# Core optimization


Mathieu Lobet, Matthieu Haefele

Maison de la Simulation, CEA, CNRS, Université Paris-Sud, UVSQ,
Universite Paris-Saclay, F-91191 Gif-sur-Yvette, France
(mathieu.lobet@cea.fr)

# I. Hardware description

# Levels of parallelism

# Processor architecture

| Core | |
|---|---|
| Thread | Thread |
| Cache L1 | Cache L2 |

| Core | |
|---|---|
| Thread | Thread |
| Cache L1 | Cache L2 |

| Core | |
|---|---|
| Thread | Thread |
| Cache L1 | Cache L2 |

| Core | |
|---|---|
| Thread | Thread |
| Cache L1 | Cache L2 |

Cache L3

# Levels of caches

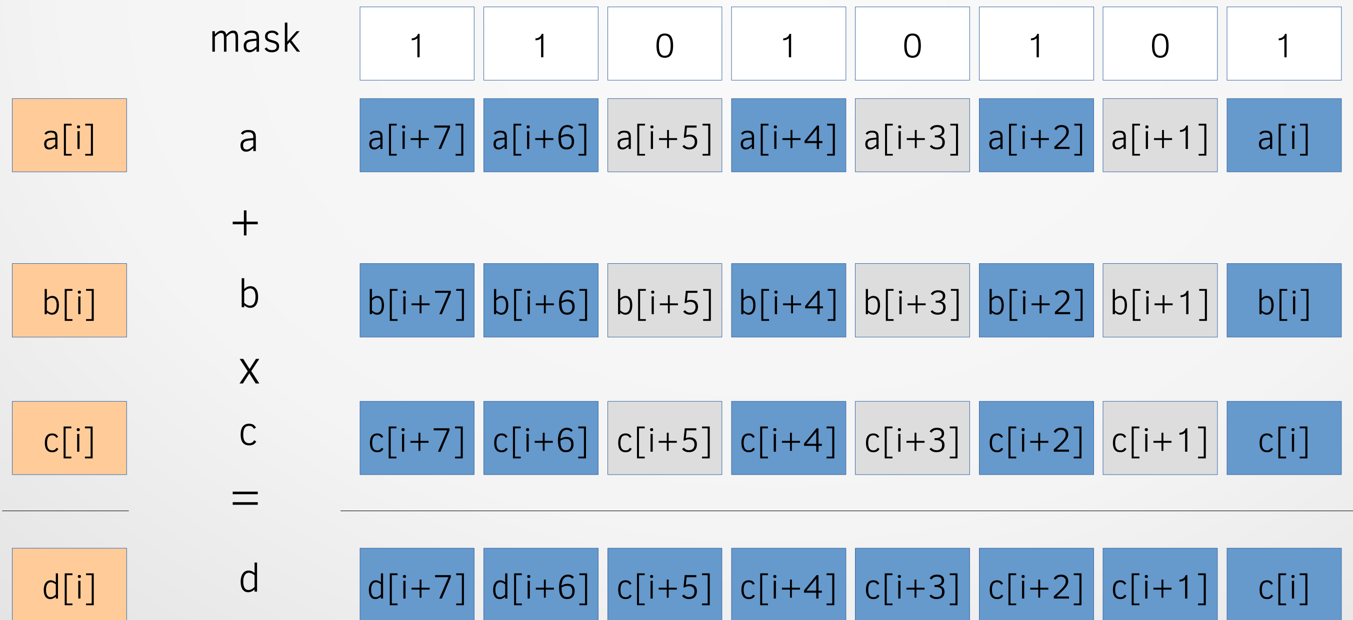
| | Size | Bandwidth | Latency |
|---|---|---|---|
| Cache L1 | ~32 kB | ~4000 Gb/s | ~1 ns<br>~5 cycles |
| Cache L2 | ~256 kB - 1Mb | ~1000 Gb/s | ~3 ns<br>~10 cycles |
| Cache L3 | ~30Mb | ~1000 Gb/s | ~20 ns |
| HBM | ~10Gb | ~500 Gb/s | ~100 ns |
| RAM | ~100 Gb | ~100 Gb/s | ~100 ns |

*https://www.7-cpu.com/cpu/Skylake_X.html*

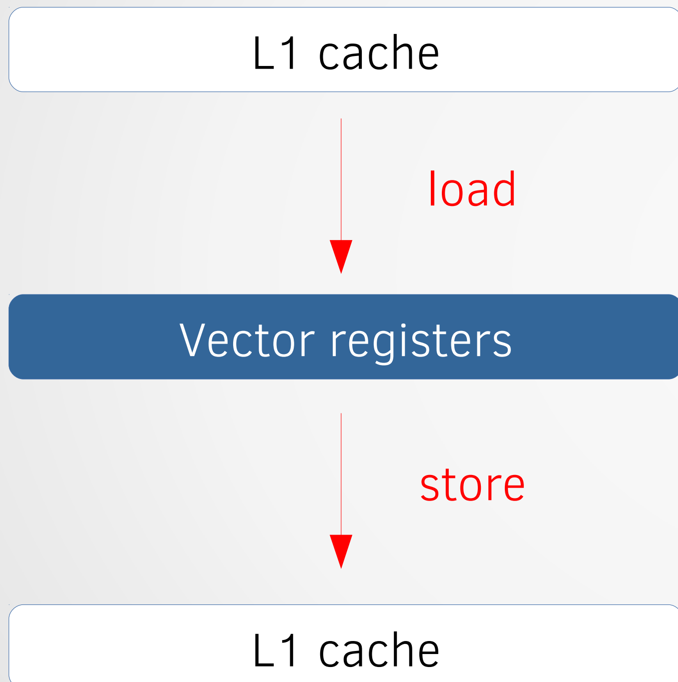
# SIMD vectorization description

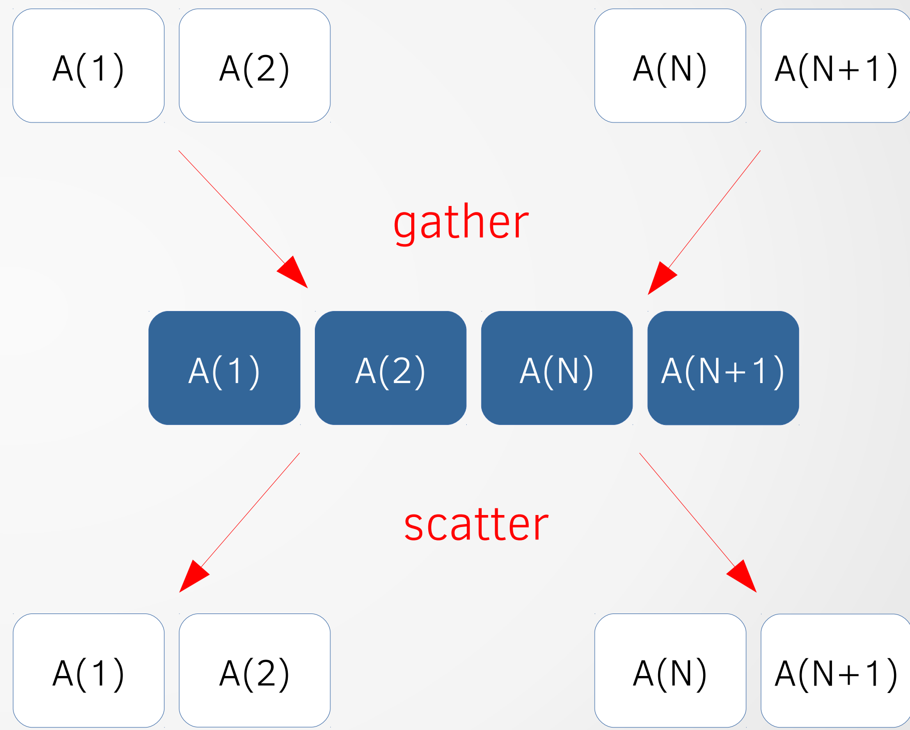
SIMD : Single Instruction Multiple Data

```
For i from 1 to N :
    If (condition on i) :
        d[i] = a[i] + b[i]*c[i]
```



# Notion of load and store



# Vectorization potential speedup

| | Instruction Set | Vector size |
|---|---|---|
| Skylake, KNL | AVX512 | 8 |
| Broadwell, Haswell | AVX2 | 4 |
| Ivy bridge | AVX | 4 |

# Processor and core performance metrics

Parallelism / Threading

- Number of cores / threads

Computational power

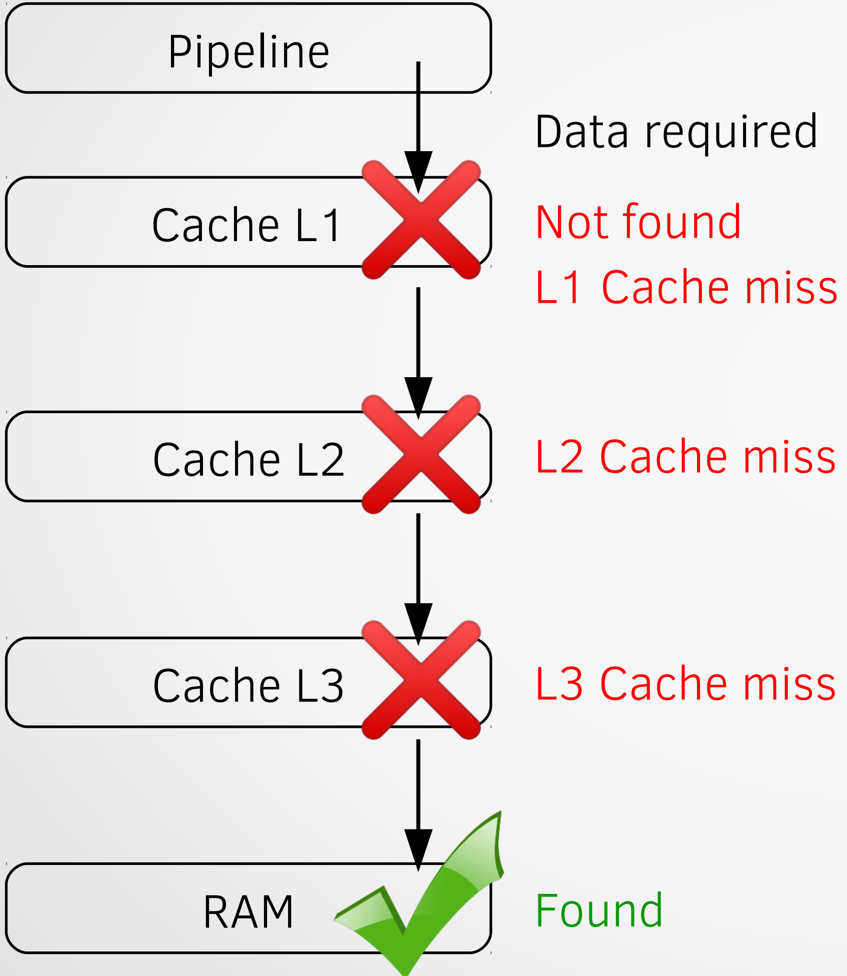
- Core frequency
- Vector register length

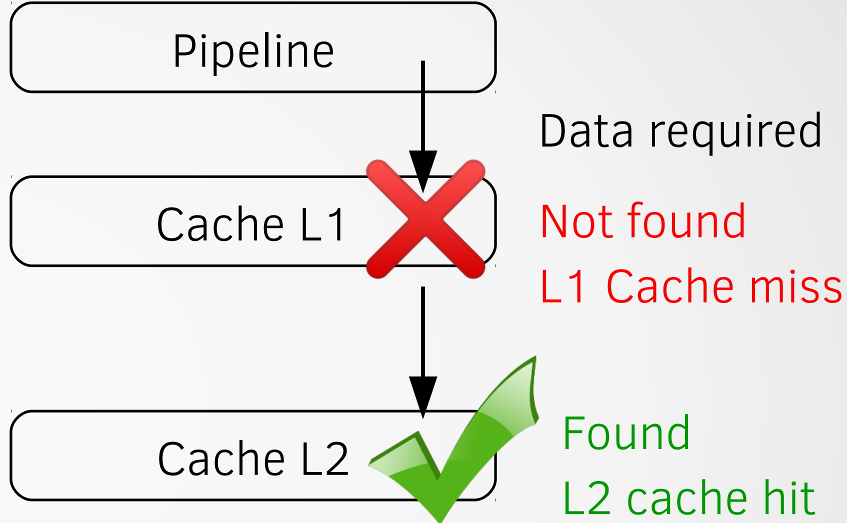
Memory

- Cache levels
- Cache size, bandwith and latency
- RAM bandwith and latency

# II. Core level parallelization and otimization

# Cache hit and cache miss



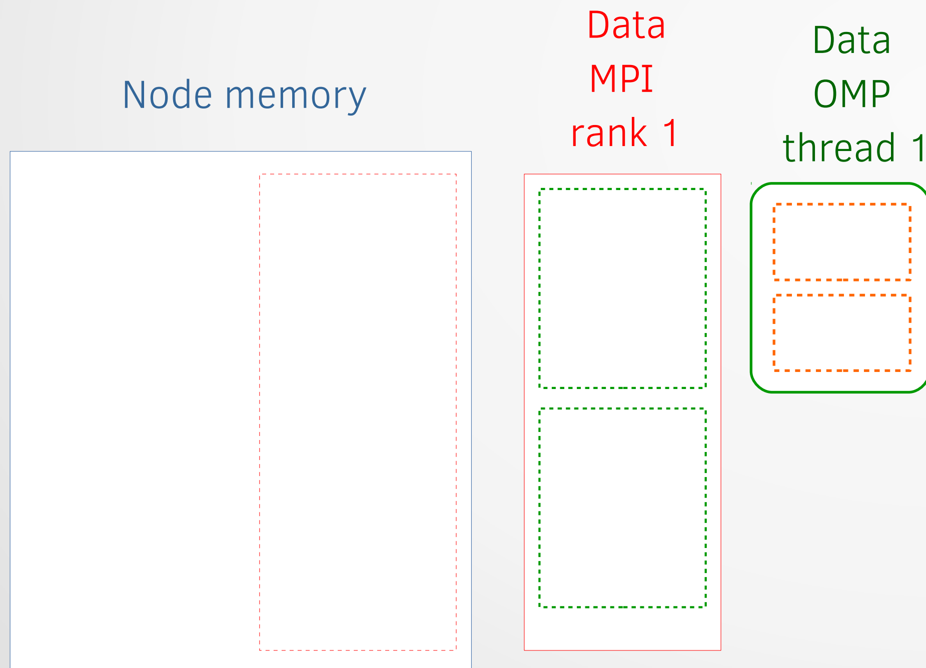



# Cache blocking

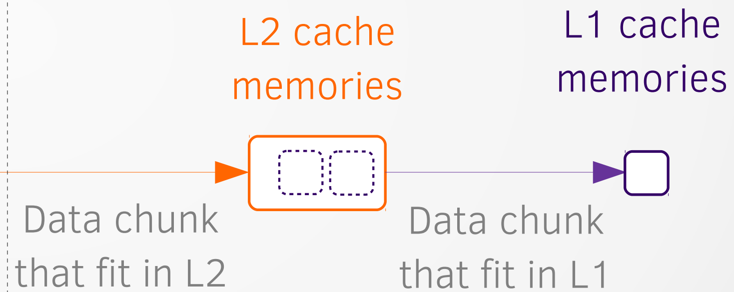
Decomposition of the workload into chunk that fit the targeted level of cache :

- Diminish number of cache misses / reuse data in cache
- Reduce memory bandwidth pressure : improve performance of memory bound algorithms

Hybrid parallel memory decomposition




Cache blocking




Chunk size < L2
Chunk number > number of OMP thread

# Vectorization factors

Factors that prevent vectorization:

- Loop data dependencies
- Pointer aliasing
- Function call in a loop

Factors that slow down vectorization:

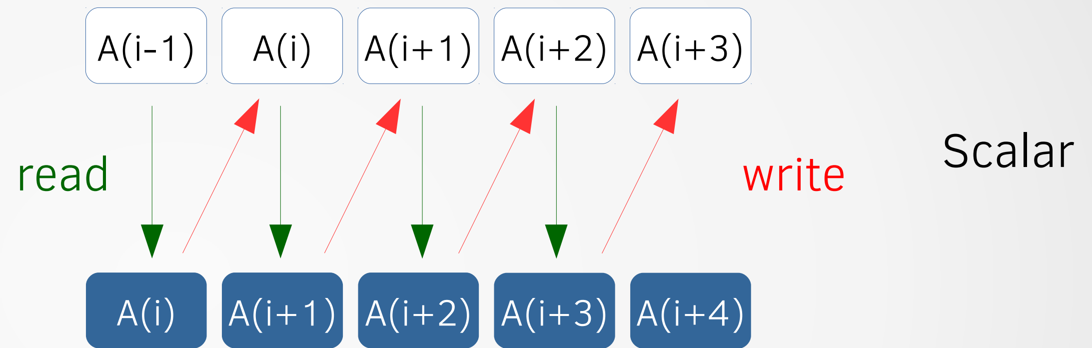
- Non-contiguous / indirect memory accesses
- Latency and throughput
- Small arithmetic intensity
- Reduction
- Mixed data types
- Branching
- Small tripcounts
- Non-multiple of the vector length
- Complex math instructions (sqrt, division...)
- Register spilling

*software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf*
*software.intel.com/en-us/articles/explicit-vector-programming-in-fortran*
*software.intel.com/en-us/articles/getting-started-with-intel-composer-xe-2013-compiler-pragmas-and-directives*

# No vectorization: data dependencies

Read-after-write (RAW) – flow dependency

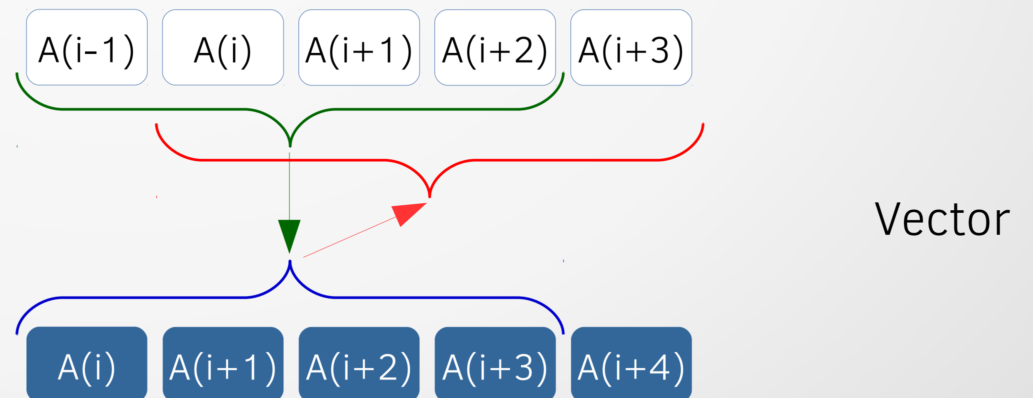
```
DO i = 2,N
  A(i) = 0.5*A(i-1) + 10.
END DO
```




Scalar

```
B = A

DO i = 2,N
  A(i) = 0.5*B(i-1) + 10.
END DO
```




Vector

*https://www.nersc.gov/users/computational-systems/edison/programming/vectorization/*

# No vectorization: data dependencies

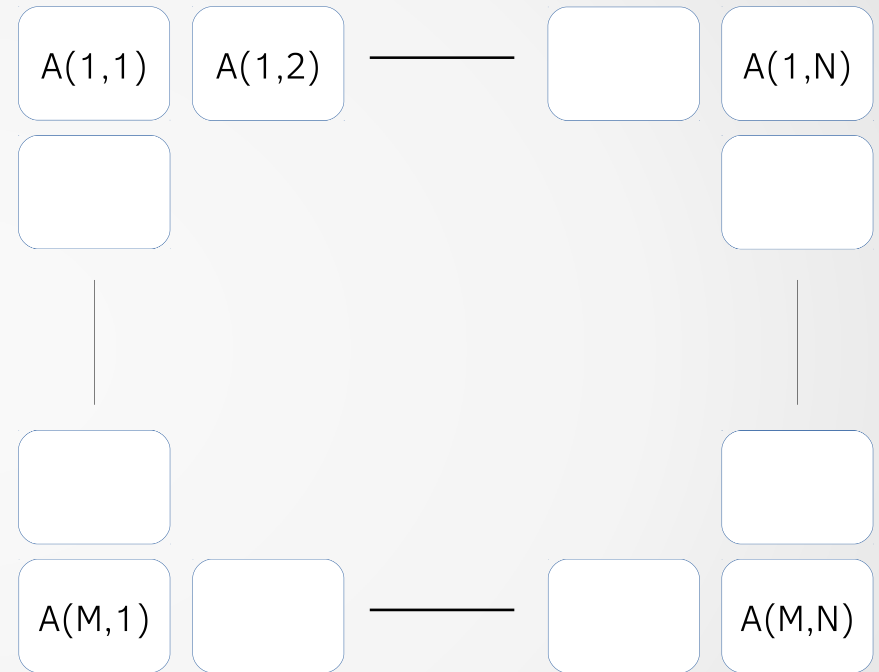
Write-after-write (WAR) – anti dependency

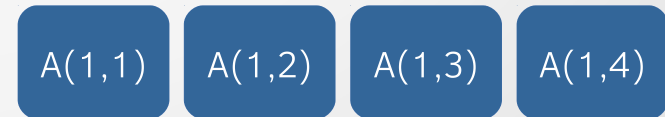
```
DO i = 2,N
  A(i-1) = B(i)
  A(i) = 2*i
ENDDO
```

*https://www.nersc.gov/users/computational-systems/edison/programming/vectorization/*

# Non-efficient vectorization: non-contiguous access

```
DO i = 1,N
  DO j = 1,M
    C(j,i) += B(j,i) * A(i,j)
  END DO
END DO
```




Vector register

# Non-efficient vectorization: non-contiguous access

Modify data structure or loop order to have continuous memory access

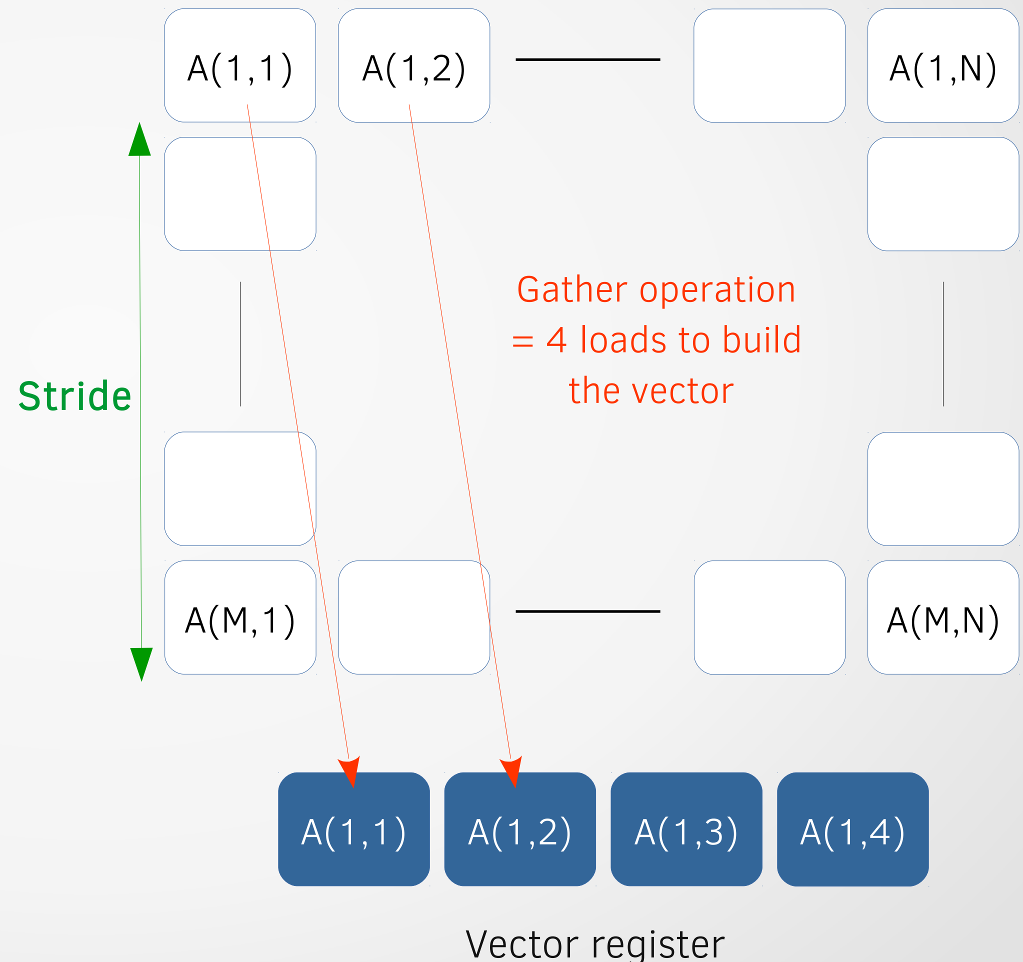
Non efficient vectorization: gather operation

```
DO i = 1,N
  DO j = 1,M
    C(j,i) += B(j,i) * A(i,j)
  END DO
END DO
```

Efficient vectorization

```
Aᵀ = Transpose(A)

!$OMP DO COLLAPSE(2) SIMD
DO i = 1,N
  DO j = 1,M
    C(j,i) += B(j,i) * Aᵀ(j,i)
  END DO
END DO
```



# Non-efficient vectorization: indirect access

## Modify data structure to have continuous memory access

Non efficient vectorization: gather operation

```
DO i = 1,N
    D(i) += B(i) * A(C(i))
END DO
```

Efficient solution depends in the case

```
DO i = 1,N
    A2(i) = A(C(i))
END DO

DO i = 1,N
    D(i) += B(i) * A2(i)
END DO
```

*software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf*
*software.intel.com/en-us/articles/explicit-vector-programming-in-fortran*
*software.intel.com/en-us/articles/getting-started-with-intel-composer-xe-2013-compiler-pragmas-and-directives*

# Non-efficient vectorization:  if-condition

Eliminate if-statements inside intenisve loops

Not efficient even in scalar

```
DO i = 1,N
    IF (Condition independent of
i)
        D(i) = C(i) + A(i) * B(i)
      ELSE
        D(i) = A(i) + C(i) * B(i)
    ENDIF
END DO
```

Efficiently Vectorized

```
If (Condition independant of i)
    !$OMP SIMD
    DO i = 1,N
        D(i) = C(i) + A(i) * B(i)
    ENDDO
ELSE
    !$OMP SIMD
    DO i=1,N
        D(i) = A(i) + C(i) * B(i)
        ENDDO
END DO
```

*software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf*
*software.intel.com/en-us/articles/explicit-vector-programming-in-fortran*
*software.intel.com/en-us/articles/getting-started-with-intel-composer-xe-2013-compiler-pragmas-and-directives*

# Non-efficient vectorization: strided data structure

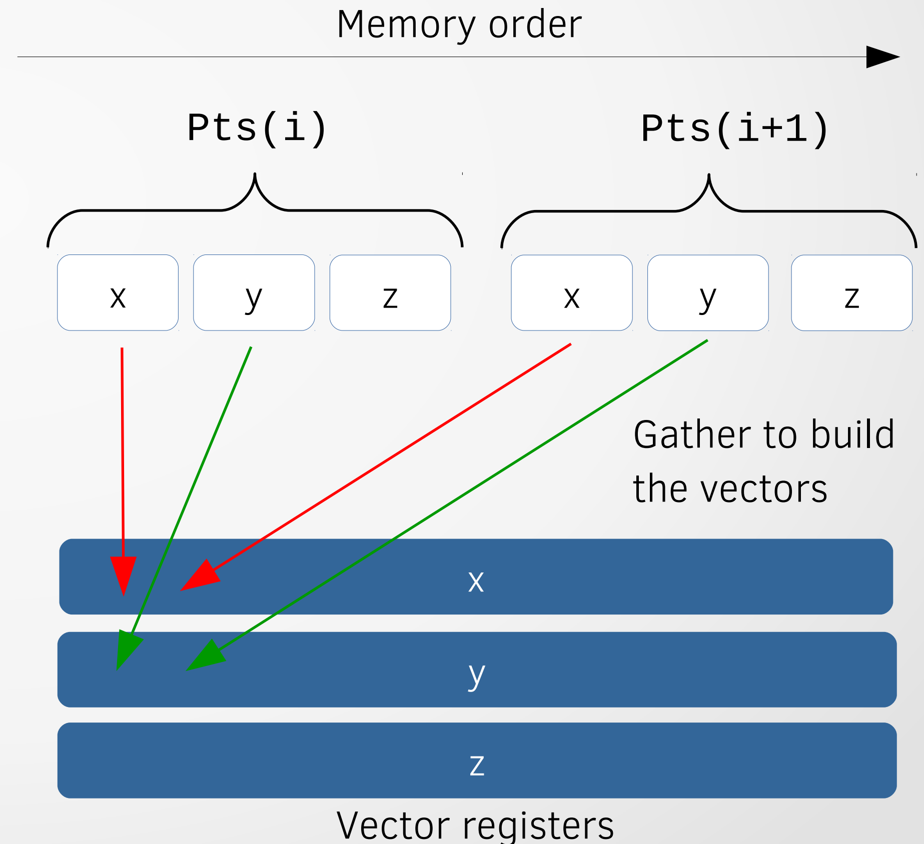
Modify data structure or loop order to have continuous memory access

Array of Structure: non-unit stride but good memory locality

```
TYPE Point
    REAL :: x,y,z
END TYPE

TYPE(Point), DIMENSION(N) :: Pts
REAL, DIMENSION(N)        :: norm


DO i=1,N
    norm(i) = Pts(i)%x**2 + Pts(i)%y**2
            + Pts(i)%z**2
END DO
```



# Non-efficient vectorization: strided data structure

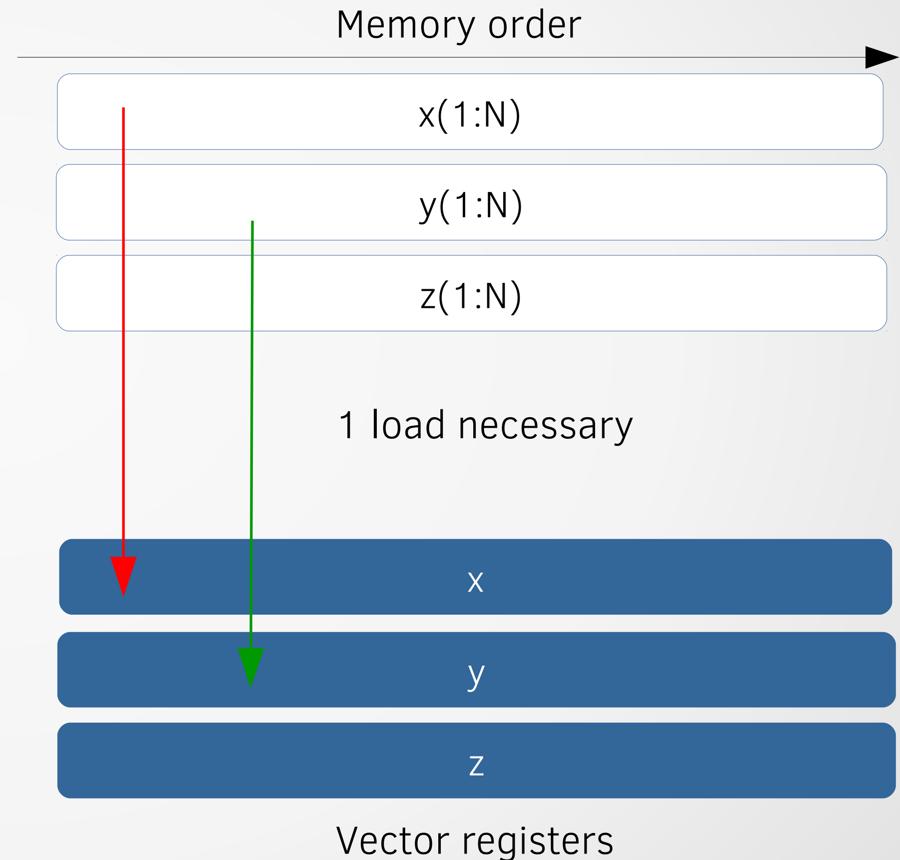
Modify data structure or loop order to have continuous memory access

Structure of Array: vectorizable

```
TYPE Point
    REAL, DIMENSION(N) :: x,y,z
END TYPE

TYPE(Point)              :: Pts
REAL, DIMENSION(N)       :: norm

DO i=1,N
    norm(i) = Pts%x(i)**2 + Pts%y(i)**2
             + Pts%z(i)**2
END DO
```



Best solutions is often Array of Structure of array = cache-blocking + structure of array

# Non-efficient vectorization: data alignment

Explicitely specify the alignment, use smart padding

Non-aligned access

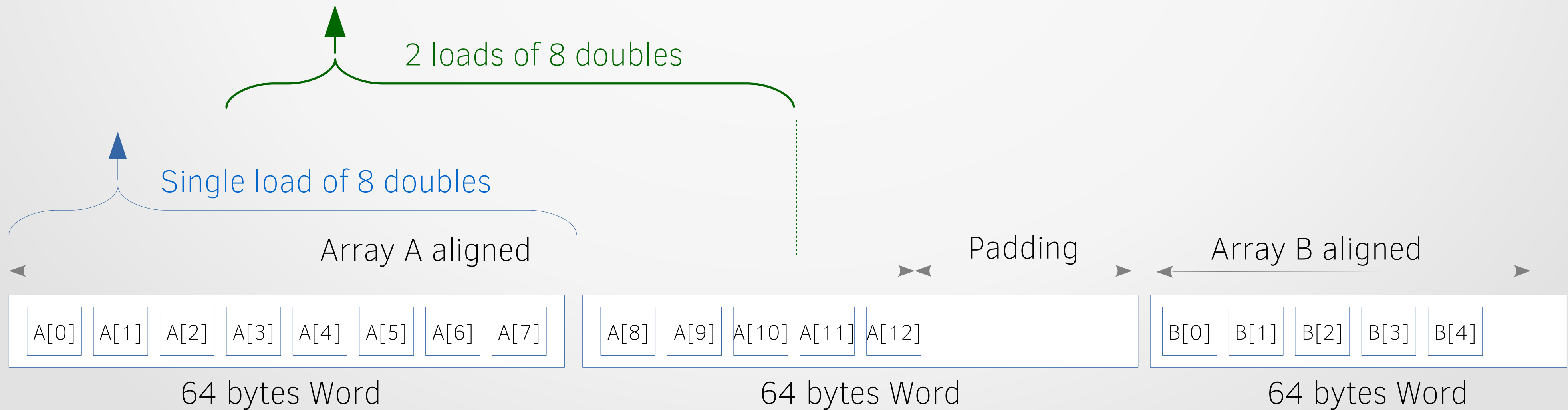
```
REAL, DIMENSION(13) :: A

DO i=3,10
    A(i) = D(i) + B(i)*C(i)
END DO
```

Aligned with initial padding

```
REAL, DIMENSION(-4:13) :: A

DO i=3,10
    A(i) = D(i) + B(i)*C(i)
END DO
```




*https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization*

# Explicit vectorization - directives

Help the compiler using Intel (!DIR$, #pragma) and OpenMP (!$OMP) directives:

- OMP SIMD / VECTOR : force vetcorization
- IVDEP : ignore vector dependencies
- INLINE, FORCEINLINE : enable function inlining
- DISTRIBUTE POINT : advice the compiler to split a loop where the directive is located
- ATTRIBUTES ALIGN : alignment of arrays

```
REAL, DIMENSION(N) :: A, B, B, D

!$OMP SIMD
DO i=1,N
    A(i) = D(i) + B(i)*C(i)
END DO
```

*https://software.intel.com/en-us/articles/explicit-vector-programming-in-fortran*

# Alignment

## Align your data : 64 bytes with AVX512

- Data alignment means putting the data at a memory adress that is a modulo of the word size
- Increase efficiency of data loads and stores from the different memory to the vector registers
- On KNL, 64 bytes alignment is optimal for vectorization
- Induce a padding between data structures

Compiler flag: `-align array64byte`

C/C++ declaration

```
float A[13]
__attribute__((aligned(64))); # Static
float * A = _mm_malloc(13, 64); #
Dynamic
float * A = _aligned_malloc(13,64); #
Windows
```

C/C++ pragmas

```
#pragma vector aligned
```

C/C++ confirmation before loops

```
__assume_aligned(A, 64);
__assume(n%16==0); # specify that n is
multiple of 16
```

Fortran declaration

```
real :: A(13)
Real, allocatable :: A
!dir$ attributes align:
64:: A
```

Fortran directives

```
!DIR$ VECTOR ALIGNED
```

Fortran confirmation before loops

```
!DIR$ ASSUME_ALIGNED A: 64
!DIR$ ASSUME (mod(n,16)
.eq. 0)
```

*https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization*