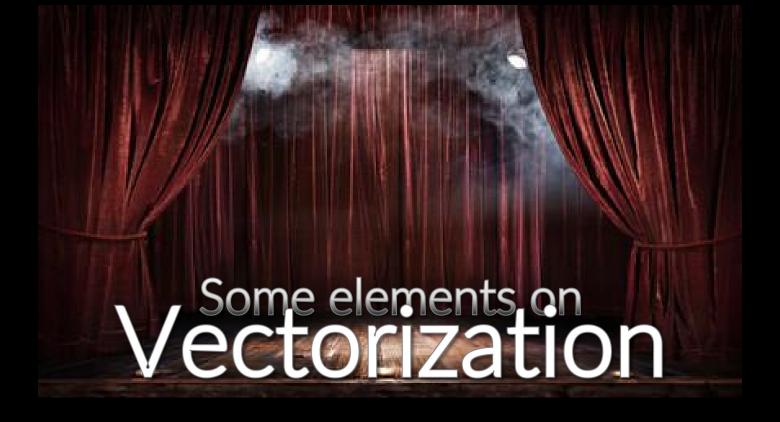




Luca Tornatore - I.N.A.F.

Advanced HPC 2024-2025 @ Università di Trieste







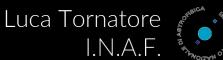


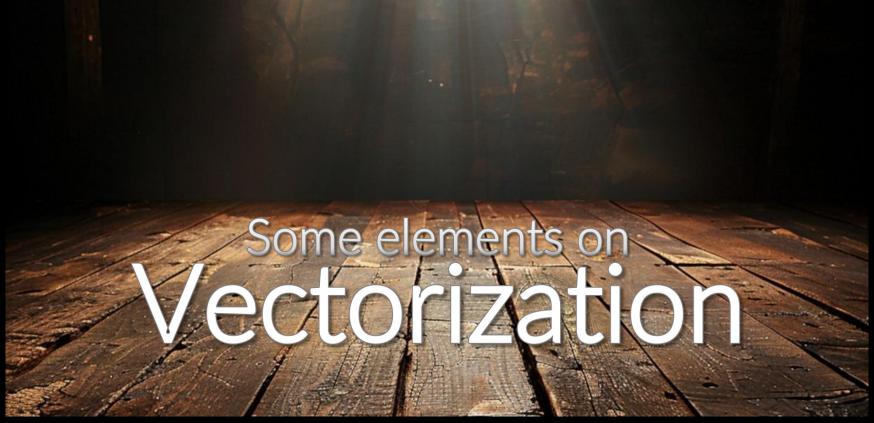
















Outline

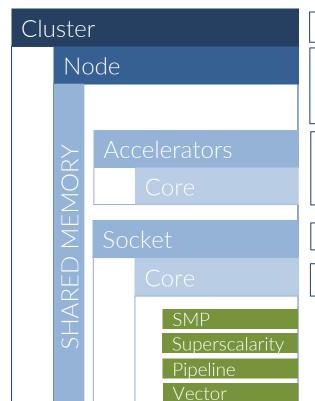
- Introduction
- Understanding the CPU's capability
- Vectorization by compiler's wow effect
- Vectorization by vector types
- Vectorization by OpenMP SIMD

[o]

Introduction

- Recap: the levels of parallelism
- What are vector registers and vector ops
- Why vectorization?
- Evolution of vector sizes and instructions
- High-level and Low-level approaches for vectorization

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

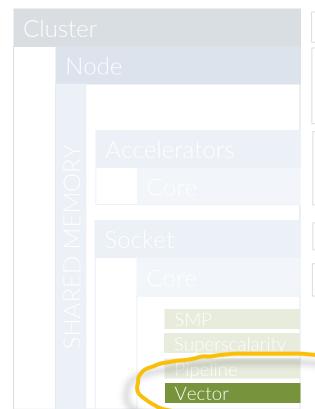
Group of instructions sharing functional units thanks to out-of-order capability

Multiple instructions sharing functional units

Single instructions on Multiple Data



Recap: the levels of parallelism



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

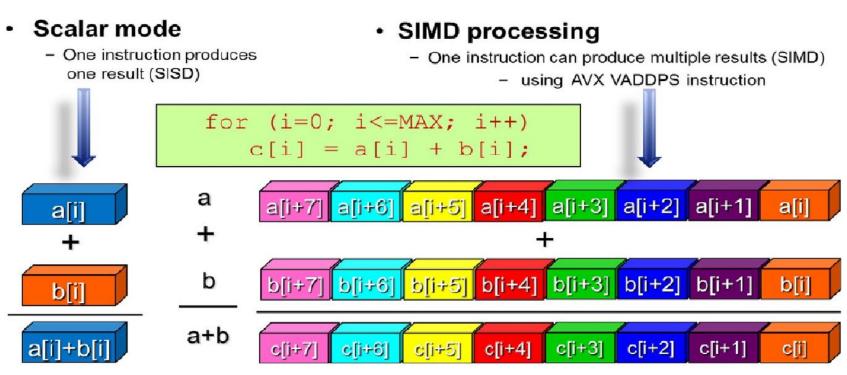
Multiple instructions sharing functional units

Single instructions on Multiple Data



Vector instructions

What is the nature of vector operations?



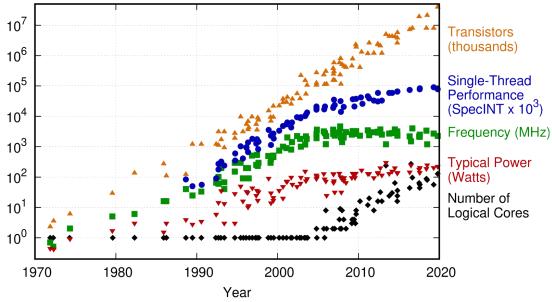


Recap: why vectorization?

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 k^2 . But the number of these elements per unit area increases by k^2 , 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



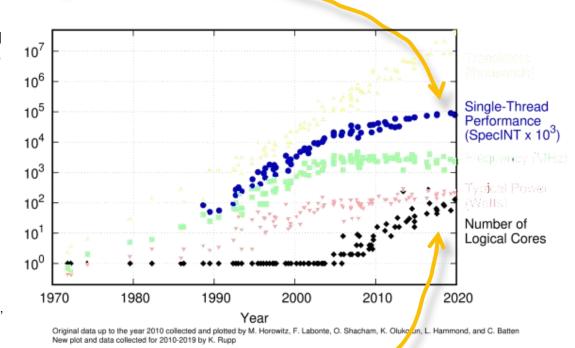
Recap: why vectorization?

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 k^2 , 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



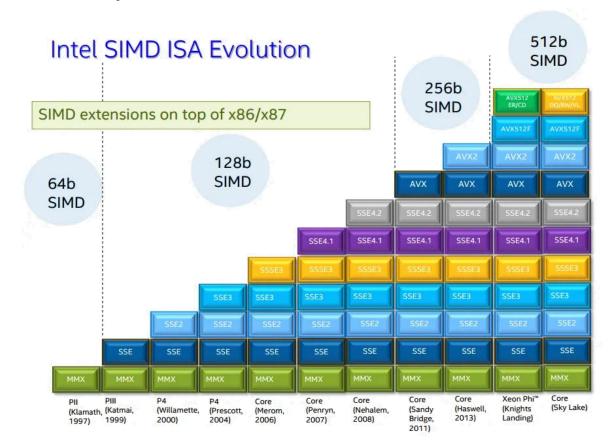
More cores to increase parallelism since the frequency has hit the power wall



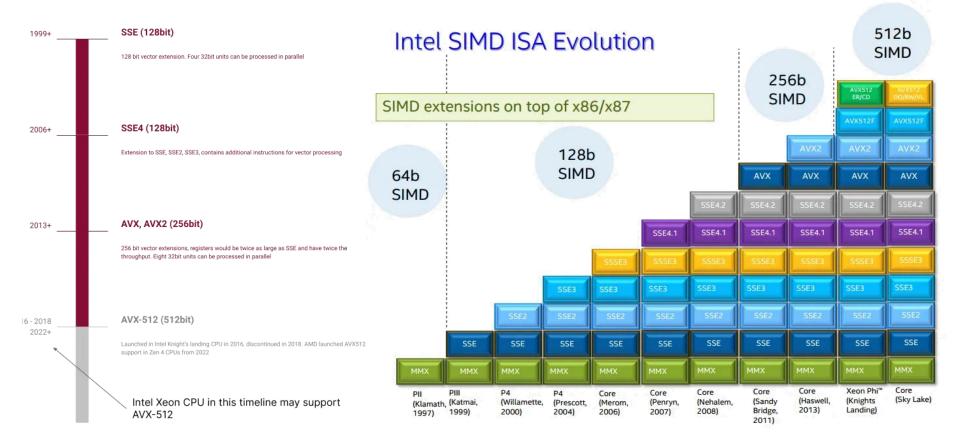
Vector instructions, evolution

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.

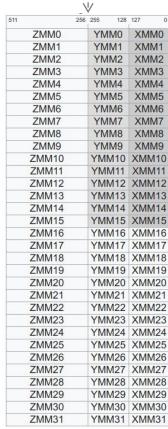


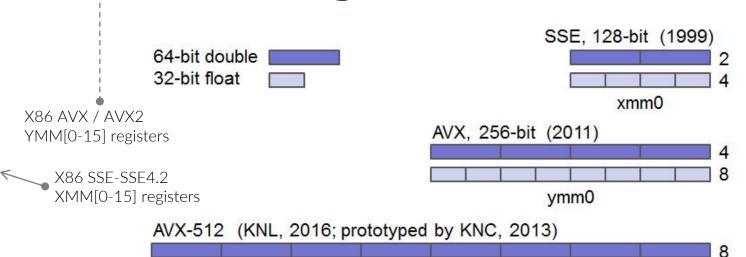
Vector instructions, evolution





Vector instructions, registers size





A speedup as large as the size of the registers (4x, 8x) may be possible in scientific codes

zmm0

However, that result is rarely achieved for a number of reasons that we'll discuss in the next slides.

Also, due to power consumption, normally the CPU frequency is throtthled down when intensive vector operations are running

16

From the Wikipedia page

on AVX-512

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



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



✓ Compiler Auto-Vectorization

C++ classes

✓ Compiler-dependent vector types

Intrinsics

Assembler





How to check for SIMD capablities

- Checking the CPU flags from command line
- Checking the CPU capabilities via builtin functions
- Compile code dependent on the capabilities of the target machine

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



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");
```

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
```

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

Vectoriztion/O .../discover vector support.o

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
```

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

```
:> $ gcc -o discover_vector_support
discover_vector_support.c
:> $ ./discover_vector_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 **-03** does not make it to behave appropriately

Exercise: compile & run support.c

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

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

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

```
:> $ gcc -msse4.2 -o ...
:> $ ./discover_vector_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

Exercise: compile with icx and clang

ARM, POWER, and others have their own specific targets

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

Add the directive

#pragma GCC target("avx2")

→ Vectoriztion/0 .../declare target.c at the top of the code
However this goes in the direction of being strongly compiler-dependent,
which is not advisable

```
#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)

- Recap of vector types & sizes
- How you can define your own "adaptive" vector types

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	128bits
64bits	m64	8x	4x	2x	1x	
128bits	m128i	16x	8x	4x	2x	256bits
256bits	m256i	32x	16x	8x	4x	
512bits	m512i	64x	32x	16x	8x	512bits

FLOATING-POINT types

types name	single precision	double precision
m128	4x	-
m128d	-	2x
m256	8x	-
m256d	-	4x
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/0_explore_first_steps/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 )
```

Defining vector types via intrinsics

What you do in real codes is to have large **#idef** regions in which you define the macros for the precise operations that you need. This requires to call the AVX512, AVX, SSEx. intrinsics that refer to that operations with the suited types.

For instance a "vector summation" for int, double and single precision FP

```
VDSUM (v1, v2, result)
VFSUM (v1, v2, result)
VISUM (v1, v2, result)
```

with a standard call throughout your code

We'll see more details when discussing the usage of intrinsics

```
#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

- How to use the compiler and Basic requirements for loop vectorization
- Getting insights from a simple example
- Recap on critical path and loop unrolling
- Associative math
- What could go wrong?
- Inefficiencies



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
- ...



Loops Auto-vectorization

Ask the compiler to vectorize

Vectorization/headers/vector_pragmas.h

Tipically -02 or even -03 are required to vectorize for all compilers, in addition to explicit compiler-dependent options (why not to opt for 03, anyway?).

GCC

-ftree-vectorize	enables loops vectorization
-funroll-loops	enables the loop unrolling (may or may not issue faster code)
-march=native	specifies the current one as target architecture
-mtune=native	ask maximum code tuning for the host architecture

clang

-mllvm	
-force-v	vector-width=4

Intel

-xHost	enables the maximum vectorization available on the target cpu
-axFLAG1 -axFLAG2	enables code for multiple targets



Auto vectorization of loops

Ask the compiler to report on optimizations and vectorization

GCC

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

clang

-Rpass=loop-vectorize	identifies loops that were successfully vectorized
-Rpass-missed=loop-vectorize	identifies loops that were NOT successfully vectorized
-Rpass-analysis=loop-vectorize	dentifies the statements that caused vectorization to fail. If in addition -fsave-optimization-record is provided, multiple causes of vectorization failure may be listed

enables the output of diagnosis

Intel

	enables the output of diagnosts
	<pre>n = 1 loops succesfully vectorized</pre>
-qopt-report= <n></n>	n = 2 loops not vectorized, and the reasons for
-qopt-report-file= <i>name</i>	n = 3 dependency informations
	n = 4 reports vectorization issues only
	n = 5 reports vect. issues with dependency infos



Auto vectorization of Give hints and directions to the compiler

Vectorization/headers/ vector_pragmas.h

GCC

#pragma GCC ivdep	ignore potential loop-carried dependencies that are not formally proven
#pragma GCC unroll n	unroll the subsequent loop body n times

clang

#pragma clang ivdep	ignore potential loop-carried dependencies
<pre>#pragma clang vectorize (<enable disable>)</enable disable></pre>	enable the vectorization of the following loop
<pre>#pragma clang vectorize_width (N)</pre>	specifies the VL
<pre>#pragma clang interleave (<enable disable>)</enable disable></pre>	enable the unrolling of the following loop
<pre>#pragma clang interleav_count (N)</pre>	specifies how many iterations are to be unrolled

check the compiler's manual for comprehensive and updated descriptions



Auto vectorization of sps Give hints and directions to the compiler

→ Vectorization/headers/ vector_pragmas.h

Intel

#pragma ivdep	ignore potential loop-carried dependencies
#pragma vector always	vectorize even if the estimated gain is low or negative
#pragma vector aligned	assume aligned data
#pragma vector unliagned	assume unaligned data

check the compiler's manual for comprehensive and updated descriptions



Auto vectorization of

Give hints and directions to the compiler

```
#define STRINGIFY(X) #X
#define DO PRAGMA(x) Pragma (#x)
#if defined( GNUC ) && !defined( clang )
#pragma message "using GCC"
#define TVDFP
                          Pragma("GCC ivdep")
#define LOOP VECTORIZE
                          Pragma("GVV ivdep")
#define LOOP VECTOR LENGTH(N)
#define LOOP UNROLL
                          Pragma("GCC unroll")
#define LOOP UNROLL N(N)
                          DO PRAGMA(GCC unroll N)
#elif defined( clang )
#define TVDFP
                             Pragma("clang ivdep")
#define LOOP VECTORIZE
                             DO PRAGMA(clang loop vectorize( enable ))
#define LOOP VECTOR LENGTH(N) DO PRAGMA(clang vectorize width( N ))
#define LOOP UNROLL
                             DO PRAGMA(clang loop interleave( enable ))
#define LOOP UNROLL N(N)
                             DO PRAGMA(clang loop interleave count( N ))
```

Pragma("ivdep")

Pragma("ivdep")

Pragma("vector always")

Pragma("vector aligned")

check the example file

Vectorization/headers/vector_pragmas.h

to have an example of how you may define your own pragmas without bothering about which compiler you're using

#elif defined(ICC)

#define LOOP VECTORIZE

#define VECTOR ALIGNED

#define VECTOR ALWAYS

#define IVDEP

#define VECTOR UNALIGNED Pragma("vector unaligned")

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**).

Let's compile the code

Vectorization/1_simple_loop/sum_loop.c

for both integer and float data, asking for the vectorization report and generating the assembler:

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

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

```
gcc -S -fverbose-asm -masm=intel

-03 -march=native -mtune=native -ftree-vectorize -funroll_loops
-fopt-info-vec-optimized -fopt-info-vec-missed -fopt-info-loop -fopt-info-loop-missed
-DDTYPE=[INTEGER|FPSP] -o sum_loop.[int|float].s sum_loop.c
```



Let's inspect what assembler the compiler issues for int type

```
.L4:
# sum_loop.c:51:_
       vpaddd ymm0, ymm0, YMMWORD PTR [rax] * # vect sum 11.16, vect sum 11.16, MEM <vector(8) unsigned int> [(uint32 t *) 69]
                                           # ivtmp.22,
              rax. 32
                                            53, ivtmp.22
              rdx, rax

    the actual loop

                                           # tmp142, vect sum 11.16
       vmovdqa xmm1, xmm0
       vextracti128
                     xmm0, ymm0, 0x1
                                           # tmp143, vect sum 11.16
                                           # niters vector mult vf.10,
              edx, ecx
       MOV
       vpaddd xmm0, xmm1, xmm0
                                           # 41, tmp142, tmp143
                                           # niters vector mult vf.10,
              edx, -8
              cl. 7
                                           # N.
                                           # tmp145, 41,
       vpsrldq xmm1, xmm0, 8
       vpaddd xmm0, xmm0, xmm1
                                           # _43, _41, tmp145
       vpsrldq xmm1, xmm0, 4
                                           # tmp147, 43,
       vpaddd xmm0, xmm0, xmm1
                                           # tmp148, 43, tmp147
              eax. xmm0
                                           # <retval>, tmp148
       vmovd
       ie
                                           #.
       vzeroupper
                           vpaddd
                                             ymm0, ymm0, YMMWORD PTR
                           add
                                             rax, 32
                                             rdx, rax
                           CMD
                            ine
                                              . L4
```

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

- vpadd target, opA, opB \rightarrow 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
```

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

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

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

Let's inspect what the compiler issues for int type

```
.L4:
# sum loop.c:51:
                    sum += array[i];
                                              # vect sum 11.16, vect sum 11.16, ME
       vpaddd ymm0, ymm0, YMMWORD PTR [rax]
               rax. 32
                                              # ivtmp.22.
               rdx, rax
                                              # 53, ivtmp.22
       vmovdqa xmm1, xmm0
                                              # tmp142, vect sum 11.16
       vextracti128
                       xmm0. vmm0. 0x1
                                              # tmp143, vect sum 11.16
                                              # niters vector mult vf.10, N
               edx, ecx
                                              # _41, tmp142, tmp143
       vpaddd xmm0, xmm1, xmm0
               edx. -8
                                              #_niters vector mult vf.10,
       and
               cl. 7
       test
                                              # tmp145, 41,
       vpsrldq xmm1, xmm0, 8
       vpaddd xmm0, xmm0, xmm1
                                                _43, _41, tmp145
       vpsrldq xmm1, xmm0, 4
                                              # tmp147, 43,
       vpaddd xmm0, xmm0, xmm1
                                              # tmp148, 43, tmp147
                                              # <retval>. tmp148
               eax, xmm0
       vzeroupper
                                   if the iteration space was not exactly
                                   divisible by 8, we need to account for
                                   the remainder iterations:
                                   otherwise, we jump to a subsequent
```

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



More details on how to read this assembly in the slides at the end

HINT: a nice repository to explore assembler instructions is

https://www.felixcloutier.com/x86/



label

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]
        CMD
               гах, гсх
                .L4
        ine
```

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**



To further test the result of the compilation, let's profile the codes using perf find the details in Vectorization/1_simple_loop/compile_sum_loop

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



For a recap on Performance Monitoring Units (PMUs), perf and the perf profiling, have a look at the pdf

Materials/profiling with perf.pdf

which is the same that you have seen in the HPC basic course

If you work on a big.LITTLE system, you may encounter problems in getting the PMUs working properly. You need to export a precise LIBPFM_FORCE_PMU



What I want to convince you with the following slides is:

- integers can be easily vectorized by the compiler because there are no contraints on the reshuffling of the operations
- FP do not have this advantage in fact even when some vectorization is possible, the *critical paths* that we insert through the semantics hinder the expression of whole the vectorization
- "unsafe math" options, by relaxing the IEEE complaiancy, may offer a viable solution but they also bring in some perils. You need to understand well what they are about.
- Re-engineering the code may express much more parallelism, and the vectorization achieved by the compiler may be comparable to that with "unsafe math".

look at the comments in Vectorization/1_simple_loop/compile_sum_loop .



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

```
Performance counter stats for 'CPU(s) 1':

10,939,564 cpu_core/cpu-cycles/
10,360,258 cpu_core/instructions/
250,000 cpu_core/fp_arith_inst_retired.256b_packed_single/
1,000,000 cpu_core/fp_arith_inst_retired.scalar_single/
0.005113507 seconds time elapsed
```



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.

result of the compilation, let's profile the codes using perf

125k are vec addition in the summation loop

summation loop	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 <

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



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

$$s += a_i$$

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(*) by -fassociative-math.

	vectorized	vectorized unsafe-math
metrics	250,000 vector 1,000,000 scalar	500,000 vector 0 scalar

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





Just a recap about the issue brought in the optimization of loops by the non-commutativity of floating-point airthmetics.

Please get back to the lectures on loop- and pipeline- optimizations given in the basic HPC course for a more comprehensive treatment.

```
If we excute the loop
   for ( int i = 0; i < N; i++ )
       S += a[i];</pre>
```

we immediately appreciate that it encodes a precise, although obvious, order of operations:

i	s :
0	a[0]
1	a[0] + a[1]
2	a[0] + a[1] + a[2]
3	a[0] + a[1] + a[2] + a[3]
	• • •



That pattern is clearly different than that encoded by different loop forms

more instruction parallelism is exposed here: a[2]+a[3] could be "in flight" at the moment in which a[0]+a[1] is being added to S

i S :
$$a[0] + a[1] + (a[2] + a[3]) \leftarrow a[4] + a[5] + (a[6] + a[7])$$

and even more instruction parallelism is exposed here: four elements and three adds of the next iteration can be "in flight" while the current iteration is being processed



That pattern is clearly different than that encoded by different loop reshapes

```
for ( int i = 0; i < N; i+=2 )
   S += (a[i] + a[i+1]);
      a[0] + a[1]
      (a[0] + a[1]) + (a[2] + a[3])
for ( int i = 0; i < N; i+=4 )
  S += (a[i] + a[i+1]) + (a[i+2] + a[i+3]);
      (a[0] + a[1]) + (a[2] + a[3])
```

(a[4] + a[5]) + (a[6] + a[7])

However, although much more parallelism has been exposed in these two re-shapings of the loop [NOTE THAT the compiler CAN NOT perform them if not explicitly authorized] there is still a fundamental one.

All the partial summation are added up, in a precise order, to the same variable, the unique accumulator S.

This will lead to the execution of a lot of bit reshuffling and scalar add, in place of clean vector operations

There is an obvious fix for that.



That pattern is clearly different than that encoded by different loop reshapes

```
for ( int i = 0; i < N; i+=4 ) {
    S0 += a[i];
    S1 += a[i+2];
    S2 += a[i+2];
    S3 += a[i+3]; }</pre>
```

Using more partial accumulators, of course

```
i S0 S1 S2 S3
0 a[0] a[1] a[2] a[3]
1 a[0]+a[4] a[1]+a[5] a[2]+a[6] a[3]+a[7]
```



That pattern is clearly different than that encoded by different loop reshapes

```
for ( int i = 0; i < N; i+=4 ) {
        S0 += a[i];
        S1 += a[i+2];
        S2 += a[i+2];
        S3 += a[i+3];
array a
S0-S3
iter 0
iter 1
iter 2
```

Using more partial accumulators, of course

That exposes a natural vector pattern

iter 3

Now that we have refreshed the critical paths, and we understand why the poor performance of the automatic vectorization, we can re-write the loop differently:

- unrolling
- using separated accumulators

and we obtain:

	unroll vectorized	vectorized unsafe- math
metrics	500,000 vector 0 scalar	500,000 vector 0 scalar

```
dtype sum loop ( dtype * restrict array, const
uint N )
  double sum0 = 0;
  double sum1 = 0;
  double sum2 = 0;
  double sum3 = 0;
 uint N4 = N%0xFFFFFFFC;
  for ( uint32_t i = 0; i < N4; i++ ) {
     sum0 += array[i];
     sum2 += array[i+1];
     sum3 += array[i+2];
     sum3 += array[i+3]; }
  < .. remainder loop and reduction on sum? >
  return sum0;
```

Vectorization/1_simple_loop/sum_loop.unroll.c



Now that we have refreshed the critical dtype sum loop (dtype * restrict array, const path: perfo vecto Hence, re-engineering the code *before* using the auto-vectorization diffe of the compiler can greatly enhance the achieved vectorization - unr because we expose more parallelism while explicitly relaxing - in a - usir controlled and correct way - the order of the operations. and Use "unsafe math", "associative math" and alike options cum grano salis metri sum? >

Vectorization/1_simple_loop/sum_loop.unroll.c



→Vectorization/3 fma loop/test.c

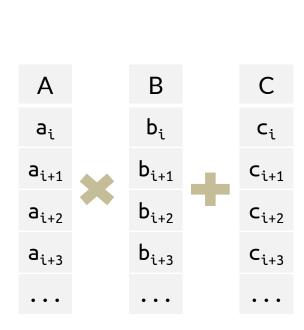
```
double kernel ( double *A, double *B, double *C, int N )
{
  double sum = 0;
  for ( int i = 0; i < N; i++ )
     sum += A[i]*B[i] + C[i];
  return sum;
}</pre>
```

This case has an higher possible parallelism since the +reduction comes after the multiply-and-add vector operation.

It means that a vector of partial results

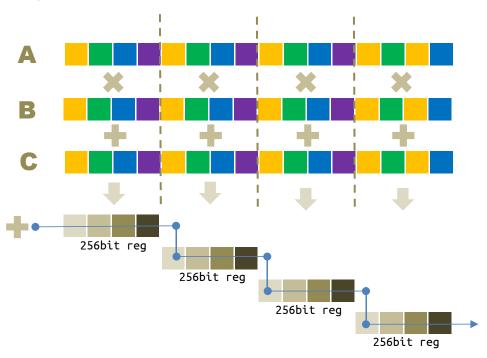
$$D[i:i+n] = A[i:i+n]xB[i:i+n]+C[i:i:n]$$

can be preemptively prepared, waiting for the **D[**] entries to be summed up serially





In fact, the compiler opts for this implementation:



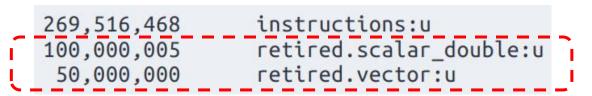
```
double kernel ( double *A, double *B, double *C, int N )
{
   double sum = 0;
   for ( int i = 0; i < N; i++ )
      sum += A[i]*B[i] + C[i];
   return sum;
}</pre>
```

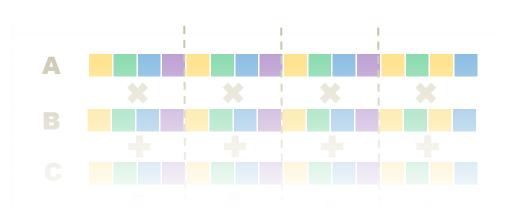
Lots of computation are performed via FMA on vector registers, 256b or 512b depending on what is available on the cpu.

However, lots of reshuffling is needed and all the reduction summations are serial







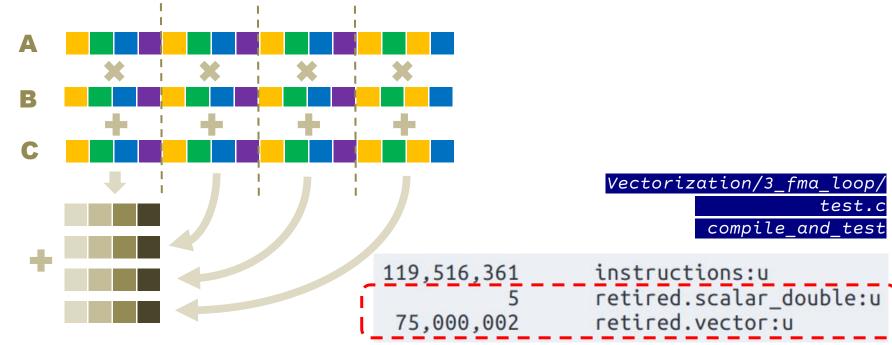


example run with AVX/AVX2 and size 100M elements: 25M + 25M FMA vector ops (delivered in 1ins)

100M summations



Opting either for the usage of the "associative math" flags or for explicit loop unrolling (with separated accumulators) results in a different, fully vectorizable pattern



Associative math



Every compiler allows the users to relax the rules for IEEE FP arithmetics.

Many of these rules ensure the correct treatment of under-normal numbers. 0+ and 0

Many of these rules ensure the correct treatment of under-normal numbers, 0+ and 0-, Inf and NaN. As such, relaxing the checks and the branches that deal with those cases, may incur in some un-correct result.

Explore two resources to have a better comprehension of this

https://simonbyrne.github.io/notes/fastmath/

https://kristerw.github.io/2021/10/19/fast-math/

Appreciate that also explicit tests as **isinf(x)** or **isnan(x)** will be disabled. As such, to be confident in activating those switches, you must be sure that your code is not incurring in those cases. However, by definition, the tests performed without those flags are not definitive because once the operations are re-shuffled, which is due exactly to those flags, the results may be different.

As we have just seen, to express the whole inherent loop data- and instruction- parallelism is of primary importance to either

authorizing the compiler to re-shuffle floating-point operations

or

unroll the loops to expose more instruction- and data- parallelism

Since the flag "just reorder ops associatively" does not exist, my personal advice is to opt for the manual refactoring of the code.

This way, in fact, you can reliably test for the correctness of the results while

achieving the same vectorization.



- 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 @ IT41
- Cornell Virtual Workshop on Vectorization



[3]

Auto-vectorization

General guidelines for loop vectorization (hold also for non compiler-based vectorization)

- What could go wrong?
 - non-contiguous mem access
 - data dependencies
 - memory ambiguity
- Inefficiencies:
 - unaligned memory
 - SoA vs AoS vs SoAoS



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 Note that a countable while loop is vectorizable as a for loop is

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

```
for ( int i = 0; i < size_of_data(); i++ )
{
    ....
}
example of non-vectorizable for loop, due to
unpredictable iteration space</pre>
```

Basic requirements for vectorization

- countable loops, with no data-dependent control flow The iteration space must be known at run-time.
- 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)

Basic requirements for vectorization

• countable loops, with no data-dependent control flow The iteration space must be known at run-time.

• 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)

avoid function calls

Obvious, for the same reason mentioned above.

Vectorizable functions are an exception.



What could go wrong?

non-contiguous memory access

loops that access data with a non-unit stride, or even worse with an irregular 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/2 loop dependencies/non contiquous access.c



What could go wrong?

non-contiguous memory access

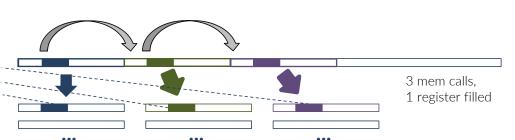
contiguous memory access results in fetching and using of entire lines of cache; the bandwidth utilization is optimal

memory segments mapped to cache lines

3 mem calls,
3 cache lines,
3 register filled

registers |

strided memory access results in gathering sparse memory location, accessed with separate memory calls, with a lot of overhead in memory calls and in mem moves from/to registers



These loops **could** be vectorized but the compiler may evaluate that the cost of the memory overhead exceeds the gain from the vectorization.



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.

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

- 1. Read-After-Write
- 2. Write-After-Read
- 3. Write-After-Write
- 4. Memory ambiguity



What could go wrong? Data Dependencies

Read-After-Write (flow-dependency)

Vectorization/2_loop_dependencies/raw.c

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 without leading to wrong results.

However, let's consider the following 2D case:

```
for ( int i = 1; i < M; i++ ) {
    a[i][0] = c[i][0];
    for ( int j = 1; j < N; j++)
        a[i][j]= a[i][j-1] + c[i][j];</pre>
```

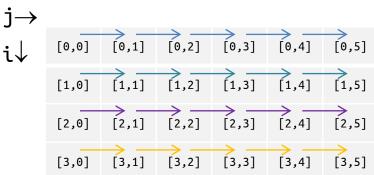
The dependency is carried in the *inner* loop, while the *outer* loop iterations do not exhibit any dependency and the outer loop vectorization can be performed.

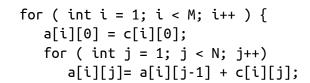


What could go wrong? Data Dependencies

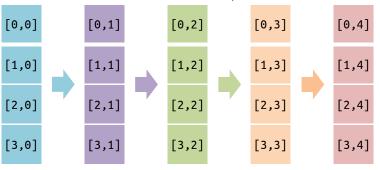
Read-After-Write (flow-dependency)

Consider the dependency graph of the iteration space. It clearly shows that there are no vertical dependencies (i.e. carried in the outer loop).





4 (or 8) outer iterations can be executed together because there are no vertical dependencies



However, if the elements are stored in memory in row-major order that require a lot of memory gather and scatter opearations. If not supported in hw, they must be emulated, which is even more expensive.

We may consider to write the data column-major, if possibile (it depends on the context).

The compiler tipically will not be able to perform this optimization.



What could go wrong? Data Dependencies

- Read-After-Write (flow-dependency) Vectorization/2 loop dependencies/raw.c A variable is written in an interation and read on a subsequent one
- Write-After-Read (anti-dependency) → Vectorization/2 loop dependencies/war.c 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



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.

```
double sum = 0;
for ( int i = 1; i < N; i++ ) {
    a[i-1] = function1(i);
    a[i] = function2(i); }
```



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



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 pointers.

Then the execution will be routed to the correct version at run-time



What could go wrong? Memory disambiguation

Unclear dependencies

Tipically due to memory aliasing

```
void function ( int *a, int *b, int *c, int N )
                                                              The restrict keyword grants to the
    for ( int i = 0; i < N; i++ )
                                                              compiler that every restrict pointer will
      c[i]=a[i]+b[i];
                                                              never overlap with any other pointer
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 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 to data and than a constant pointer to constant data...

What could be inefficient? Unaligned memory

A variable is said to be *aligned* in memory when its first byte's address is a multiple of the variable's type size:

```
2 bytes: 0, 2, 4, 6, 8 .. are aligned placements
short int
                     4 bytes: 0, 4, 8, 12, 16 .. are aligned placements
int / float
long long / double
                     8 bytes: 0, 8, 16, 24 .. are aligned placements
```

Consequently, an aligned variable is also aligned w.r.t the cache line it belongs to: it means that an aligned variable will never cross 2 cache lines.

What could be inefficient? Unaligned memory

However, let's imagine that you allocate a bunch of memory, and you consider it as an array of elements with byte size s_B . Let's say that

- its starting byte is aligned at a_B bytes (actually, that is usually expressed in bits)
- you access this array by vectors of element-size V_L (meaning that every vecot has V_L elements); hence, you will access $V_L \times s_B$ bits every time.

Then if

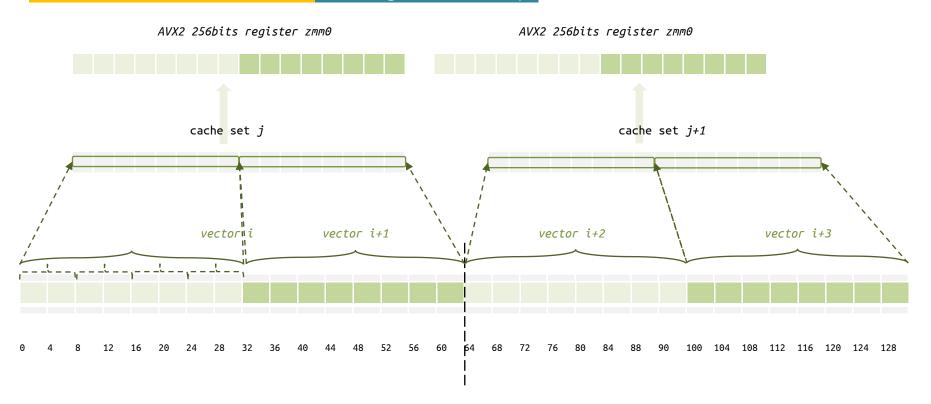
 $(a_B+V_L\times s_B)$ % $C_L==0$ (C_L being the cache capacity in terms of elements) all the vectors will stay in the same cache line and will require 1 load; otherwise, a vector will be mapped into two different cache lines and will require 2 loads.

Let's examine two examples in the following slides.



What could be inefficient? Unaligned memory

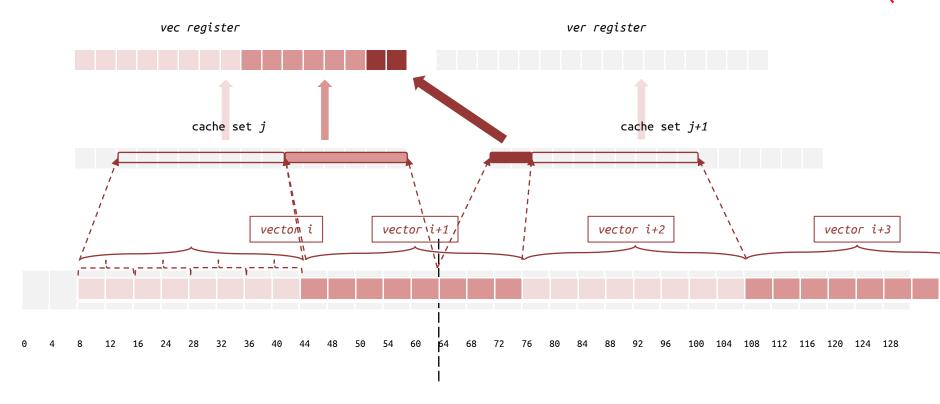
an array of float aligned at **64bits** accessed as vector of size 4 (cache capacity is 16 with a std cache line of 64B) $a_B = 8B$, $V_L = 4$, $s_B = 4$, $C_L = 16$ every vector will occupy exactly 1 cache line





What could be inefficient? Unaligned memory

an array of float aligned at **16bits** accessed as vector of size 4 (cache capacity is 16 with a std cache line of 64B) $a_B = 8B$, $V_L = 4$, $s_B = 4$, $C_L = 16$





What could be inefficient? Data structures

- Keep together data that are used together in vector operations (see slides "SoA vs AoS")
- The smallest the data type you use, the larger the number of data that you can fit in cache and in the vector registers
- Do not mix same-type different-size data

```
func ( float *A, float *B, double *C, int N) {
   for ( int i = 0; i < N; i++ )
        A[i] = B[i]*C[i]; }</pre>
```

the type conversion renders things less efficient. long double type is NOT supported on GPUs.



What could be inefficient? SoA over AoS

Very commonly the data are multi-dimensional and hence the most natural representation is by a structure that encapsulates all the quantities for a single data point.

Every entry of the array that contains all the data points is then a structure:

```
data[i].pos[3], data[i].acceleration[3], data[i].mass, data[i].composition[n_elements], ...
```

That is called an "Array of Structures" (AoS) and is a very convenient representation in many respects. For instance, when you perform a domain decomposition it is very immediate to send and receive data points.

However, when the structures collects many and diverse data, the SoA have several inconvenience too when computational performance is considered.



What could be inefficient? SoA over AoS

To understand why, consider the example of this structures that occupies 96bytes.

```
struct Particle {
    double pos[3], accel[3], vel[3];
    double mass;
    long long int ID;
    float composition[n_el]; }
```

As a first consideration, even a single particle could not fit in a cache line. Second, most probably pos[3] will be used, likely with mass, to get accel[3] and then vel[3] will be update.



What could be inefficient? SoA over AoS

Very likely every of these calculation may have a high degree of vectorization. However, there is no way in which those quantities can be loaded from memory to vector register efficiently without lots of overhead, either using gathering instructions or allocating additional memory and moving there only the variables used for every calculation.

So, a very first step consist in opting for $\bf S$ tructures of $\bf A$ rrays of (small) $\bf S$ tructures ($\bf SoAoS$) :

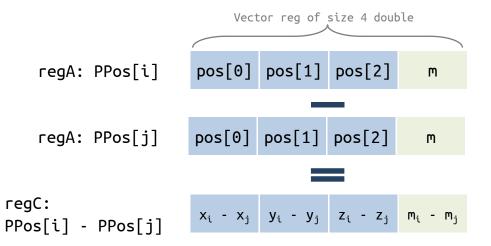


What could be inefficient? SoA over AoS

```
struct Ppos { double pos[3], double mass; };
struct Paccel { double accel[3]; };
struct Pvel { vel[3]; }
struct Pchemistry { float elements[n_el]; };
```

```
PPos *mpositions;
Paccel *accelerations;
Pvel *velocities;
Pelems *Pchemistry;
long long int *PIds;
```

Clearly that alleviates the aforementioned issues but it is still sub-optimal. For instance, considering a force computation between 2 particles, we would have



What could be inefficient? SoA over AoS

A vector subtraction returns in **regC** the 1d distance in every dimension.

However, its fourth entry would contains a to-be-discarded value (the mass difference).

Vector reg of size 4 double

After that, we need to take the square invert of regC and to perform an horizontal addition over it to get $1/r^2$.

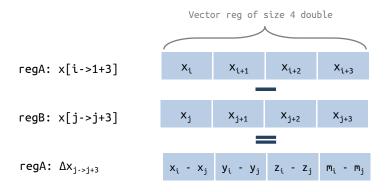
Eventually, we'll need to multiply $regA \times regB$ to get $m_i \times m_j$ (this time wasting 3 out of 4 entries), and to multiply the result by the scalar value of $1/r^2$ to have the force between two particles.

What could be inefficient? SoA over AoS

If, instead, we opt for a Structure of Arrays (SoA) approach:

```
*x, *y, z*, *mass, *xacc, *yacc, *zacc, *xvel, *yvel, *zvel;
double
long long *ids;
```

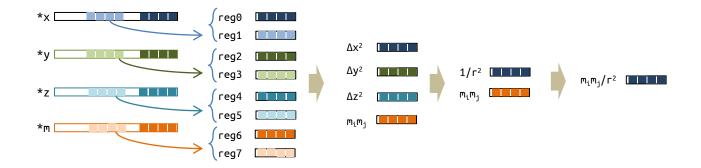
the possible vectorization is definitely higher:



With 3 vec **sub** we'll get Δx , Δy , Δz among four particles; four additional vec **mul** would convey Δx^2 , Δy^2 , Δz^2 and $[m_i m_i, m_{i+1} m_{i+1}, \dots]$.

A final vec **mul** will convey the force beween four particles.

What could be inefficient? SoA over AoS



Data rearranged in **SoA** approach definitely would expose more, and more efficient, vectorization opportunities.

That said, however, what is really convenient in every specific case may depend on several additional factors and that, in the context of this lecture, hinders more than just mentioning and explaining the vectorization advantage of SoA.



[> Challenges <]

- 1. write a code to generate the **Mandelbrot set** for a given region of the complex plane, and compare the naive version with a vectorized one.
- 2. write a code for a **convolution kernel** (for instance, to blur an image), and compare the naive version and the vectorized one
- 3. write a very simple **N-body code** that performs the estimate of the $1/r^2$ force among N_p particles, updates their accelerations and their velocity in a timestep Δt .
 - As above, try to vectorize the code and compare the relative performance.



Vectorization by vector types

- How to define a vector type
- Memory alignment
- In practice
- Unrolling by hands
- Conditional evaluations



Vector types in C/C++

De facto standard to declare vector types in C / C++ (*)

```
typedef type name __attribute__ ((vector_size ( bytes ) ));
          type
                      the basic type for every element of the vector
                      the name with which you will refer to the type in your code
           name
           bytes
                      the vector width in bytes
vector of 4 double
         typedef double v4d __attribute__ ((vector_size ( 4*sizeof(double) ) ));
vector of 8 double
         typedef double v8d attribute ((vector size ( 8*sizeof(double) ) ));
vector of 4 integers
         typedef int v4i attribute ((vector size ( 4*sizeof(int) ) ));
```

^(*) There are third-party libraries that provide interfaces to low-level types and intrinsics, especially for C++. Check Highway, EVE, xsimd, VCL (Vector Class Library)



Vector types in C/C++

Let's say that we declare a vector of 4 double

```
typedef double v4d __attribute__ ((vector_size ( 4*sizeof(double) ) ));
```

Is it possible to access singularly each one of the double in the vector? No, exactly as you can not access the single bytes in a double. However, there a standard way to do that:

```
typedef union {
  v4d V;
  double v[VD_SIZE];
}v4d_u;
```



Vector types in C/C++

[> Challenge <] ..well, not that big

Declare a vector of 4 double

```
typedef double v4d __attribute__ ((vector_size ( 4*sizeof(double) ) ));
and by using

typedef union {
   v4d  V;
   double v[VD_SIZE];
   }v4d_u;
```

find the order of elements in the vector: is the first double placed at the lowest or at the highest memory address?



Vector types :: memory alignment

- It is mandatory that memory regions accessed by vector instructions are aligned (see this slide)
 32B for AVX/AVX2 and 64B for AVX512 (on x86 64 architecture)
- Loops are translated by the compiler:
 - ► Peeling scalar until aligned addresses are reached
 - ► Main body well vectorized from aligned addresses
 - ► Remainder scalar ops if the number of elements is not a multiple of the VL

Vector types :: memory alignment

- It is mandatory that memory regions accessed by vector instructions are aligned.
- Loops translated by the compiler consist of three sections:
 - ► Peeling loop scalar until aligned addresses are reached
 - → avoided by proper alignment
 - ► Main body well vectorized from aligned addresses
 - ► Remainder loop scalar ops if the number of elements is not a multiple of the VL
 - → (maybe) avoided by hinting the compiler e.g. __attribute((__assume(N%4==0))) or by accounting for the main body and the remainder explicitly



Vector types :: memory alignment

Controlling the alignment

- Allocated memory
 - ► C11 void * aligned_alloc (size_t alignment, size_t size)
 - ▶ POSIX int posix_memalign (void **memptr, size_t alignment, size_t size)
- Static allocation (i.e. variables or automatic arrays)
 _attribute__ ((aligned(base))) <var>
- Assuming alignment
 __assume_aligned(<array>, base)



Vector types :: in practice

[> Challenge <]

Re-implement the following loop using the vector types and check what changes in the generated assembler code and in the run-time

Unrolling loops by hands



When unrolling a loop by hands, which you tipically do in order to expose more parallelism as we have discussed in the previous slides, you also need to separately handle the main loop and the remind loop.

```
for (int i = 0; i < N; i++ ) ...
```

becomes

```
int N4 = (N/4)*4; --> integer division ensures that N4 is the largets multiple of 4 that is smaller than N
```

```
for ( int i = 0; i < N4; i+=4 ) ... main body
```



Unrolling loops by hands



You can give some additional hint to the compiler so that to avoid the issuing of additional codes to check the peeling

```
The line

int N4 = (N/4)*4;

→ Vectorization/3_fma_loop/hint_compiler<..>.c
```

can be rewritten as

this will (may, on some compilers) hint the compiler that N4 is a multiple of 4, without need of issuing additional code.

Alternatively,

```
N4 __attribute((__assume(N4 % 4 == 0)));
may also work (on some compilers)
```

Vector types :: in practice

[> Challenge <]

check Vectorization/3_fma_loop/test.c CASE=2

1) define the appropriate vector variables

```
#define VD SIZE 4
 typedef double v4df attribute ((vector size (VD SIZE*sizeof(double))));
 typedef union
    v4df V:
                                      vA[i] will represent as a vector the first 4 doubles
    double v[VD SIZE];
                                      in the array A, vA[i+1] the second quadruplet and so on.
  } v4df u;
                                      The same for vB and B, and vC and C.
v4df *vA = (v4df*)A; ///declare vA as a pointer to v4df, and make it pointing to A
v4df *vB = (v4df*)B; \frac{1}{2} declare vB as a pointer to v4df, and make it pointing to B
v4df *vC = (v4df*)C; // declare vC as a pointer to v4df, and make it pointing to C
v4df vsum = \{0\}; // declare the stack variable vsum, and initialize it to zero
```

Vector types :: in practice

[> Challenge <]

check Vectorization/3_fma_loop/test.c CASE=2

2) Re-define the iteration space

```
int N4 = N&OxFFFFFFFC;
                                                   The union of vector and array
for ( int i = 0 ; i < N4; i++ )
                                   // main body
                                                   is to be used exactly in
  vsum += vA[i] * vB[i] + vC[i];
                                                   these cases
v4df u *vsum u = (v4df u*)&vsum; // summation of the 4 entries of vsum
vsum u \rightarrow v[0] += vsum u \rightarrow v[1] + // vsum[0] will contain the scalar sum
              (vsum u->v[2] + vsum u->v[3]);
sum += A[i]*B[i] + C[i]:
                                   // sum up the final elements
sum += vsum u->v[0];
```

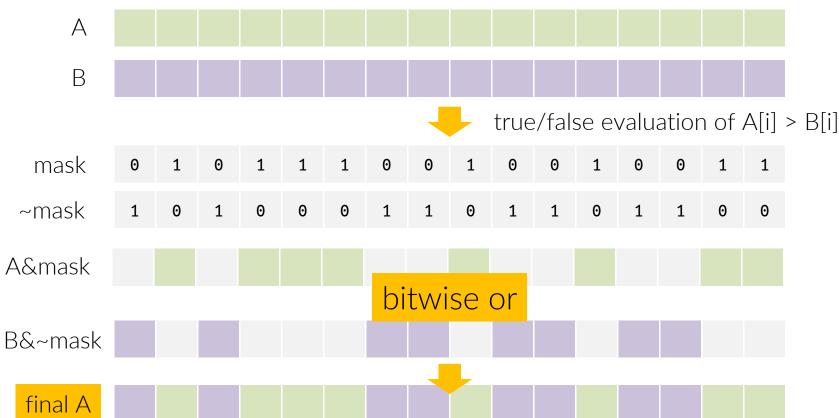
Vector types :: conditional evaluation

Although life is a long sequence of IF-THIS-THEN-THAT and we stated at the begin of these series of lectures that the conditionals hinder the vectorization, or at least render it much more difficult.

The only "easily" vectorizable if-conditions are those solvable as "masquerable" events

```
void kernel( double * restrict A, double * restrict B, const int N )
   // element-wise swap of A and B so that
   // A[i] > B[i] for every i
   {
     for ( int i = 0; i < N; i++ )
        if ( A[i] < B[i] ) swap(A, B);
     return;
}</pre>
```

Vector types :: conditional → masking





Vector types :: conditional evaluation → masking

The concept of vectorized conditional flow can really be thought as "projecting" through a mask so that the final values that are "impressed" in the memory positions pass through the "transparent" entries and are blocked by "opaque" ones.

Many "projected" values could of course overlap by using multiple masks combines with bit-wise operators.



Actually this code can be easily fully vectorized by the compiler itself with the appropriate compilation flags.

We use it as a testbed to experiment how the conditionals could be expressed using vector types.

```
void kernel( double *A, double *B, int N )
   // element-wise swap of A and B so that
      A[i] > B[i] for every i
       for ( int i = 0; i < N; i++ ) {
         double min = (A[i]>B[i] ? B[i] : A[i]);
         double max = (A[i] >= B[i] ? A[i] : B[i]);
         A[i] = max;
        B[i] = min; }
     return;
```

ightarrow Vectorization/3_conditionals/vector_conditionals.a.c



```
void kernel( double *restrict A, double *restrict B, const int N )
                                                      dvector t *vA = (dvector t *) builtin assume aligned(A, VALIGN);
for ( int i = 0; i < N; i++ )
                                                      dvector t *vB = (dvector t *) builtin assume aligned(B, VALIGN);
    double min = (A[i]>B[i] ? B[i] : A[i]);
                                                      int VN = (N/DVSIZE)&0xFFFFFFC;
    double max = (A[i] >= B[i] ? A[i] : B[i]);
                                                      IVDEP
    A[i] = max;
                                                      LOOP VECTORIZE
    B[i] = min;
                                                      LOOP UNROLL N(4)
                                                      for ( int i = 0; i < VN; i++ )
                                                           dvector_t a = vA[i];
                                                           dvector_t b = vB[i];
                                                           llvector t keep = (vA[i]>=vB[i]);
                                                           vA[i] = (dvector_t)(((llvector_t)a & keep) |
                                                                               ((llvector_t)b & ~keep));
                                                           vB[i] = (dvector_t)(((llvector_t)b & keep) |
                                                                               ((llvector t)a & ~keep));
```

plain version

version A

int j = VN*DVSIZE;

process (&A[j], &B[j], N-j+1);

```
void kernel( double *restrict A, double *restrict B, const int N )
                    cast to pointers to vector type
                        assume memory alignment
                                                                dvector t *vA = (dvector t *) builtin assume aligned(A, VALIGN);
       (note: the arrays must have been allocated
                                                                dvector t *vB = (dvector t *) builtin assume aligned(B, VALIGN);
                                             aligned)
                                                               int VN = (N/DVSIZE)&0xFFFFFFFC;
remap the iteration space & signal that no peeling
is needed
                                                                IVDEP
                                                                LOOP VECTORIZE
                 unroll & vectorization hints / requests
                                                               LOOP UNROLL N(4)
                                                                                                                  build the mask as a vector of
                                                               for ( int i = 0; i < VN; i++ )
                                                                                                                  integers - the bitfield must have
                                                                                                                  the same than the data you will
                                                                                      = vA[i];
                                                                     dvector t a
make it local - i.e. suggest that data should be resident in
                                                                                                                  mask
                                                                                      = vB[i]:
                                                                     dvector t b
registers
                                                                     llvector t keep = (vA[i]>=vB[i]);
final masking: (a & keep) will contain a's entries where
mask==1 and 0 elsewhere; at the opposite, (b & ~keep)
                                                                     vA[i] = (dvector_t)(((llvector_t)a & keep) |
will contain b's entries where the mask==0.
                                                                                          ((llvector t)b & ~keep));
The | is blending the two in one.
                                                                     vB[i] = (dvector t)(((llvector t)b & keep) |
                                                                                          ((llvector t)a & ~keep));
dvector t
VALIGN
DVSIZE
                                                               int j = VN*DVSIZE;
IVDEP
                                                               process ( &A[j], &B[j], N-j+1);
LOOP VECTORIZE
LOOP UNROLL N
                                                                version A
are defined in
                 /Vector/header/* pragmas.h
```



void kernel(double *restrict A, double *restrict B, const int N) cast to pointers to vector type assume memory alignment dvector_t *vA = (dvector_t *)__builtin_assume_aligned(A, VALIGN); (note: the arrays *must* have been allocated dvector t *vB = (dvector t *) builtin assume aligned(B, VALIGN); aligned) int VN = (N/DVSIZE)&((-(int)DVSIZE); remap the iteration space & signal that no peeling TVDFP is needed LOOP VECTORIZE LOOP UNROLL N(4) unroll & vectorization hints / requests for (int i = 0; i < VN; i++) dvector u vAu, vBu; make it local using union type: we will be able to access vAu.V = vA[i]: individual elements vBu.V = vB[i];for (int j = 0; j < DVSIZE; j++) { vAu.v[j] = (vAu.v[j] >= vBu.v[j]? vAu.v[j] : vBu.v[j]);masking is done individually via the union trick $vBu.v[j] = (vAu.v[j] >= vBu.v[j]? vBu.v[j] : vAu.v[j]); }$ vA[i] = vAu.V;reassign to array's elements int i = VN*DVSIZE: Exercise < process (&A[j], &B[j], N-j+1);

version B

Inspect the assembler code and find that the issued code is basically the same in the three cases

```
void kernel( double *restrict A, double *restrict B, const int N ) \,
```

```
double sum = 0;
for ( int i = 0; i < N; i++ )
    {
      double tmp = A[i]*B[i];
      if ( tmp > 0.5 ) sum += B[i];
    }
return sum;
```

[> Exercise <]

Let's analyze that together

```
plain version
```

```
dvector_t *vA = (dvector_t *)__builtin_assume_aligned(A, VALIGN);
dvector_t *vB = (dvector_t *)__builtin_assume_aligned(B, VALIGN);
int VN = (N/DVSIZE)&0xFFFFFFC;
IVDEP
LOOP VECTORIZE
LOOP UNROLL N(4)
for ( int i = 0; i < VN; i++ )
      dvector t a = vA[i]*vB[i];
      dvector t b = vB[i];
      llvector t keep = (a > 0.5);
      vsum += (dvector_t)( (llvector_t)b & keep );
  dvector u *vsum u = (dvector u*)&vsum;
  vsum u - v[0] + vsum u - v[1] + (vsum u - v[2] + vsum u - v[3]);
  int j = VN*DVSIZE;
  double sum = 0;
  sum += process ( &A[j], &B[j], N-j+1);
  sum += vsum u \rightarrow v[0];
```

version A



Luca Tornatore

[5]

Vectorization by OpenMP SIMD directive

(Essential elements)

- How to define a vector type
- Memory alignment
- In practice
- Unrolling by hands
- Conditional evaluations



The SIMD construct

The OpenMP compiler may transform a loop that is marked with the simd construct, into a SIMD loop. As a result, multiple iterations of the loop may be executed concurrently by a single thread. A SIMD loop of length n, consists of the logical iterations $0, 1, \ldots, n-1$. The numbering denotes the sequential order in which loop iterations are executed.

A SIMD chunk refers to the set of iterations that are executed concurrently by a SIMD instruction in a single thread. Within a chunk, each iteration is executed by a SIMD lane, which refers to the mechanism that a SIMD instruction uses to process one data element. The number iterations in a SIMD chunk is the vector length.

The OpenMP specification describes the execution model of the simd construct as follows: "When an OpenMP thread encounters a simd construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread." Intentionally, this allows for much freedom in the implementation.

excerpts from "OpenMP - The Next Step, MIT Press



The omp SIMD :: syntax

```
#pragma omp simd [clause[[,] clause]. . . ]
    for-loops
list of clauses
  private (list)
  lastprivate (list)
  reduction (identifier: list)
  collapse (n)
  simdlen (length)
  safelen (length)
  linear (list[: linear-stepJ)
  aligned (list{:alignment])
```

private and **lastprivate** hasve exatly the same meaning as in "thread-related" OpenMP

The execution instances in this context are the SIMD lanes; hence, an instance of every private variables is created per SIMD lane.

```
int i;
double tmp, sum=0;

#pragma omp simd private(tmp) reduction(+:sum)
for ( int i = 0; i < N; i++ )
    {
        tmp = function( b[i] );
        array[i] = tmp;
        sum += tmp;
    }

// note: i is lastprivate</pre>
```

The omp SIMD :: syntax

```
#pragma omp simd [clause[[,] clause]. . . ]
     for-loops
                                                     redution and collapse also retain their usual
                                                     meaning and usage
list of clauses
   private (list)
                                                      int
                                                            i:
   lastprivate (list)/
                                                      double sum=0;
   reduction (identifier: list)
   collapse (n)
                                                      #pragma omp simd reduction(+:sum)
                                                       for ( int i = 0; i < N; i++ )
   simdlen (length)
                                                         for ( int j = 0; j < M; j++ )
   safelen (length)
   linear (list[: linear-stepJ)
                                                              tmp = function( a[i], b[j] );
                                                              arrav[i] = tmp;
   aligned (list{:alignment])
                                                              sum += tmp;
```

The omp SIMD :: simdlen clause

```
#pragma omp simd [clause[[,] clause]. . . ]
for-loops
```

list of clauses

```
private (list)
lastprivate (list)
reduction (identifier: list)
collapse (n)
simdlen (length)
safelen (length)
linear (list[: linear-stepJ)
aligned (list{:alignment])
```

The **simdlen** clause has the goal of suggesting the element-size of the vector lane to be adopted. Let the count of elements in the vector.

```
int i;
double tmp;

#pragma omp simd simdlen(32/sizeof(double))
for ( int i = 0; i < N; i++ )
    {
        tmp = function( b[i] );
        array[i] = tmp;
    }

// note: i is lastprivate</pre>
```

from "OpenMP - The Next Step, MIT Press



The omp SIMD :: safelen clause

```
#pragma omp simd [clause[[,] clause]. . . ]
for-loops
```

list of clauses

```
private (list)
lastprivate (list)
reduction (identifier: list)
collapse (n)
simdlen (length)
safelen (length)
linear (list[: linear-stepJ)
aligned (list{:alignment])
```

The **safelen** clause specifies the loop-carried dependence distance: the positive integer argument provides an upper limit to the length within which it is safe to vectorize the loop.

In the following example, we instruct the compiler that 8 is a safe length for vectorization (note that there are no other hints about the value of k in the code itself)

```
#pragma omp simd simdlen(8)
for ( int i = k; i < N; i++ )
a[i] = pow(b[i-k], 3.14);</pre>
```

inspired from "OpenMP - The Next Step, MIT Press



In general, every non-linear element in the execution flow is at high risk a disruption in the vectorization. Function call above all, but also, for instance, conditionals.

OpenMP SIMD offers the possibility of automatically build vector version of code segments through the **declare simd** directive (see fig. aside for the concept).

These function can then be called from a simd loop.

```
for (int i=0; i<8; i++)
a[i] = f(b[i]);

a[0] = f(b[0])
a[1] = f(b[1])

a[0] a[1] a[2] a[3] = F(b[0] b[1] b[2] b[3])

a[3] = f(b[3])

a[4] = f(b[4])

a[5] = f(b[6])

a[6] = f(b[6])

a[7] = f(b[7])
```

from "OpenMP - The Next Step, MIT Press

#pragma omp declare simd [clause[[,] clause..]
 function declaration



```
#pragma omp declare simd
double foo( double x, double y, double z)
{
    double res = x*y +z;
    return res;
}

void loop( double *a, double*b, double *c, int N)
{
    #pragma omp simd
    for ( int i = 0; i < n; i+=4 )
        a[i] = foo( a[i], b[i], c[i] );
}</pre>
```

The declare simd directive instructs the compiler to generate at least one additional simd version or the function foo because it will be called from a simd loop.

That variant will be called from inside the simd loop in the function **loop**.

However, when declaring foo, can we give to the compiler more details about how x, y and z are changing? In this form, the best assumption is that they must be incremented by 1 in the simd version

uniform, linear, simdlen, and aligned clauses convey additional informations about how to generate the simd version of a function.

uniform when listed as uniform, a parameter is intended to be invariant for all the concurrent calls to the function, i.e. all the simd lanes will observe the same value

linear at odds, being linear means that a parameter changes linaerly with the indicated step

simdlen retains the same meaning

aligned has exactly the meaning than in normal code

simdlen (length)
linear (list[:linear-step])
aligned (list[:alignment])
uniform (argument-list)
inbranch
notinbranch



```
#pragma omp declare simd uniform(x, y, d1, i, a) linear(j)
void saxpy_2d(float *x, float *y, float a, int d1, int i, int j)
 y[(d1*i)+j] = a*x[(d1*i)+j] + y[(d1*i)+j];
```

```
#pragma omp declare simd simdlen(16)
char F(char x, char y, unsigned char mask)
  return (x + y) & mask;
void img_mask(char *img1, char *img2, int n, unsigned char *m)
 #pragma omp simd simdlen(16)
 for (int i=0; i<n; i++) {
   img1[i] = F(img1[i], img2[i], m[i]);
 } // End simd region
```

In this example, the compiler is informed that x and y must be kept constant (i.e. the starting points of the arrays do not change).

As well, the scalar variables a, d1, i and a will be constant.

Instead, j, that is the indexing variable, will be increased +1 for every lane.

In the second example, x, y and char are considered to be linear with step 1, and the adopted size for the vector is 16

```
The third example illustrates the usage of aligned, which instructs the compiler
about the expected alignment of data arrays
```

```
#pragma omp declare simd linear(src,dst) \
                    aligned(src,dst:16) simdlen(32)
void copy32x8(char *dst, char *src)
   *dst = *src;
#pragma omp declare simd uniform(x,y) linear(i) \
                         aligned(x,y:64) simdlen(16)
float saxpy(float a, float *x, float *y, int i)
 return a * ((x[i]) + (y[i]));
```

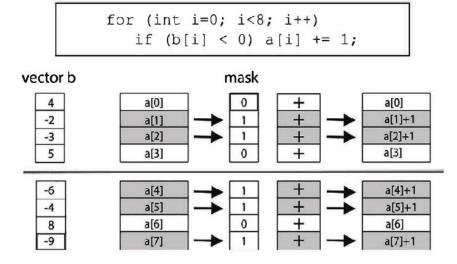


The omp SIMD :: conditionals

OpenMP simd offers two additional clauses that work as assertions: inbranch and notinbranch.

inbranch informs the compiler that the function will be called from within a conditional branch, whereas notinbranch garantees the compiler that this will not happen.

The compiler can then opt for very different optimizations and how to manage the vectorization.



from "OpenMP - The Next Step, MIT Press

The omp SIMD :: conditionals

OpenMP simd offers two additional clauses that work as assertions: inbranch and notinbranch.

inbranch informs the compiler that the function will be called from within a conditional branch, whereas notinbranch garantees the compiler that this will not happen.

The compiler can then opt for very different optimizations and how to manage the vectorization.

```
#pragma omp declare simd inbranch
float do_mult(float x)
 return (-2.0*x);
#pragma omp declare simd notinbranch
extern float do_pow(float);
void simd_loop_with_branch(float *a, float *b, int n)
  #pragma omp simd
  for (int i=0; i<n; i++) {
    if (a[i] < 0.0)
     b[i] = do_mult(a[i]);
   b[i] = do_pow(b[i]);
 } /* --- end simd region --- */
```

from "OpenMP - The Next Step, MIT Press



The omp SIMD :: conditionals

Finally, multiple simd versions of the same function can be generated with multiple declarations:

```
#pragma omp declare simd linear(pixel) uniform(mask) inbranch
#pragma omp declare simd linear(pixel) notin branch
#pragma omp declare simd
extern void compute_pixel(char *pixel, char mask);
```

from "OpenMP - The Next Step, MIT Press

Addressing modes in x64 assembly



TBD

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

That's all folks, have fun

