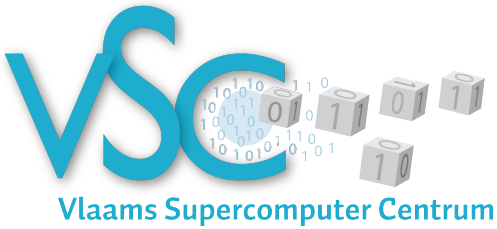
HPC TNT - 2

Tips and tricks for
Vectorization approaches to efficient code

HPC core facility CalcUA



ANNIE CUYT ☐ STEFAN BECUWE ☐ FRANKY BACKELJAUW ☐ [ENGEL]BERT TIJSKENS

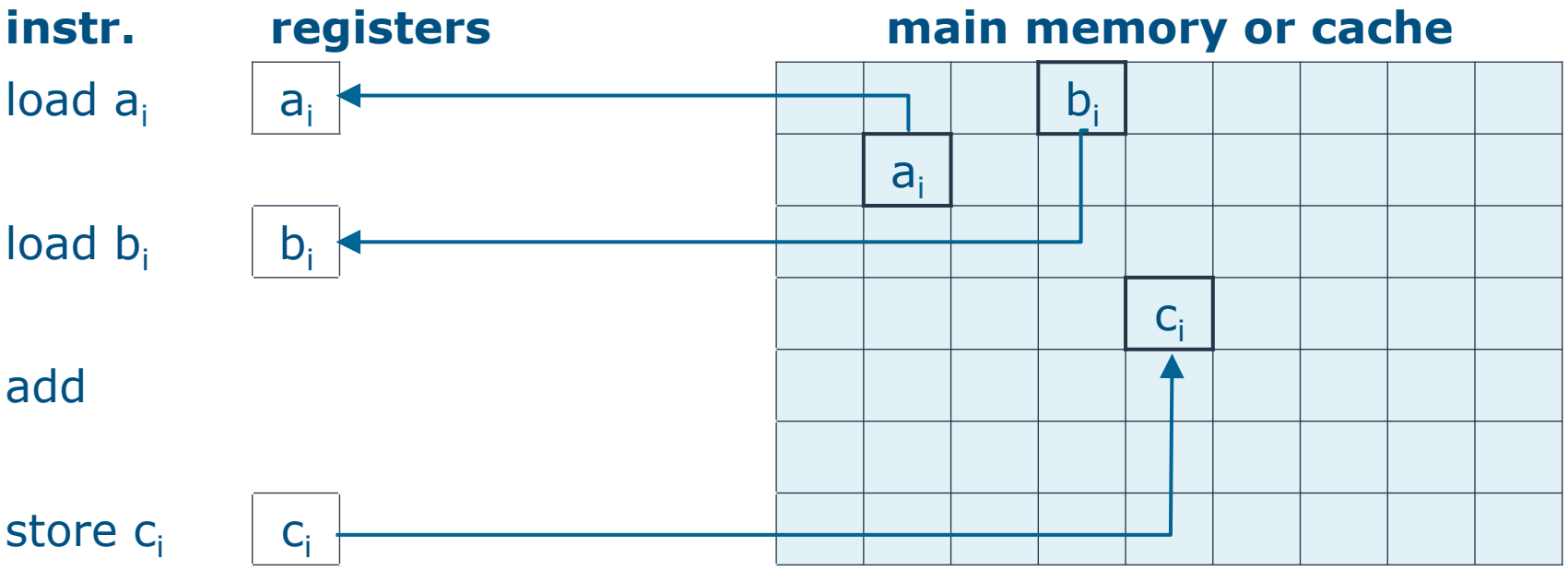
Overview

- Introduction
  - What is vectorization
  - Why does it matter
  - When does it not matter
- Different approaches
- Case study – magnetization
- Final remarks

- Focus of this talk is on big picture and background not on the details

What is vectorization

```
do i=1,n
   c(i) = a(i)+b(i)
end do
```

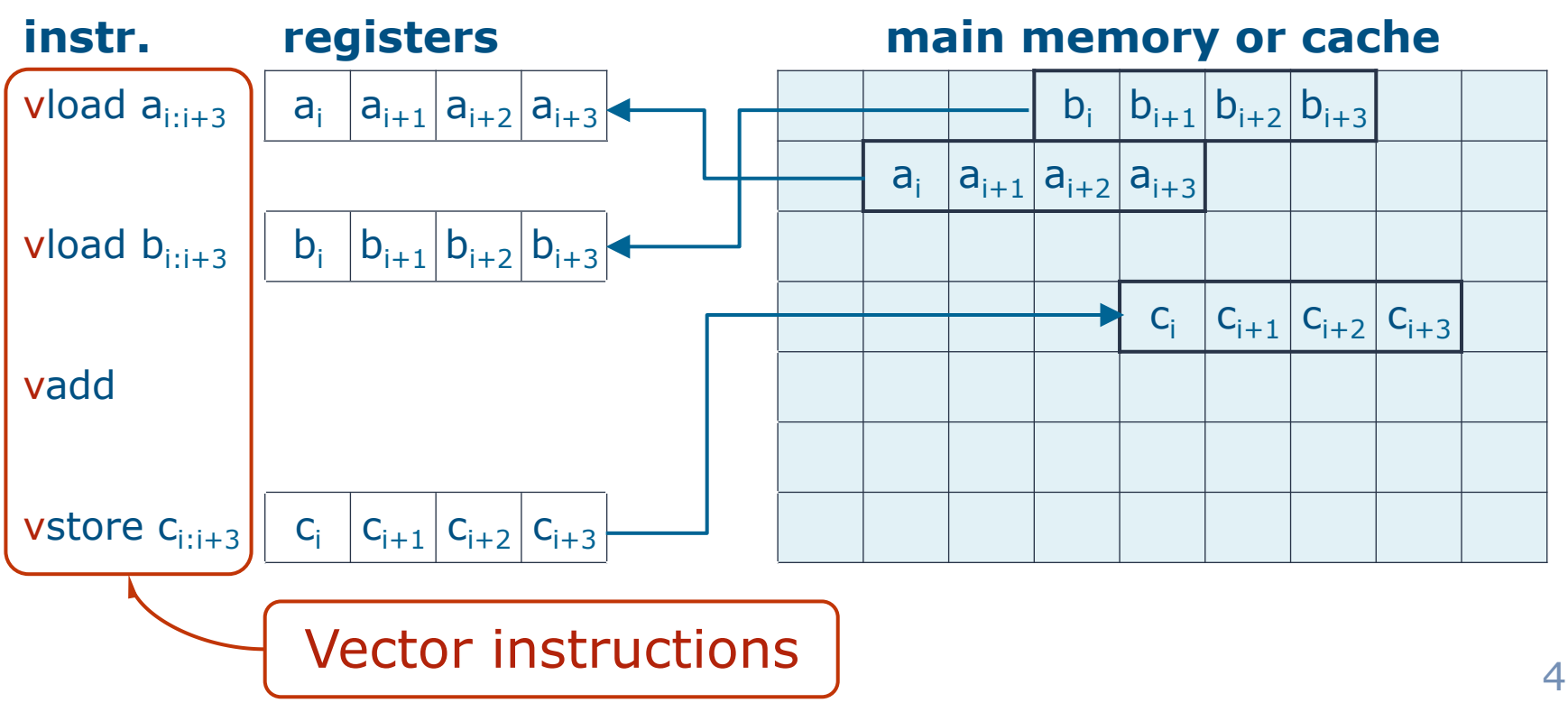


What is vectorization

```
do i=1,n,4
   c(i:i+3) = a(i:i+3)+b(i:i+3)
end do
```

Vector width

SIMD
Single Instruction
Multiple data



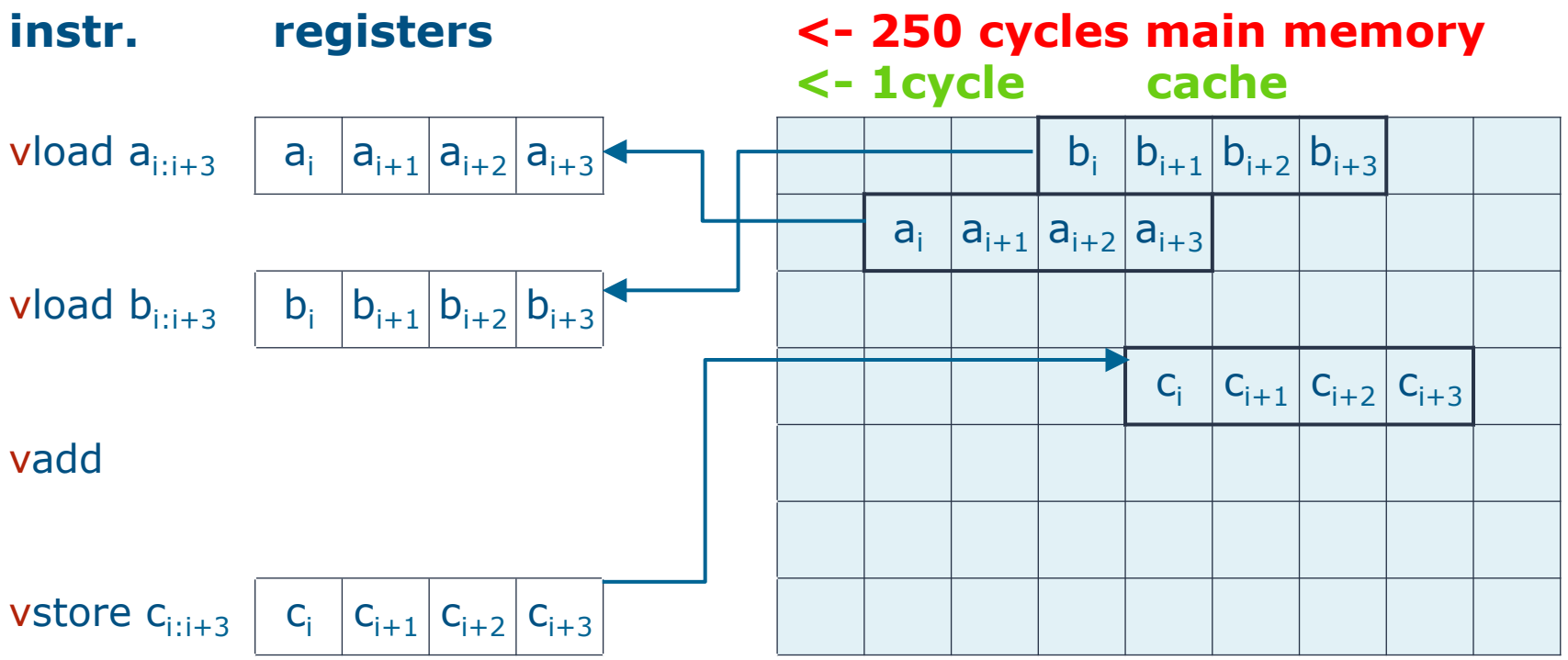
Vector width

Cpu	Instruction set	Vector width	Vector width SP	Vector width DP	
Westmere	SSE4.2	128 bit	4	2	
Ivy bridge	AVX	256 bit	8	4	**Hopper**
Haswell	AVX2	256 bit	8	4	
Xeon Phi	AVX-512	512 bit	16	8	

Failing to vectorize implies that your application runs **at best** at no more than 25% (12.5%) of peak performance for DP (SP)

when not to focus on vectorization (yet)

```
do i=1,n,4
   c(i:i+3) = a(i:i+3)+b(i:i+3)
end do
```



When not to focus on vectorization (yet)

- Application with slow data traffic: every item needs to be fetched from main memory or from another node
  - Reorganize data layout
  - Sort data in the order the application needs them

- use PerfExpert on Hopper to find out about data traffic problems

Organize data

State of all persons

List of ill people

List of healthy people

```
do i=1,n_ill:
   do j=1,n_healthy
      if contaminate(persons(ill(i)),persons(healthy(j)))
         move_to_ill(healthy(j))
   end do
end do
```

State of ill people

State of healthy people

```
do i=1,n_ill:
   do j=1,n_healthy
      if contaminate(ill_persons(i),healthy_persons(j))
         move_to_ill(healthy_persons(j))
   end do
end do
```

When not to focus on vectorization (yet)

- Application saturates bandwidth
  - The bottleneck is not computation
  - Try to do all possible work on data when it is in cache (blocking/tiling)
  - If there is not much to compute vectorization will not help

- use PerfExpert on Hopper to find out

Blocking/tiling

```
// inefficient
// except small problems

For all items
   Do this
For all items
   Do that
For all items
   Do something else


// data transferred 3x
```

```
// efficient
// chunk = collection of items
//       that fits in L1 cache
For all chunks
   For all items in chunk
      Do this
   For all items in chunk
      Do that
   For all items in chunk
      Do something else


// data transferred once
```

Approaches to vectorization

- Assembly code
- Intrinsics
- Auto-vectorization by compiler
  - GCC
  - Intel compiler suite
- low level library
  - boost.simd
  - Vc
  - Intel Cilk++
- optimized high level library
  - Eigen
  - Intel MKL

Approaches to vectorization
1. Assembly

- Write assembly code that use SIMD instructions (SSE, AVX, …)

```
for (int i = 0; i<N; ++i) C[i] = A[i] + B[i];
```

V=AVX, mov=move, u=unaligned, p=packed, s=single precision

```
NEXT_ITERATION:
vmovups ymm1, ymmword ptr [edi+edx*4+8] ; copy 32 bytes from B to ymm1
vmovups ymm0, ymmword ptr [esi+edx*4+8] ; copy 32 bytes from B to ymm1
vaddps ymm1, ymm1, ymm0; add ymm0 to ymm1 and store the result in ymm1
vmovups ymmword ptr [ecx+edx*4+8], ymm1; copy 32 bytes from ymm1 to C
add edx, 8 ; increase loop index by 8
cmp edx, dword ptr [esi+4]
jg NEXT_ITERATION
```

Approaches to vectorization
2. Intrinsics

- Layer of data types and functions made available by compiler through include file (C/C++)

```
#include <immintrin.h>
...
__m256 ymm0, ymm1; //define the vector registers used
float a[8]={1,2,3,4,5,6,7,8};
float b[8]={2,3,4,5,6,7,8,9};
float c[8];
ymm0 = __builtin_ia32_loadups256(a); // ymm0 <= a
ymm1 = __builtin_ia32_loadups256(b); // ymm1 <= b
ymm0 = __builtin_ia32_mulps256(ymm0, ymm1); // ymm0 <= ymm0*ymm1
__builtin_ia32_storeups256(c, ymm0); // c <= ymm0
...
```

Approaches to vectorization
assembly/intrinsics

	Pro	Con
assembly	• Can be very efficient	• Hard, tedious, error-prone • Depend on hardware • Low level
intrinsics	• Can be very efficient • Stay in the language	• Only slightly less hard • Depend on hardware and on compiler • Only C/C++

Approaches to vectorization
3. Auto-vectorization

- the compiler is your friend - let the compiler handle it
- number of constraints
- you can help the compiler
  - Compiler directives
  - Language extensions (`restrict`)

Pro

- usually efficient
- relatively easy

Con

- depend (only) on compiler

Approaches to vectorization
auto-vectorization

- Constraints
  - inner loop only
  - loop count must be known in advance
  - single entry, single exit (no `break`)
  - no I/O
  - no branches (but you can use masks)
  - only intrinsic math functions and
    - C/C++ inlined functions, with no side effects
    - Fortran elemental functions
  - each iteration must be independent
  - `a(i)=a(i-4)+b(i)` is ok with vector length 2, **not** 4

Approaches to vectorization
auto-vectorization

- If the compiler cannot unambiguously decide that there is no dependency, the loop is not vectorized

  Mind pointers in C/C++/Fortran!

```
double sum(double* a, double* b, double* c, int N) {
  double ab=0;
  for( int i=0; i<N; ++i ) c[i]=a[i]+b[i-2]
}
```

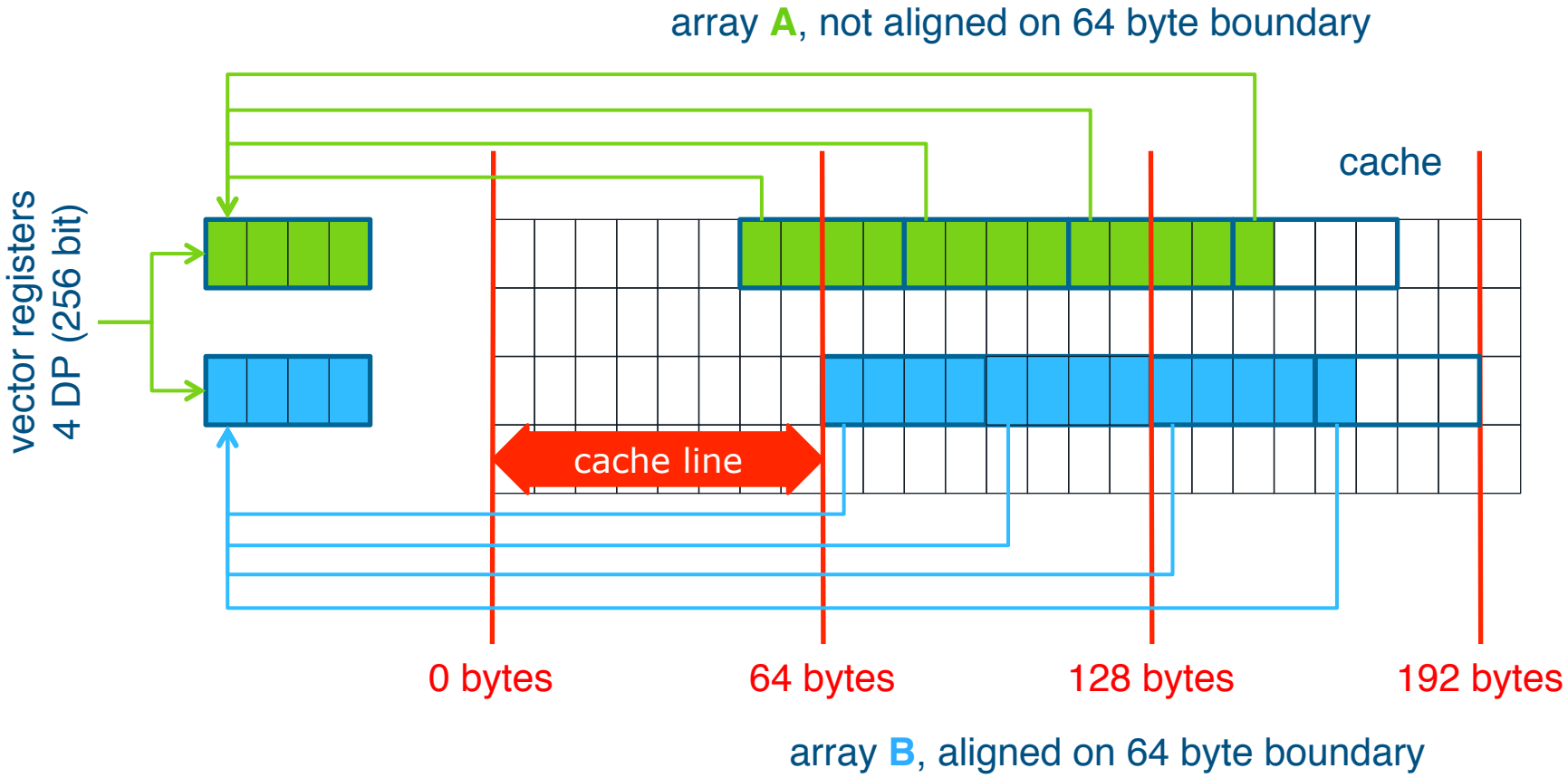
  is not vectorized:

  c may be an alias for b -> **assumed dependency**

```
double sum(double* restrict a, double* restrict b,
           double* restrict c, int N) {
  double ab=0;
    for( int i=0; i<N; ++i ) c[i]=a[i]+b[i-2]
}
```

- Fortran has arrays, C/C++ does not: `double a[]` is pointer too!

alignment issues - 1



- alignment of arrays on 64 byte boundary can have important performance effect
- increased pressure on prefetcher (every second vector comes from 2 cache lines

Universiteit Antwerpen

Vlaams Supercomputer Centrum

alignment of static arrays

- linux C/C++

```
float A[1000] __attribute__((aligned(64)));
```

- Fortran

```
real :: A(1000)
!dir$ attributes align: 64:: A
```

alignment of dynamic/allocatable arrays

- linux C/C++ (unfortunately not very elegant)
  - operator new cannot handle alignment
  - replace `malloc` and `free` with `_mm_malloc` and `_mm_free` [intel compiler only]
  - Use of `std::_aligned_malloc` and placement `new`
  - http://cottonvibes.blogspot.be/2011/01/dynamically-allocate-aligned-memory.html
  - Intel TBB provides a **portable** `cache_aligned_allocator`

- still must inform the compiler on alignment

```
void myfun( double *a, int n) {
   __assume_aligned(a, 64);
   for( int j=0; j<n; ++j )
     ++a[j];
}
```

alignment of dynamic/allocatable arrays

- Fortran

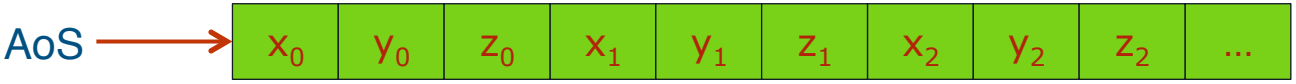
```
real, allocatable :: a(:)
!dir$ attributes align:64 :: a
```

- still must inform the compiler on alignment

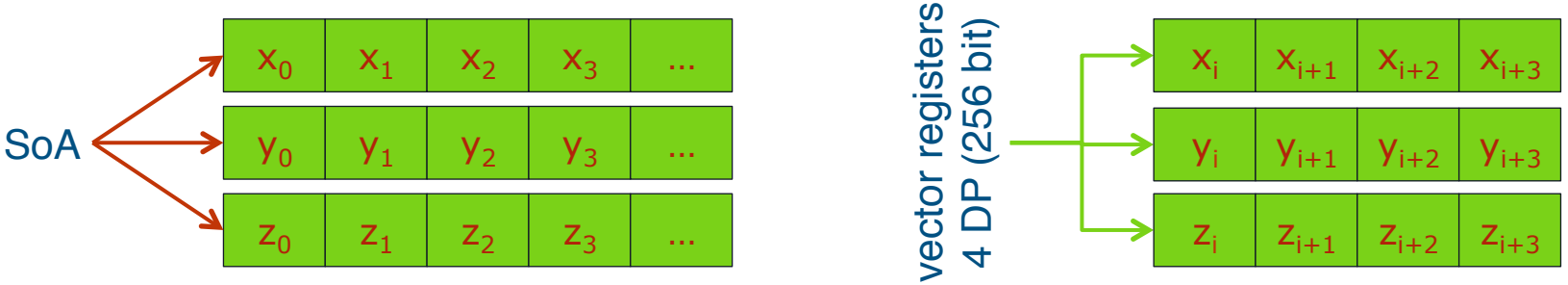
```
!DIR$ ASSUME_ALIGNED A: 64
...
do i=1, N
  A(I) = A(I) + 1
end do
```

alignment issues - 2

- compute norm of vector : $sqrt(x_i^2+y_i^2+z_i^2)$
- AoS: a single array containing (x,y,z) triplets



- SoA: separate array for x, y and z values



- SoA: vector registers can be filled with contiguous memory = as efficient as it can get
- AoS: vector registers must be filled by gathering noncontiguous elements

Approaches to vectorization
auto-vectorization

- Compiler can provide vectorization report
- Intel <15.0: -vec-report[=n]
  - n=0 no report
  - n=1 report vectorized loops
  - n=2 report vectorized and non-vectorized loops
  - n=3 2+report proven or assumed dependencies
  - n=4 report non-vectorized loops
  - n=5 4+reason why not
- Intel >=15.0: -qopt-report[=n]
  - Generate optimisation report
  - n=0..5 increasing detail
- Use report to understand and improve auto-vectorization

Approaches to vectorization
auto-vectorization

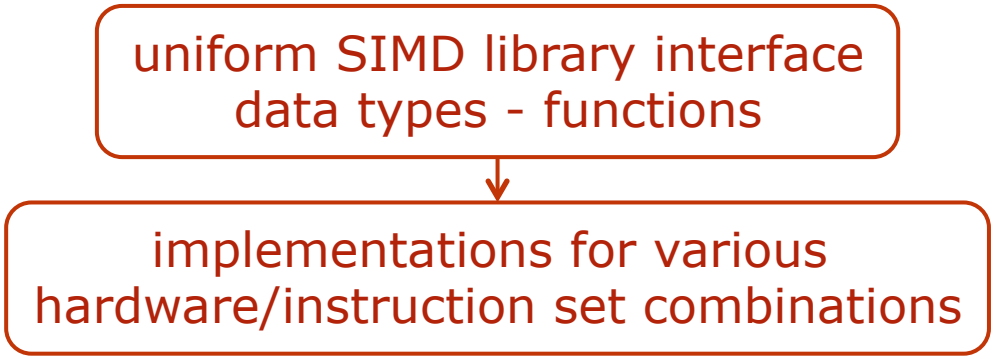
- Use compiler directives to help the compiler
  - Fortran !DEC$ *directive*
  - C/C++ #pragma *directive*
  - `ivdep` Instructs the compiler to ignore assumed vector dependencies (but not proven ones)
  - `vector {aligned|unaligned|always|temporal|nontemporal}` Specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored. Using the assert keyword with the vector {always} pragma generates an error-level assertion message saying that the compiler efficiency heuristics indicate that the loop cannot be vectorized. Use #pragma ivdep! to ignore the assumed dependencies.
  - `novector` Specifies that the loop should never be vectorized.
  - `simd` Enforces vectorization of loops.

Approaches to vectorization
auto-vectorization

- Use compiler directives with care!
  - Assert that ignoring assumed dependencies are actually non happening
- Compiler flags to force these directives application wide Use with even greater care!
- GCC has auto-vectorization, but Intel compiler should be your first choice

Approaches to vectorization
4. Low level libraries

- "architecture independent intrinsics"



- The compiler translates the generic data types and operator/function calls into assembly instructions

Approaches to vectorization
4. Low level libraries

- code can be compiled to
  - run even in the absence of SIMD hardware
  - select most efficient implementation available automatically at runtime!
- handle alignment requirements
- ideal for implementation of high level libraries
- boost.simd is component of NT^2 Numerical Template Toolbox (not part of boost (yet?)):
  - github.com/MetaScale/nt2
- Vc is found at gitorious.org/vc
- Intel Cilk++

Approaches to vectorization
low level libraries

Pro

- Efficient
- Easy
- Flexible

Con

- Depend on library
  (Boost.simd and Vc are interesting but small initiatives, support may stop at some point)
- Only C/C++
  (afaik)
- Low level
  (but higher than intrinsics)

Approaches to vectorization
5. high level libraries

- Formulate your problem in terms of library calls
- Trust that library designers have done good job
- E.g. Intel MKL, Eigen, ...

Pro

- Low development cost
- Little coding/debugging/ maintenance
- Focus on problem

Con

- Depend on library
- No competitive edge
- Not so flexible

overview

	stay away		preferred choice		preferred choice
	assembly	**intrinsics**	**auto-vect**	**low lvl lib**	**high lvl lib**
efficiency	++	++	++	++	+
easy to use	--	-	+	+	++
flexibility	++	++	++	++	-
dependency	hardware	hardware Compiler	compiler	lib	lib
portability	-	-	+	++	++
language	F90/C/C++	C/C++	F90/C/C++	C/C++	F90/C/C++

if portability is an issue and software is first concern

if solving the problem is first concern

where to look for the details

- **https://software.intel.com/en-us/intel-software-technical-documentation**

- https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf
- https://software.intel.com/en-us/mkl_cookbook
- https://software.intel.com/en-us/articles/vectorization-essential

Universiteit Antwerpen

Vlaams Supercomputer Centrum

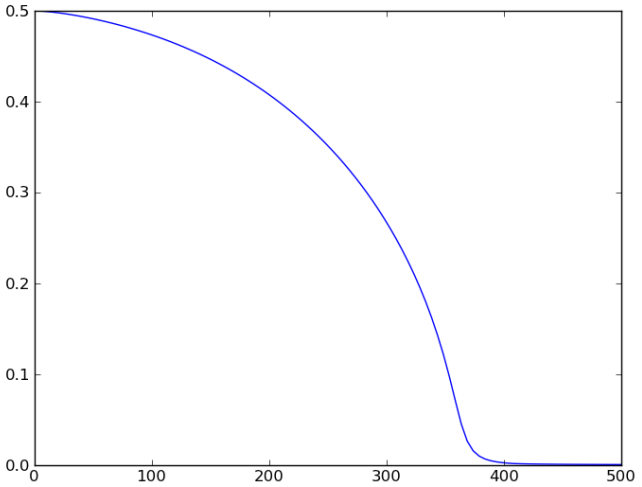
spin-waves-2.0

Prof Wim Magnus

CASE STUDY

spin-waves-2.0

- compute magnetization $m(T)$ of bulk ferro-magnets



- self consistent solution of

$$m = \frac{1}{2} \cdot \frac{1}{1 + 2\Phi(m)}, \qquad \Phi(m) = \frac{1}{N} \sum_{\mathbf{k}} \frac{1}{\mathrm{e}^{\beta\eta(\mathbf{k})m} - 1},$$

$$\beta = 1/k_{\mathrm{B}}T$$

algorithm

- loop over all *T* points (temperature)
  - initial guess m=0.5 (corresponding to $\Phi(m)=0$)
    - update $m = 1/(2+4*\Phi(m))$ until convergence

- $\Phi(m)$ is computed as

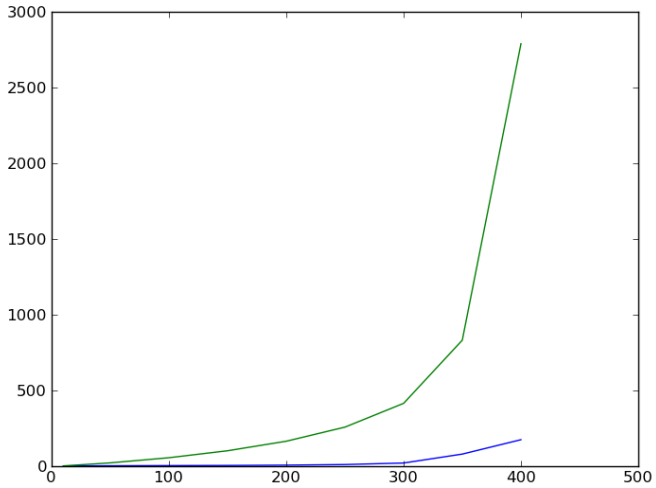
$$\Phi(m) = \sum_{n=1}^{\infty} \left(\frac{a}{\pi} \int_0^{\pi/a} \mathrm{d}q \, \mathrm{e}^{-nm\beta\eta_1(q)} \right)^3$$

  - $\Phi(m)=0$
  - loop over $n=1,\infty$
    - evaluate the integral using Gauss-Legendre-integratie on 64 points
    - term = (integral*$\alpha/\pi)^3$
    - if the term is too small
      break
      else
      $\Phi(m)$+=term

algorithm

- 4 nested loops:
- loop over *T* points
  - self-consistency iteration
    - loop over terms
      - loop over integration points
- observation:
  runtime does not increase when auto-vectorization is disabled with compiler flag
  - `-no-vec`
- auto-vectorization fails, but why?

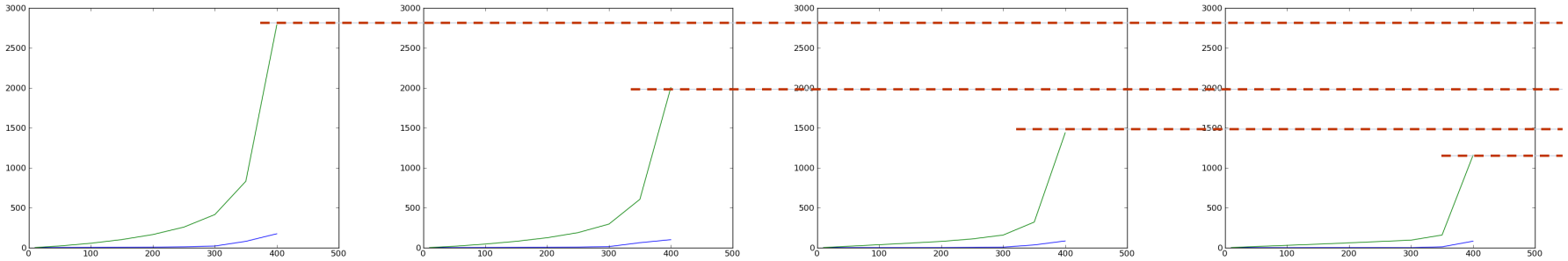
optimization 1



- number of iterations in self-consistency loop vs T

- $m(T)$ decreases monotonically
- $m(T_{i-1})$ is a better initial guess than 0.5

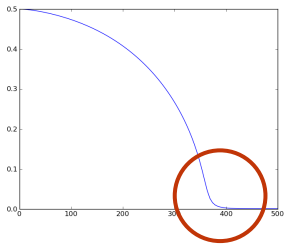
think about your problem when you code



$m_0(T_i) = 0.5$	$m_0(T_i) = m(T_{i-1})$	$m_0(T_i) =$ **linear interpolation** $m(T_{i-1}), m(T_{i-2})$	$m_0(T_i) =$ **quadratic interpolation** $m(T_{i-1}), m(T_{i-2}), m(T_{i-3})$
2789	2789->2005	2005->1439	1439->1157

2789->1157 is roughly a speed up of 2.4

optimization 2



- original code takes about 11 s to run
- 0.58 s is spent writing warnings to stdout
- when m->0 there are problems with convergence of
  - the series

$$\Phi(m) = \sum_{n=1}^{\infty} \left(\frac{a}{\pi} \int_0^{\pi/a} \mathrm{d}q \, \mathrm{e}^{-nm\beta\eta_1(q)} \right)^3$$

  - and the self-consistency loop

$$m = \frac{1}{2} \cdot \frac{1}{1 + 2\Phi(m)}$$

- I/O is expensive – don't write what you will never read
- hundreds of times the same two warnings, better use a counter and report only once

vectorization

$$\Phi(m) = \sum_{n=1}^{\infty} \left(\frac{a}{\pi} \int_0^{\pi/a} \mathrm{d}q \, \mathrm{e}^{-nm\beta\eta_1(q)} \right)^3$$

- Gauss-Legendre integration implemented using allocatable array for integration points
- the integrand is passed as function pointer
- the integrand is implemented as an *array valued function*
  - takes an array of values : the integration points
  - returns an array of values : the function values at the integration points
- array valued functions are a F90 feature.
- The returned array is automatically allocated and deallocated
- flexible and elegant but efficiency flaws:
  - lots of dynamical memory management
  - function pointer **cannot be inlined** => obstructs auto-vectorization

cure

- hardcode Gauss-Legendre integration without function pointer (two lines of code)
- elemental function allows a function that takes a scalar and returns a scalar to be applied to an array by embedding it into a vectorizable loop

```
FUNCTION Phi_integrand(ka)
   REAL (SP), INTENT(IN) :: ka(:)
   REAL (SP)             :: Phi_integrand(SIZE(ka))
   ...
```

=>

```
ELEMENTAL FUNCTION Phi_integrand(ka)
   REAL (SP), INTENT(IN) :: ka(:)
   REAL (SP)             :: Phi_integrand(SIZE(ka))
   ...
```

speedup

	runtime [s]	speedup	total speedup
original code	11.0		
optimization 1 (m_0)	4.51	2.43	2.43
optimization 2 (I/O)	3.93	1.15	2.80
vectorization	0.29	13.6	37.9

huh?

- code uses single precision and AVX (256 bit)
- vector width is 8 SP
- expected speedup of 8x from vectorization
- we gain an additional speedup of 1.7 due to removal of dynamic memory management that comes with the array valued function (integrand)
- the 0.58 s spent in I/O (optimization 2) is significant in the final code : 0.29+0.58 is 3x slower

optimization 4

- only 1 core used so far
- parallellize over N cores
  - OpenMP
  - Intel TBB
  - …
- only useful if the work load increases considerably
  - number of temperature points increases
  - single precision -> double precision (implies more iterations)
- current setup overhead would probably be larger than speedup

Final remarks

- think about the problem when you code
- C++ can be as efficient as Fortran (Tips & Tricks 1)
- Fortran also contains idioms that can kill performance (Fortran can be as inefficient as C++)
- constructs that get in the way of vectorization are often inefficient by themselves

  ⇒ the speedup of vectorization can be larger than the vector width
- it pays off to let me review your code

engelbert.tijskens@uantwerpen.be