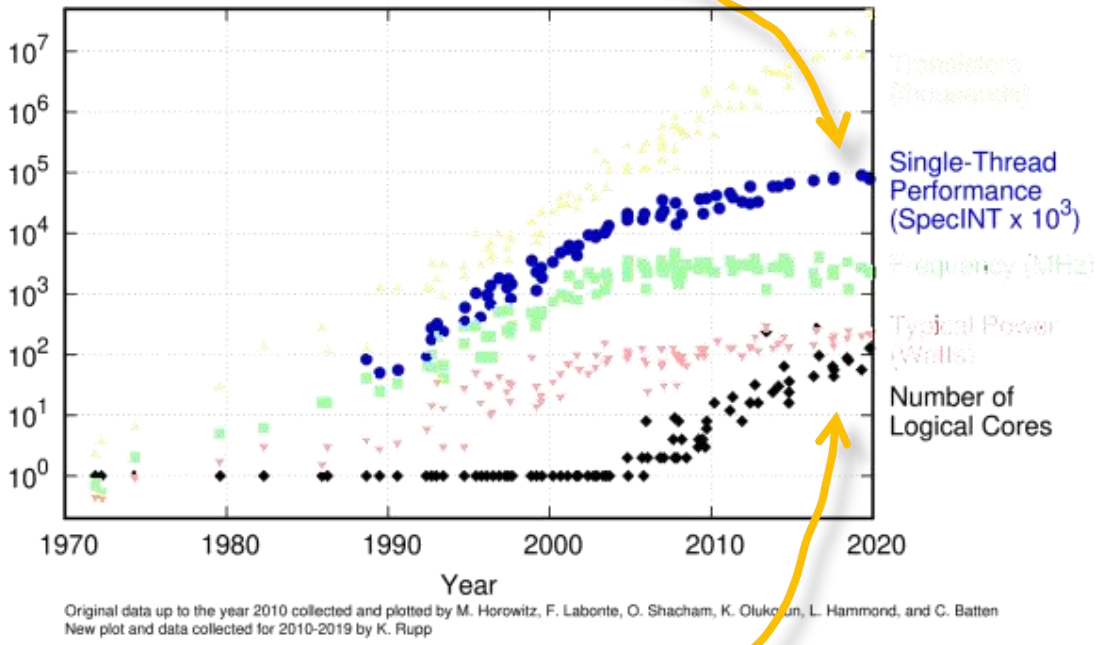
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

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



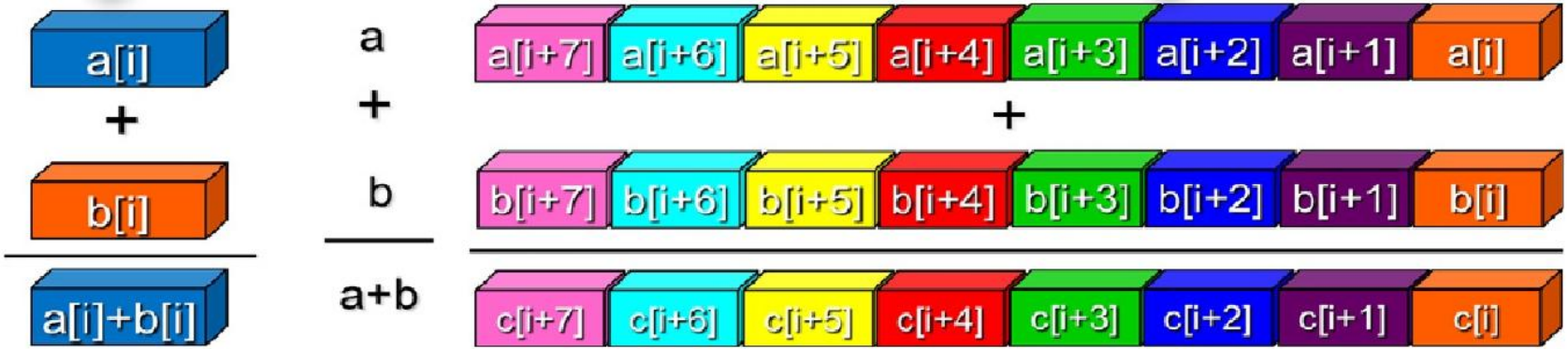
More cores to increase parallelism since the frequency has hit the power wall

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];
```

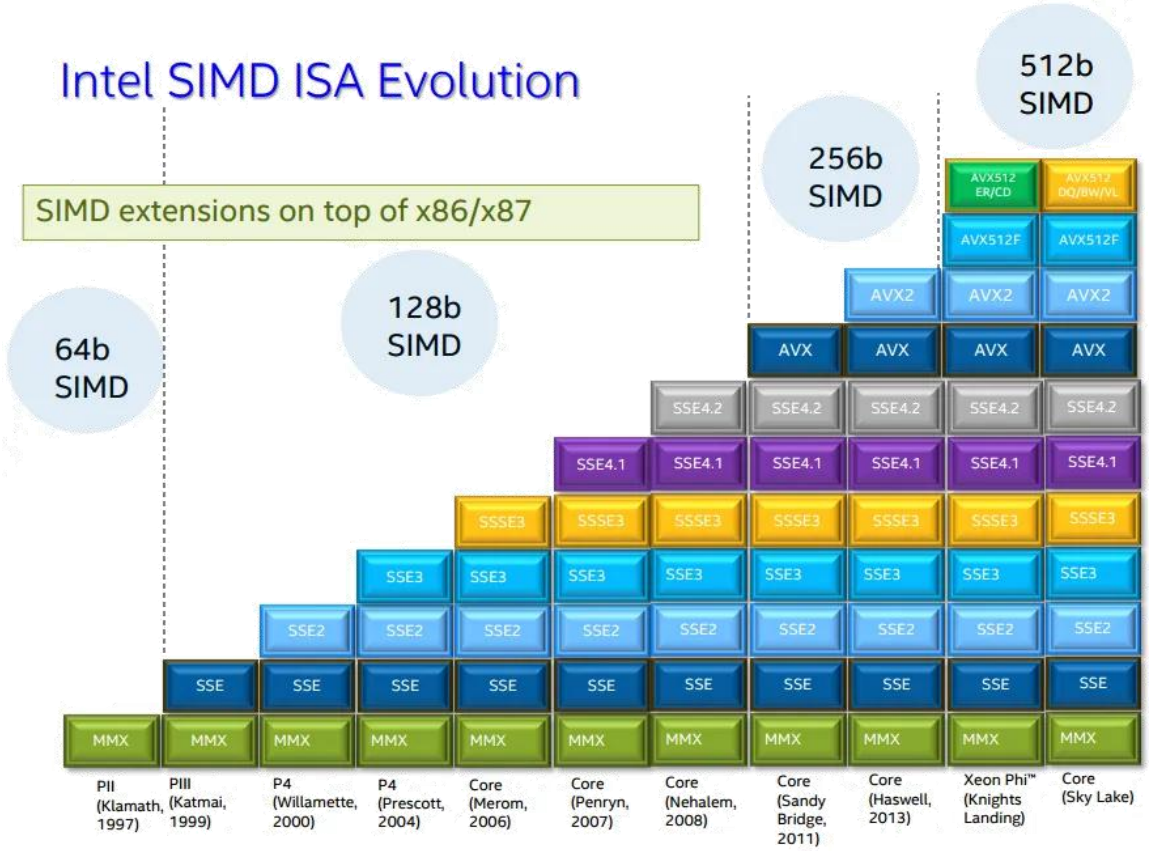


taken from "High Performance Parallelism Pearls, vol 2" , ch. 22

Vector instructions, evolution

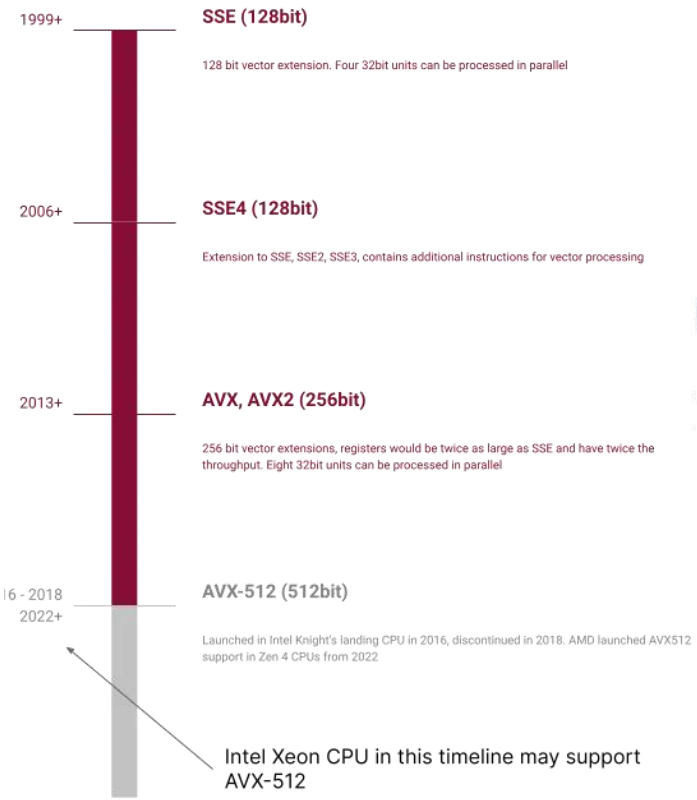
Example for Intel ISA.
AMD (3dNow),
ARM (SVE, Neon), IBM (AltiVec),
RISC-V (V), etc,
have their own.

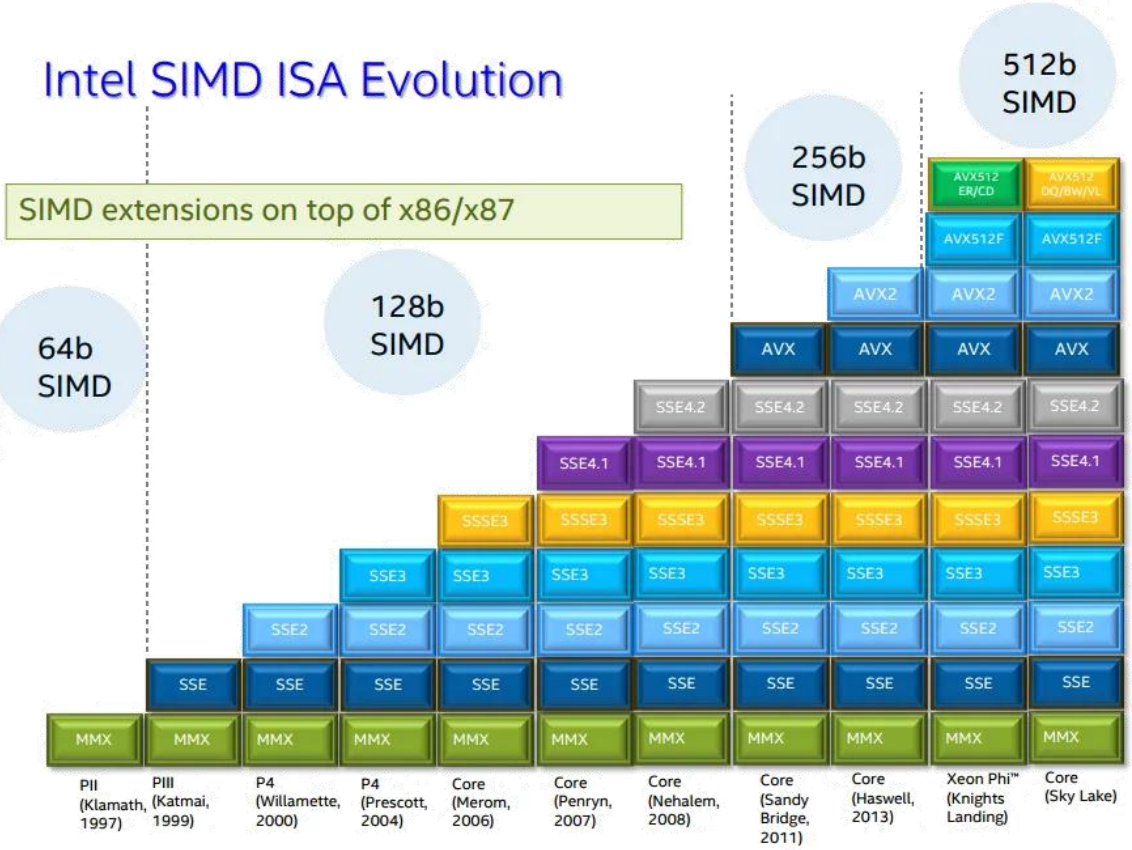
Different ISA share several similar features while having some peculiar ones.



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

Vector instructions, evolution





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

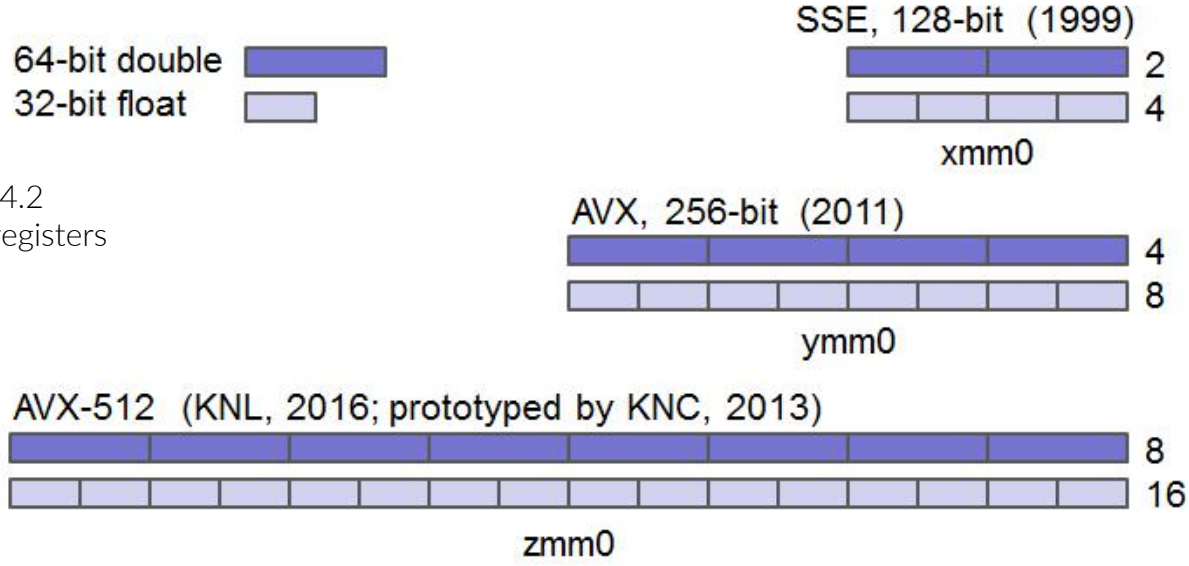
Vector instructions, registers size

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.

```
#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
```

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

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

```
#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

That`s all folks, have fun