Intel® Parallel Computing Center
Research Center for Many-core HPC

Gaining Performance Through Vectorization using Fortran

Florian Wende

October 15, 2015

Zuse Institute Berlin

Agenda

Agenda

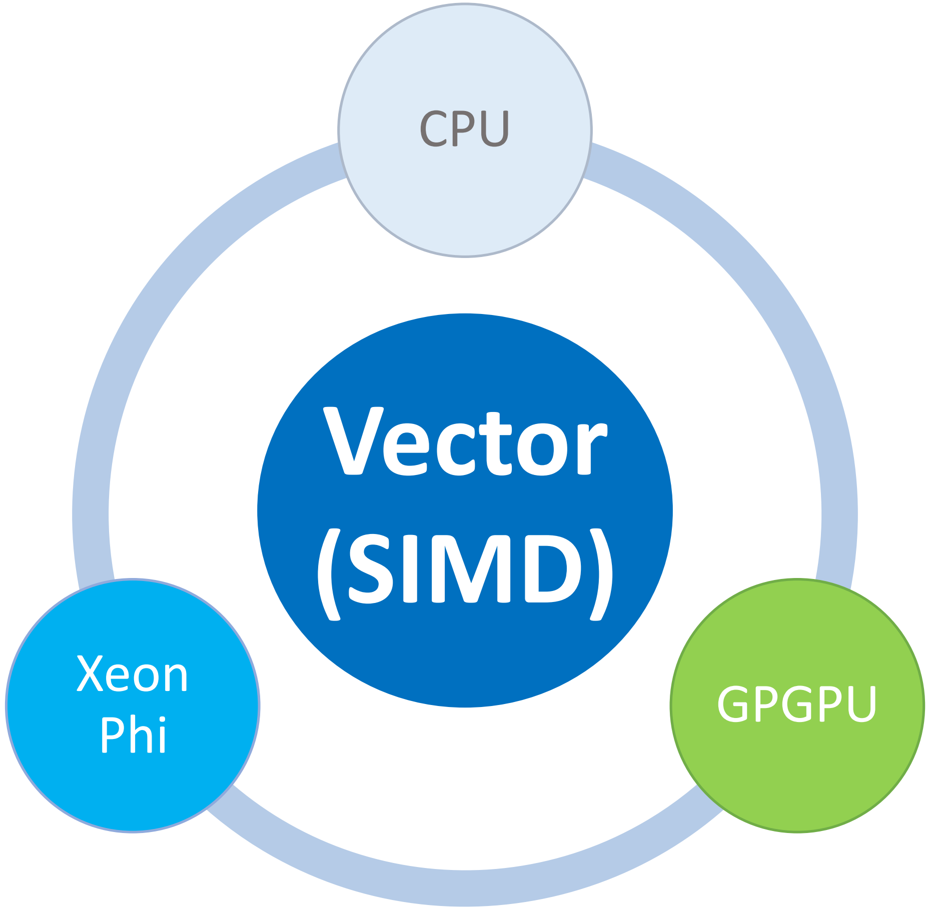
- Some general aspects
- Explicit vectorization (SIMD functions)
  - OpenMP 4.0 directives
  - Irregular control flow
  - SIMD functions
- Enhanced explicit vectorization
- Real world application: VASP

Some general aspects

How relevant is vectorization?

Modern processors increasingly draw on vectorization capabilities to achieve >TFLOP/s peak performance.

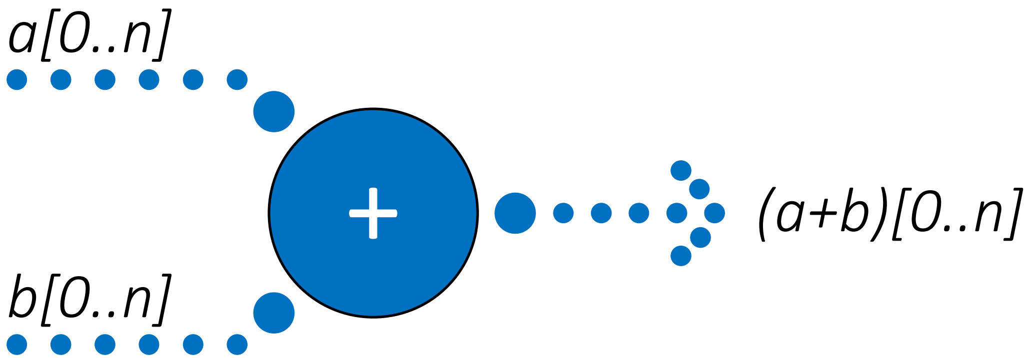
64bit vector performance on Xeon Phi: up to 8x gain over scalar execution.

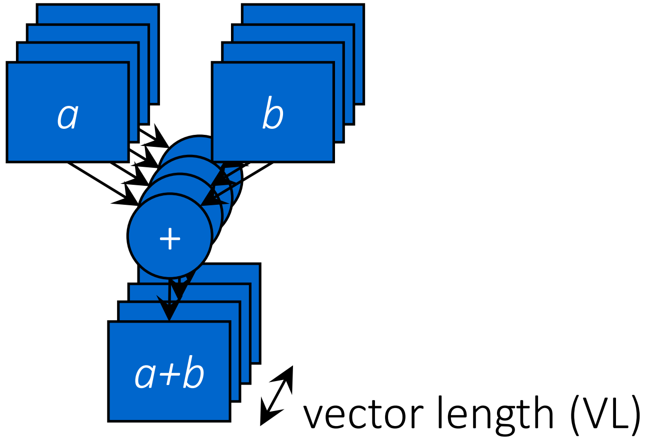


How relevant is vectorization?

Vectorization on SIMD hardware is an addition level of parallelism!

- SIMD: **S**ingle-**I**nstruction **M**ultiple-**D**ata
  → multiple data elements are processed at once.





Code base needs to consist mainly of vectorizable loops / functions.

How relevant is vectorization?

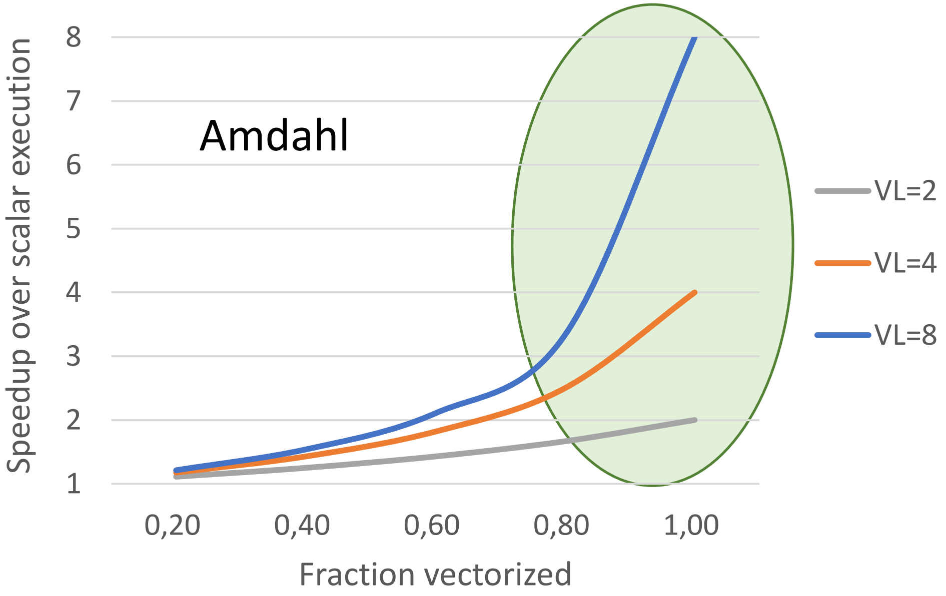
On legacy processors: nice to have.

SSE(X)

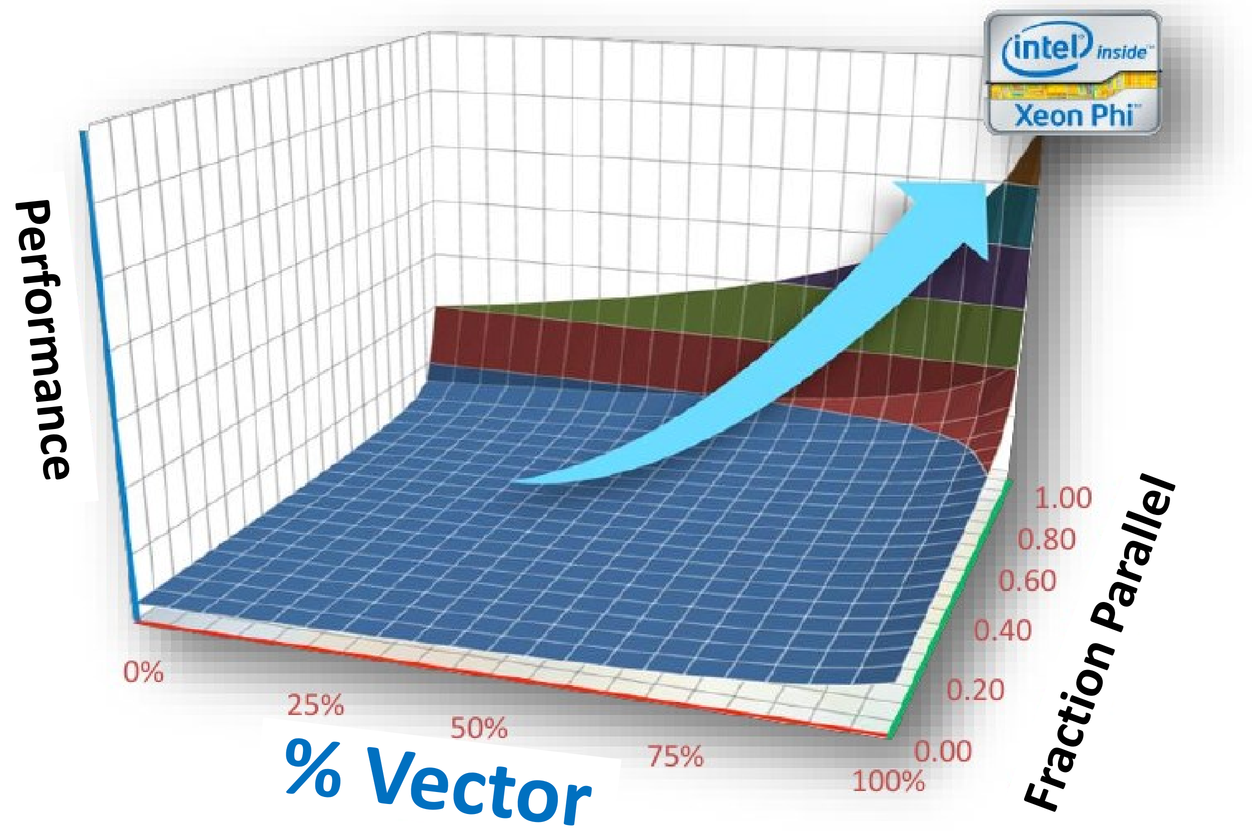
AVX(2)

On modern processors: must have.

IMCI, AVX-512



with courtesy of Intel® Corp.



How to approach vectorization?

Different levels to address vectorization in your code:

- Leave it to the compiler entirely: auto-vectorizer
- Explicit vectorization via compiler directives

Vector Modules
& Libraries

- SIMD intrinsics
- (Inline) assembly

Fortran

C/C++

high-level

low-level

Does my code vectorize? Ask the compiler

Intel compiler (15.0 onwards):

- vector report: `-opt-report=<level>` will tell you about
  - data access: alignment, gather, indexed load, scatter
  - assumed dependencies
  - loop unrolling, loop peeling, loop remainder
  - vector function calls, e.g., SVML and SIMD functions

```
vectorization support:  reference temp_dexc has aligned access...
vectorization support:  reference temp_dvxc has aligned access...
vectorization support:  unaligned access used inside loop body
loop was completely unrolled
SIMD LOOP WAS VECTORIZED
unmasked aligned unit stride loads:  28
unmasked aligned unit stride stores:  19
unmasked unaligned unit stride loads:  2
unmasked unaligned unit stride stores:  2
-- begin vector loop cost summary --
scalar loop cost:  2910.000
vector loop cost:  1315
estimated potential speedup:  1.910
lightweight vector operations:  1364
medium-overhead vector operations:  4
vectorized math library calls:  14
vector function calls:  15
-- end vector loop cost summary --
-- begin vector function matching report --
Masked function call:  EX_NREL with simdlen=2,actual parameter types:(vector,uniform)
A suitable vector variant was found (out of 2) with xmm,simdlen=2,masked,...
```

Explicit vectorization (SIMD functions)

Vectorizing compilers

Auto-vectorization

- part of the the optimization process at high optimization levels
- no annotations and compiler directives in the source code

Result: simple loops and code blocks are vectorized.

Explicit vectorization

- code annotations and compiler directives in the source code to
  - hint at vectorization potential, despite of assumed vector dependencies
  - give additional information to the compile regarding, e.g., data alignment
  - allow the compiler to use less precise but vectorization-friendly math operations

Result: complex loops are (hopefully) vectorized ☺

Explicit vectorization with OpenMP 4.0

Explicit vectorization with OpenMP 4.0 directives:

- Unified set of compiler directives supported by Intel, Cray, GNU, … → portability
- C/C++ and Fortran: for Fortran see, e.g.,
  www.openmp.org/mp-documents/OpenMP4.0.0.pdf
  www.openmp.org/mp-documents/OpenMP-4.0-Fortran.pdf

SIMD loop

```
!$omp simd [clause[[,]clause]…]
  do-loops enclosing a structured block
[!$omp end simd]

clause: safelen(length)
        linear(list[:linear-step])
        aligned(list[:alignment])
        private(list)
        lastprivate(list)
        reduction(op:list)
        collapse(n)
```

SIMD function

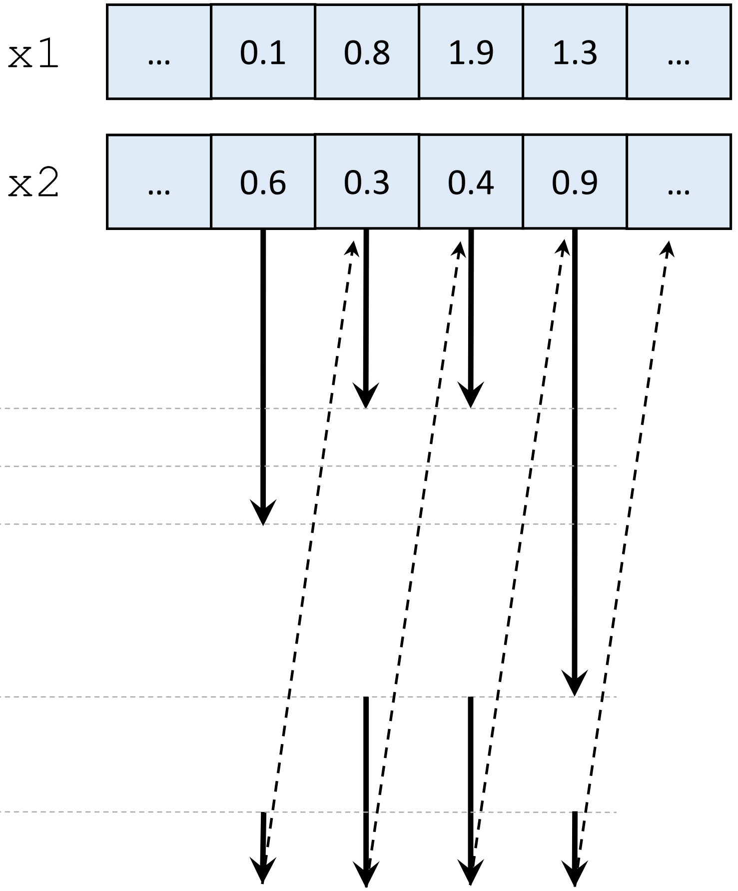
```
subroutine foo(…)
!$omp declare simd(foo) [clause[[,]clause]…]
  structured block
end subroutine foo

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

Explicit vectorization with OpenMP 4.0

Example: scalar case

```
do i=1,n
!dir$ noinline
  call foo(x1(i),x2(i),y(i))
```

```
subroutine foo(x1,x2,y)
  real*8::x1,x2,y
  if (x2.gt.0.5) then
    y=sqrt(x1)
    if (y.gt.1.0) then
      y=log(y)
    endif
  else
    y=0.0
  endif
end subroutine foo
```



Explicit vectorization with OpenMP 4.0

Example: compilation with Intel 16.0 compiler.

```
91: do i=1,n
92: !dir$ noinline
93:   call foo(x1(i),x2(i),y(i))
94: enddo

subroutine foo(x1,x2,y)
  ...
end soubroutine foo
```

Compile string for Xeon Phi:

```
ifort -mmic -O3 -openmp -align all -opt-report 5 …
```

Optimization report:

```
…
LOOP BEGIN at ***.F90(91,7)
  remark #15382: vectorization support: call to function FOO
  cannot be vectorized
  …
```

No vector version of this loop with auto-vectorizer!

Setup: n=8*1024*1024 random numbers, x1 uniformly in [0.0,2.0], x2 uniformly in [0.0,1.0]

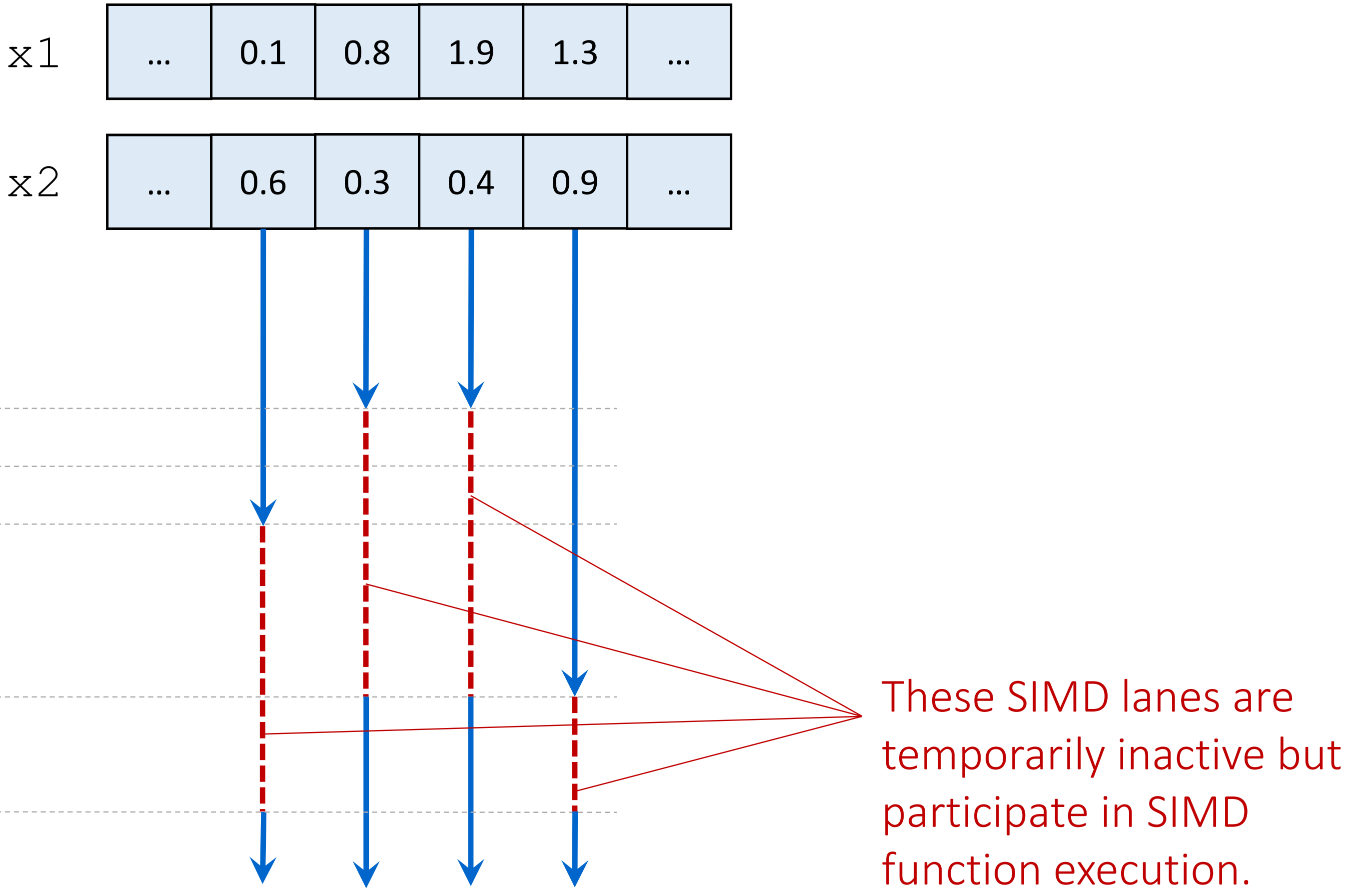
Execution time on Xeon Phi 5120D: 1200ms (reference)

Explicit vectorization with OpenMP 4.0

Example: SIMD case

```
!$omp simd
do i=1,n
!dir$ noinline
  call foo(x1(i),x2(i),y(i))
```

```
subroutine foo(x1,x2,y)
!$omp declare simd(foo)...
  real*8::x1,x2,y
  if (x2.gt.0.5) then
    y=sqrt(x1)
    if (y.gt.1.0) then
      y=log(y)
    endif
  else
    y=0.0
  endif
end subroutine foo
```



Explicit vectorization with OpenMP 4.0

Example: compilation with Intel 16.0 compiler.

```
90: !$omp simd
91: do i=1,n
92: !dir$ noinline
93:   call foo(x1(i),x2(i),y(i))
94: enddo

subroutine foo(x1,x2,y)
!$omp declare simd(foo) simdlen(8)
  …
end soubroutine foo
```

Compile string for Xeon Phi:

```
ifort -mmic -O3 -openmp -align all -opt-report 5 …
```

Optimization report:

```
…
LOOP BEGIN at ***.F90(91,7)
   remark #15388: vectorization support: scatter was generated for
   variable X1: indirect, 64bit indexed
   …
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   …
   remark #15484: vector function calls: 1
   …
   remark #15492: a suitable vector variant was found (out of 2)
   with mic, simdlen=8, unmasked, formal parameter
   types(vector,vector,vector)
   …
```

Setup: n=8*1024*1024 random numbers, x1 uniformly in [0.0,2.0], x2 uniformly in [0.0,1.0]

Execution time on Xeon Phi 5120D: 1340ms (0.9x over reference)

Explicit vectorization with OpenMP 4.0

Why slowdown?

- Assembly: `vgather` / `vscatter` / `vpshufd`???

```
Begin optimization report for FOO..zN8vvv
…
remark #15301: FUNCTION WAS VECTORIZED
remark #15415: vectorization support: gather was
generated for the variable x2: 64bit indexed
…
```

Can we fix that? Yes and no!

```
90: !$omp simd
91: do i=1,n
92: !dir$ noinline
92:   call foo(x1(1),x2(1),y(1),i)
93: enddo
```

Currently, this does not work in Fortran!

[`no suitable vector variant was found`]

```
subroutine foo(x1,x2,y,i)
!$omp declare simd(foo) simdlen(8) uniform(x1,x2,y) [ linear(i:1) ]
  real*8::x1(*),x2(*),y(*)
  integer::i
  …
  if (x2(i).gt.0.5) then  !! Replace: VAR -> VAR(i)
    …
```

Explicit vectorization with OpenMP 4.0

Lets try different values for `simdlen`: (4,) 8, 16, 32

Xeon Phi 5120D

Scalar reference: 1200ms loop execution time

	simdlen(8)	simdlen(16)	simdlen(32)
Explicit vectorization [Intel]	1340ms	1290ms	1360ms

Xeon E5-2680v3 @ 1.9 GHz (AVX base frequency)

Scalar reference: 220ms (Intel), resp. 230ms (Cray, CCE 8.4.0.219) loop execution time

	simdlen(4)	simdlen(8)	simdlen(16)	simdlen(32)
Explicit vectorization [Intel]	220ms	220ms	233ms	247ms
Explicit vectorization [Cray]	230ms	230ms	230ms	230ms

Enhanced explicit vectorization

Enhanced explicit vectorization

General idea for SIMD functions:*

- Define vector data types with generic vector length VL
- Split loops into chunks of size VL
  - Prologue: pack vectors & create vector mask
  - Call SIMD function(s) with vectors and vector mask as arguments
  - Epilogue: unpack vectors
- Expand scalar functions to SIMD functions
  - Replace scalar arguments by vector arguments
  - Move innermost loop with trip count VL inside the function:
    loop body deduced from scalar function body + additional control logic
  - Process vectors within innermost loop using explicit vectorization

Details on the next slides

* consider loop bodies as inlined functions

Enhanced explicit vectorization

```
#define VL 8 ! e.g. for 64bit words on Xeon Phi
module simd
  type,public::simd_real8
  real*8::x(0:VL-1)
  end type simd_real8
  type,public::simd_mask8
  logical::x(0:VL-1)
  end type simd_mask8
end module simd
```

Enhanced explicit vectorization

```
#define VL 8 ! e.g. for 64bit words on Xeon Phi
module simd
  type,public::simd_real8
  real*8::x(0:VL-1)
  end type simd_real8
  type,public::simd_mask8
  logical::x(0:VL-1)
  end type simd_mask8
end module simd
```

```
type(simd_real8)::vx1,vx2,vy
type(simd_mask8)::m
…
do i=1,n,VL
!$omp simd
  do ii=0,VL-1
    m%x(ii)=.false.
    if ((i+ii).le.n) then
      m%x(ii)=.true.
      vx1%x(ii)=x1(i+ii)
      vx2%x(ii)=x2(i+ii)
    endif
  enddo




enddo
```

Enhanced explicit vectorization

```
#define VL 8 ! e.g. for 64bit words on Xeon Phi
module simd
  type,public::simd_real8
  real*8::x(0:VL-1)
  end type simd_real8
  type,public::simd_mask8
  logical::x(0:VL-1)
  end type simd_mask8
end module simd


subroutine vfoo(x1,x2,y,m)
  use simd
  type(simd_real8)::x1,x2,y
  type(simd_mask8)::m
  …
end subroutine vfoo
```

```
type(simd_real8)::vx1,vx2,vy
type(simd_mask8)::m
…
do i=1,n,VL
!$omp simd
  do ii=0,VL-1
    m%x(ii)=.false.
    if ((i+ii).le.n) then
      m%x(ii)=.true.
      vx1%x(ii)=x1(i+ii)
      vx2%x(ii)=x2(i+ii)
    endif
  enddo
  call vfoo(vx1,vx2,vy,m)




enddo
```

Enhanced explicit vectorization

```
#define VL 8 ! e.g. for 64bit words on Xeon Phi
module simd
  type,public::simd_real8
  real*8::x(0:VL-1)
  end type simd_real8
  type,public::simd_mask8
  logical::x(0:VL-1)
  end type simd_mask8
end module simd


subroutine vfoo(x1,x2,y,m)
  use simd
  type(simd_real8)::x1,x2,y
  type(simd_mask8)::m
  …
end subroutine vfoo
```

```
type(simd_real8)::vx1,vx2,vy
type(simd_mask8)::m
…
do i=1,n,VL
!$omp simd
  do ii=0,VL-1
    m%x(ii)=.false.
    if ((i+ii).le.n) then
      m%x(ii)=.true.
      vx1%x(ii)=x1(i+ii)
      vx2%x(ii)=x2(i+ii)
    endif
  enddo
  call vfoo(vx1,vx2,vy,m)
!$omp simd
  do ii=0,VL-1
     if(m%x(ii)) y(i+ii)=vy%x(ii)
  enddo
enddo
```

Enhanced explicit vectorization

Expand scalar function to SIMD function

```
subroutine foo(x1,x2,y)
  real*8::x1,x2,y
  if (x2.gt.0.5) then
    y=sqrt(x1)
    if (y.gt.1.0) y=log(y)
  else
    y=0.0
  endif
end subroutine foo
```

```
subroutine vfoo(x1,x2,y,m)
  use simd
  type(simd_real8)::x1,x2,y
  type(simd_mask8)::m





end subroutine vfoo
```

Enhanced explicit vectorization

Expand scalar function to SIMD function

```
subroutine foo(x1,x2,y)
  real*8::x1,x2,y
  if (x2.gt.0.5) then
    y=sqrt(x1)
    if (y.gt.1.0) y=log(y)
  else
    y=0.0
  endif
end subroutine foo
```

```
subroutine vfoo(x1,x2,y,m)
  use simd
  type(simd_real8)::x1,x2,y
  type(simd_mask8)::m

  integer::ii
!$omp simd
  do ii=0,VL-1



  enddo
end subroutine vfoo
```

Enhanced explicit vectorization

Expand scalar function to SIMD function

```
subroutine foo(x1,x2,y)
  real*8::x1,x2,y
  if (x2.gt.0.5) then
    y=sqrt(x1)
    if (y.gt.1.0) y=log(y)
  else
    y=0.0
  endif
end subroutine foo
```

```
subroutine vfoo(x1,x2,y,m)
  use simd
  type(simd_real8)::x1,x2,y
  type(simd_mask8)::m
  real*8::temp
  integer::ii
!$omp simd
  do ii=0,VL-1
   if (x2%x(ii).gt.0.5) then
     temp=sqrt(x1%x(ii))
     if (temp.gt.1.0) temp=log(temp)
   else
     temp=0.0
   endif

  enddo
end subroutine vfoo
```

Enhanced explicit vectorization

Expand scalar function to SIMD function

```fortran
subroutine foo(x1,x2,y)
  real*8::x1,x2,y
  if (x2.gt.0.5) then
    y=sqrt(x1)
    if (y.gt.1.0) y=log(y)
  else
    y=0.0
  endif
end subroutine foo
```

```fortran
subroutine vfoo(x1,x2,y,m)
  use simd
  type(simd_real8)::x1,x2,y
  type(simd_mask8)::m
  real*8::temp
  integer::ii
!$omp simd
  do ii=0,VL-1
    if (x2%x(ii).gt.0.5) then
      temp=sqrt(x1%x(ii))
      if (temp.gt.1.0) temp=log(temp)
    else
      temp=0.0
    endif
    if (m%x(ii)) y%x(ii)=temp
  enddo
end subroutine vfoo
```

Only active lanes store their results

Enhanced explicit vectorization

Xeon Phi 5120D

Scalar reference: 1200ms loop execution time

	simdlen(8)	simdlen(16)	simdlen(32)
Explicit vectorization [Intel]	1340ms	1290ms	1360ms
Enhanced explicit vectorization [Intel]	**570ms**	**540ms**	**570ms**

Xeon E5-2680v3 @ 1.9 GHz (AVX base frequency)

Scalar reference: 210ms (Intel), resp. 230ms (Cray, CCE 8.4.0) loop execution time

	simdlen(4)	simdlen(8)	simdlen(16)	simdlen(32)
Explicit vectorization [Intel]	210ms	220ms	235ms	250ms
Enhanced explicit vectorization [Intel]	**140ms**	**130ms**	**130ms**	**130ms**
Explicit vectorization [Cray]	230ms	230ms	230ms	230ms
Enhanced explicit vectorization [Cray]	**245ms**	**156ms**	**156ms**	**154ms**

Enhanced explicit vectorization

Skip expensive code blocks:

```
subroutine foo_a(…)
  …
  a=…
  if (predicate(a)) then
    < do some compute intensive stuff >
  endif
  …
end subroutine foo_a
```

```
subroutine vfoo_a(…,m0)
  …
  type(simd_mask8)::m0,m1
  logical::true_for_any=.false.
!$omp simd reduction(.or.:true_for_any)
  do ii=0,VL-1
    …
    a=…
    m1%x(ii)=.false.
    if (m0%x(ii).and.predicate(a)) then
      true_for_any=.true.
      m1%x(ii)=.true.
    endif
  enddo
  …
end subroutine vfoo_a
```

Enhanced explicit vectorization

Skip expensive code blocks:

```
subroutine foo_a(…)
  …
  a=…
  if (predicate(a)) then
    < do some compute intensive stuff >
  endif
  …
end subroutine foo_a
```

```
subroutine vfoo_a(…,m0)
  …
  type(simd_mask8)::m0,m1
  logical::true_for_any=.false.
!$omp simd reduction(.or.:true_for_any)
  do ii=0,VL-1
    …
    a=…
    m1%x(ii)=.false.
    if (m0%x(ii).and.predicate(a)) then
      true_for_any=.true.
      m1%x(ii)=.true.
    endif
  enddo
  if (true_for_any) then
    !$omp simd
    do ii=0,VL-1
      < do some compute intensive stuff (using mask m1) >
    enddo
  endif
  …
end subroutine vfoo_a
```

skip entire code block if none of the SIMD lanes is active

Enhanced explicit vectorization

Loops on SIMD lanes: trip count known at compile time

```
subroutine foo_b(…)
   …
   do k=1,6
      < loop body >
   enddo
   …
end subroutine foo_b
```

```
subroutine vfoo_b(…,m)
   …
!$omp simd
   do ii=0,VL-1
      …
!dir$ novector
      do k=1,6
         < loop body >
      enddo
      …
   enddo
   …
end subroutine vfoo_b
```

Enhanced explicit vectorization

Loops on SIMD lanes: trip count known just at runtime

```
subroutine foo_c(…)
  …
  a=…
  do while (predicate(a))
    < loop body >
    a=…
  enddo
  …
end subroutine foo_c
```

```
subroutine vfoo_c(…,m0)
  …
  logical::true_for_any=.false.
!$omp simd reduction(.or.:true_for_any)
  do ii=0,VL-1
    …; a=…
    m1%x(ii)=.false.
    if (m0%x(ii).and.predicate(a)) then
      true_for_any=.true.
      m1%x(ii)=.true.
    endif
  enddo
  do while (true_for_any)







  enddo
  …
end subroutine vfoo_c
```

Enhanced explicit vectorization

Loops on SIMD lanes: trip count known just at runtime

```
subroutine foo_c(…)
  …
  a=…
  do while (predicate(a))
    < loop body >
    a=…
  enddo
  …
end subroutine foo_c
```

```
subroutine vfoo_c(…,m0)
  …
  logical::true_for_any=.false.
!$omp simd reduction(.or.:true_for_any)
  do ii=0,VL-1
    …; a=…
    m1%x(ii)=.false.
    if (m0%x(ii).and.predicate(a)) then
      true_for_any=.true.
      m1%x(ii)=.true.
    endif
  enddo
  do while (true_for_any)

!$omp simd
    do ii=0,VL-1
      < loop body (using mask m1) >
      a=…




    enddo
  enddo
  …
end subroutine vfoo_c
```

Enhanced explicit vectorization

Loops on SIMD lanes: trip count known just at runtime

```
subroutine foo_c(…)
  …
  a=…
  do while (predicate(a))
    < loop body >
    a=…
  enddo
  …
end subroutine foo_c
```

```
subroutine vfoo_c(…,m0)
  …
  logical::true_for_any=.false.
!$omp simd reduction(.or.:true_for_any)
  do ii=0,VL-1
    …; a=…
    m1%x(ii)=.false.
    if (m0%x(ii).and.predicate(a)) then
      true_for_any=.true.
      m1%x(ii)=.true.
    endif
  enddo
  do while (true_for_any)
    true_for_any=.false.
!$omp simd reduction(.or.:true_for_any)
    do ii=0,VL-1
      < loop body (using mask m1) >
      a=…
      if (m1%x(ii).and.predicate(a)) then
        true_for_any=.true.
      else
        m1%x(ii)=.false.
      endif
    enddo
  enddo
  …
end subroutine vfoo_c
```

Enhanced explicit vectorization

Loops on SIMD lanes: trip count known just at runtime

The kernel:

```
subroutine foo(x1,x2,d,y)
  real*8::x1,x2,y
  integer::d,i,imax
  i=1
  imax=int(d*x2)
  y=0.0
  do while (i.le.imax)
    y=sqrt(x1+y)
    if (y.gt.1.0) y=log(y)
    i=i+1
  enddo
end subroutine foo
```

d=20 (maximum number of loop interations)
n=8*1024*1024 random numbers
x1 uniformly in [0.0,2.0]
x2 uniformly in [0.0,1.0]

Xeon Phi 5120D

Scalar reference: 12.5s loop execution time

	simdlen(8)	simdlen(16)	simdlen(32)
Explicit vectorization [Intel]	8.7s	9.1s	9.6s
Enh. explicit vectorization [Intel]	**5.3s**	**4.4s**	**4.7s**

Xeon E5-2680v3 @ 1.9GHz (AVX base frequency)

Scalar reference: 2.6s (Intel), resp. 2.5s (Cray, CCE 8.4.0) loop execution time

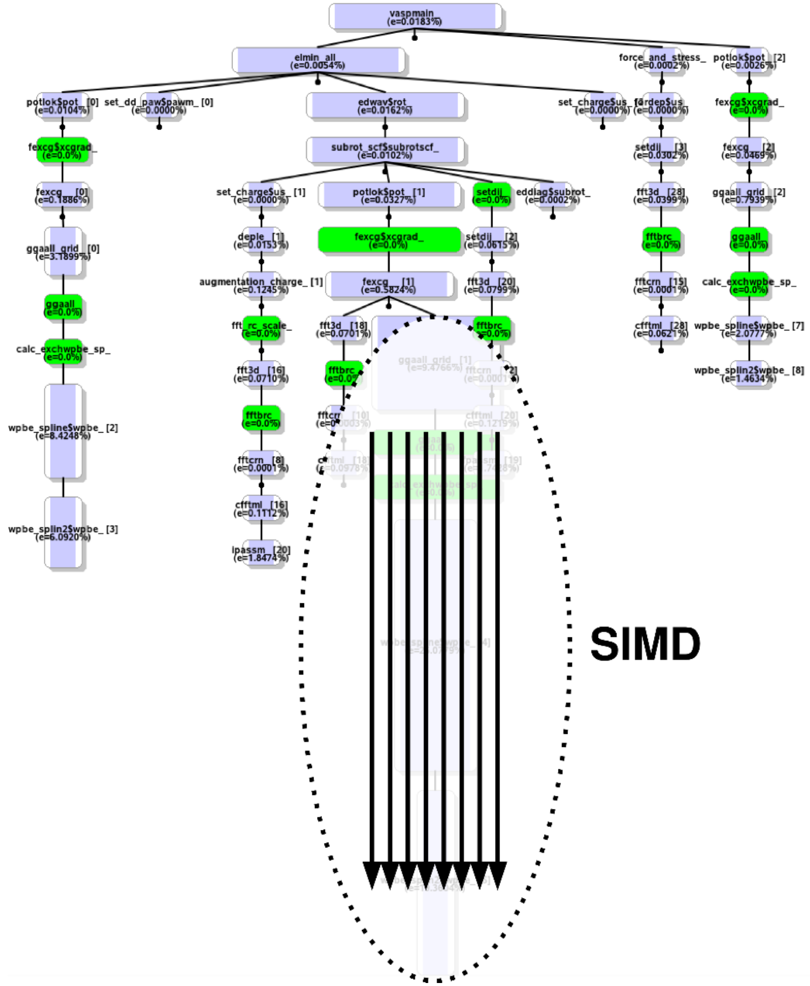
	simdlen(4)	simdlen(8)	simdlen(16)	simdlen(32)
Explicit vectorization [Intel]	2.0s	1.9s	2.1s	2.2s
Enh. expl. vectorization [Intel]	**1.8s**	**1.6s**	**1.7s**	**1.7s**
Explicit vectorization [Cray]	2.5s	2.5s	2.5s	2.5s
Enh. expl. vectorization [Cray]	**1.7s**	**1.5s**	**1.6s**	**1.7s**

Real world application: VASP

Enhanced explicit vectorization in VASP

VASP is a plane wave electronic structure code to model atomic scale materials from first principals

- ❖ Widely used in material sciences
- ❖ Historically grown with lots of features: transition to manycore era requires to address threading + SIMD
- ❖ Modernization of the code base by VASP-team + Intel + ZIB + NERSC



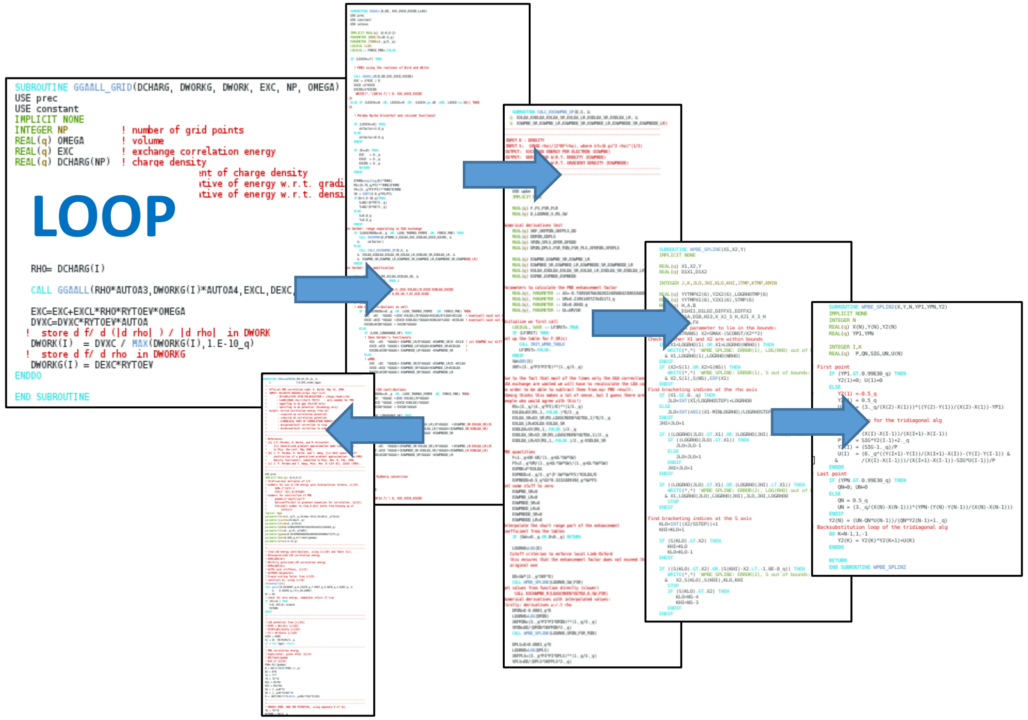
Input: Octanol molecule in a large volume

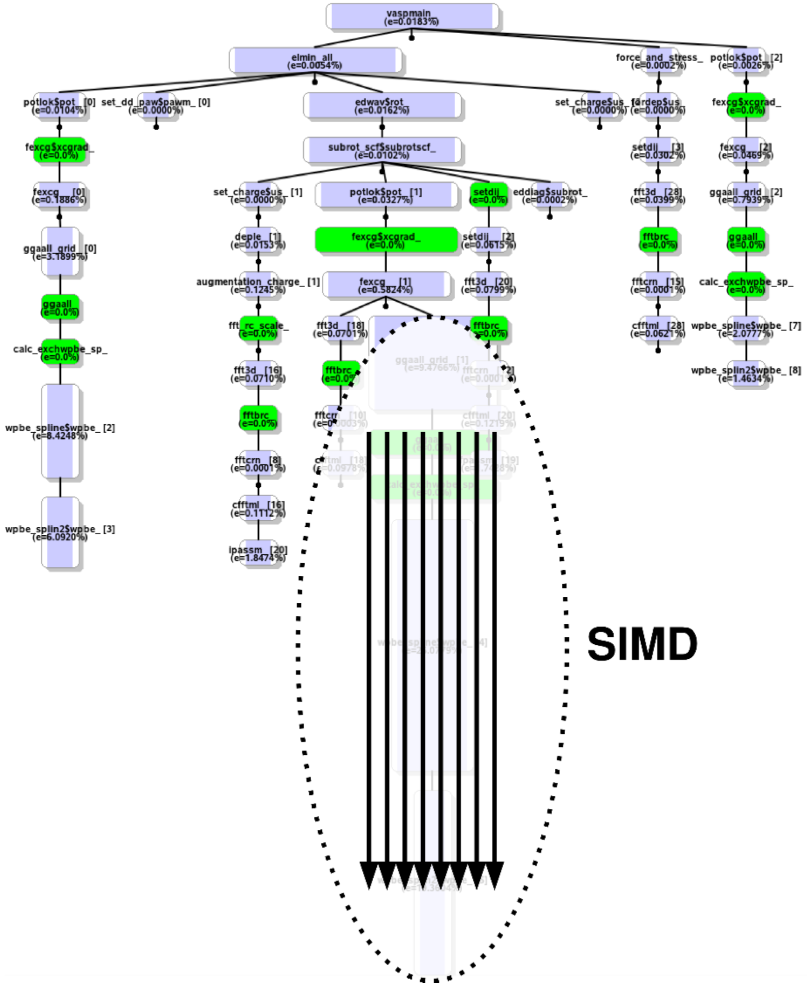
- ❖ Suitable for investigations on SIMD functions and coding strategies in VASP (and other apps)
- ❖ ~40% of total time spent in a single but complex hotspot loop: study Xeon Phi offloading and OpenMP 4.0 SIMD functionality

Enhanced explicit vectorization in VASP

SIMD functions in VASP

- Hotspot loop: nested function calls with almost no vectorization potential within the functions → vectorization along the function calling tree





Enhanced explicit vectorization in VASP

Challenges/optimizations:

1. Debugging the code: At first, explicit vectorization scheme gave wrong results. How to find the error without the SIMD version of the code at hand?
2. Gather operation within one of the SIMD functions: we serialize the gather.

```
subroutine vfoo(…)
  …
!$omp simd
  do ii=0,VL-1
    < code block A >
    < gather operation >
    < code block B >
  enddo
  …
```

* *some variables in* A *need to be stored when also used in* B

```
subroutine vfoo(…)
  …
!$omp simd
  do ii=0,VL-1
    < code block A* >
  enddo
!dir$ novector
  do ii=0,VL-1
    < gather operation >
  enddo
!$omp simd
  do ii=0,VL-1
    < code block B* >
  enddo
  …
```

Alternatively, with OpenMP 4.0 SIMD-enabled function scheme (not tested yet):

Place gather operation inside a non-SIMD function. The call then will be serialized within other SIMD functions.

Enhanced explicit vectorization in VASP

Challenges/optimizations (cnt.):

3. Per-SIMD-lane array expansion

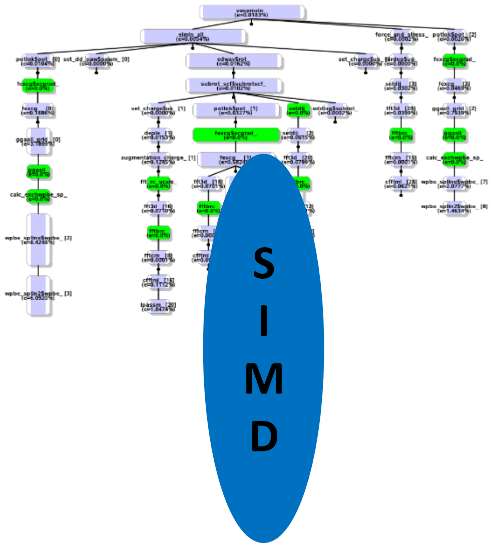
```
subroutine foo(…)
  real*8::a(6)
  …
  do k=1,6
    a(k)=…
  enddo
  …
  call bar(…,a)
  …
```

```
subroutine vfoo(…)
  use simd
  type(simd_real8)::a(6)
  …
  do k=1,6
!$omp simd
    do ii=0,VL-1
      a(k)%x(ii)=…
    enddo
  enddo
  …
  call vbar(…,a)
  …
```

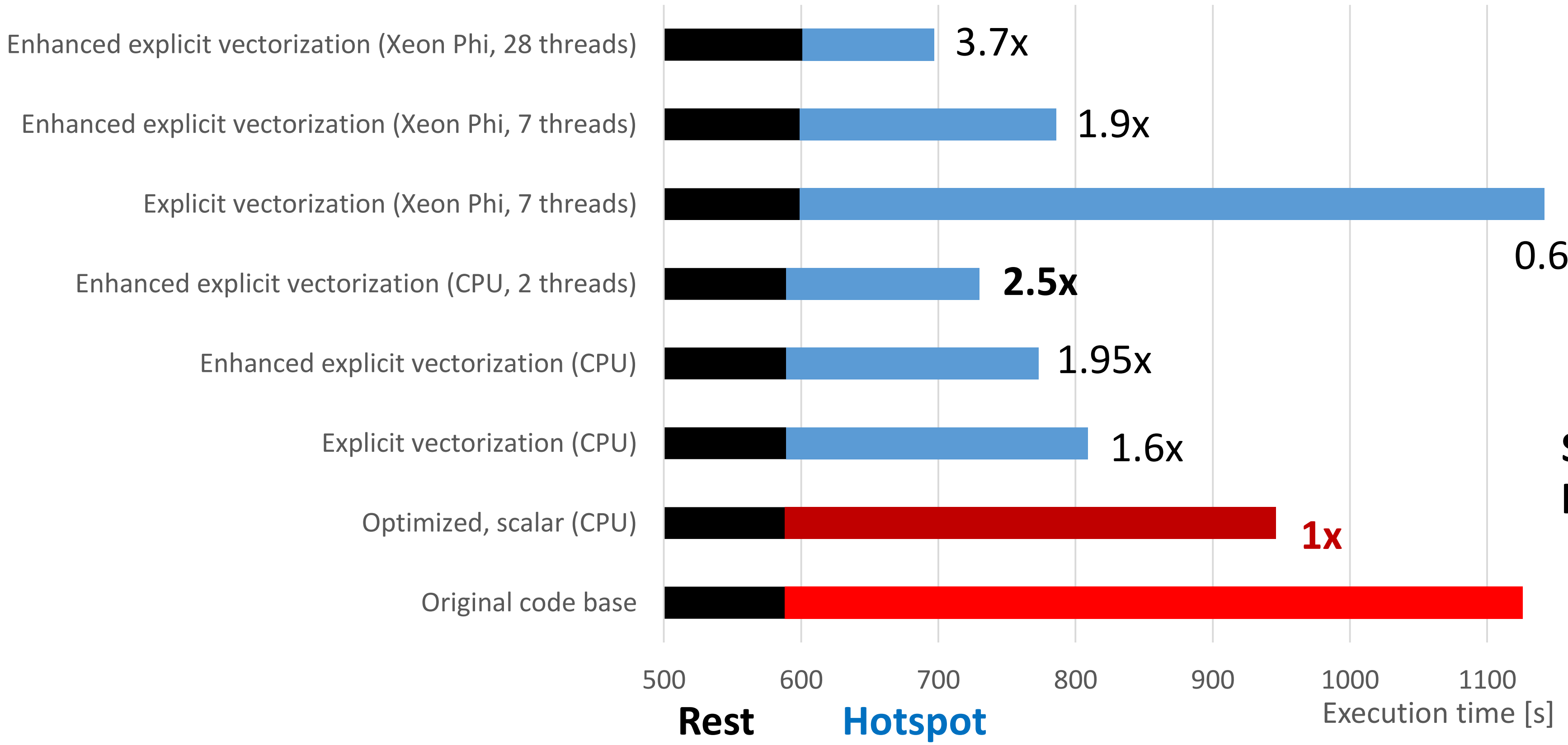
No problem with the enhanced explicit vectorization scheme.
The current Intel Compiler seems to have a problem here when using OpenMP 4.0 SIMD functions. The issue is known to Intel.

Enhanced explicit vectorization in VASP

Some performance results: Octanol in large volume



Xeon E5-2670v2 [+ Xeon Phi 5120D], 8 MPI ranks



Speedups are for the hotspot part only!

Wrap up

Wrap up

SIMD vectorization is one of the key challenges when writing code for modern processors with SIMD functional units

- Portability: **use OpenMP 4.0** for CPU and Xeon Phi programming.
  What about GPGPUs?
- Performance: if your code does not vectorize or shows poor SIMD performance due to complex coding patterns and/or irregularities
  - try to simplify or reformulate your algorithm
  - **try using vector data types to tell the compiler what to do exactly** (maybe you already have an idea): you can stick with OpenMP 4.0 directives as addressed in this webinar
- Correctness: Everything around SIMD directives is really advanced compiler magic and you never get in touch with that. But what if you need to in order to fix errors? Using vector data types may help, as you then have the vector code at hand.

Code samples are available online!
https://github.com/flwende/simd_webinar

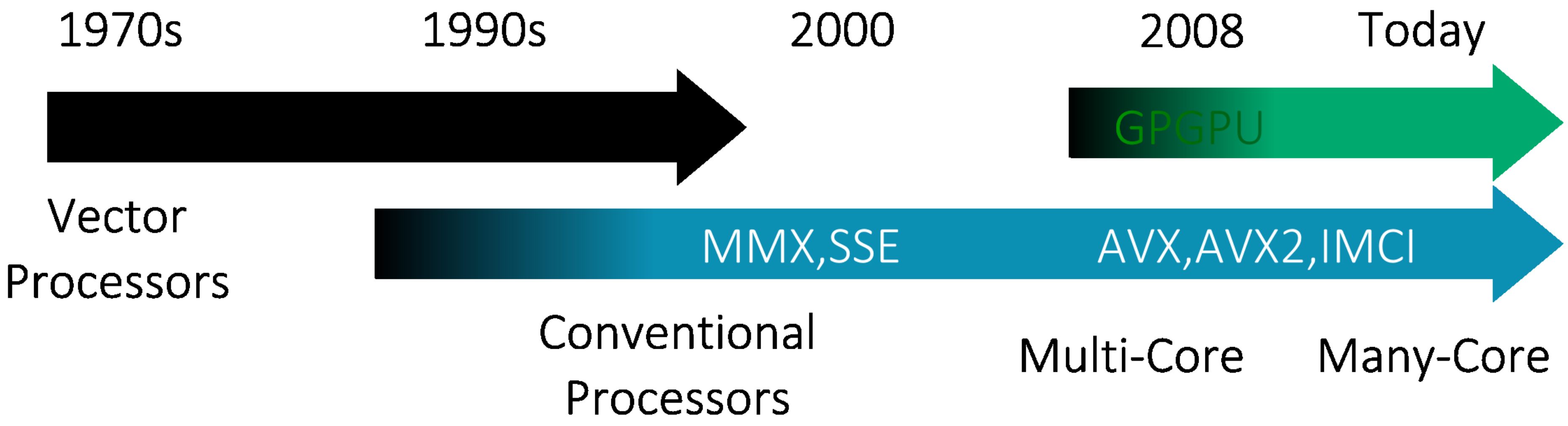
Thanks

Backup

Vector execution is nothing totally new!

Not so long ago, codes have been heavily optimized for vector processing.
Today: revival of vector processing as a 3rd layer of parallelism.

Multiprocessing [+ Multithreading [+ Vector]]



What can be vectorized?

Candidates: loops & code blocks applying the same OP to different data.

However, data dependencies might prevent vectorization:

- ❖ Write-after-read (anti dependence)
- ❖ Write-after-write (output dependence)
- ❖ Read-after-write (flow dependence)

```
do i=8,N
  a(i)=a(i+X)+C
```

SIMD execution with vector length (VL) 2

```
a(8)  =a(8  +X)+C
a(9)  =a(9  +X)+C
a(10)=a(10+X)+C
a(11)=a(11+X)+C
a(12)=a(12+X)+C
a(13)=a(13+X)+C
a(14)=a(14+X)+C
a(15)=a(15+X)+C
a(16)=a(16+X)+C
…
```

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

What can be vectorized?

Candidates: loops & code blocks applying the same OP to different data.

However, data dependencies might prevent vectorization:

- Write-after-read (anti dependence)
- Write-after-write (output dependence)
- Read-after-write (flow dependence)

```
do i=8,N
  a(i)=a(i+X)+C
```

SIMD execution with vector length (VL) 2

```
a(8)  =a(8  +X)+C
a(9)  =a(9  +X)+C
a(10)=a(10+X)+C
a(11)=a(11+X)+C
a(12)=a(12+X)+C
a(13)=a(13+X)+C
a(14)=a(14+X)+C
a(15)=a(15+X)+C
a(16)=a(16+X)+C
…
```

X=-1

Read-after-write dependence

```
a(8)  =a(8  -1)+C
a(9)  =a(9  -1)+C
a(10)=a(10-1)+C
a(11)=a(11-1)+C
a(12)=a(12-1)+C
a(13)=a(13-1)+C
a(14)=a(14-1)+C
a(15)=a(15-1)+C
a(16)=a(16-1)+C
…
```

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

What can be vectorized?

Candidates: loops & code blocks applying the same OP to different data.

However, data dependencies might prevent vectorization:

- ❖ read-after-write (flow dependence)
- ❖ write-after-read (anti dependence)
- ❖ write-after-write (output dependence)

```
do i=2,N
  a(i)=a(i+X)+C
```

pseudo vector version: VL 2

```
do i=2,N,2
  {a(i),a(i+1)}={a(i+X),a(i+X+1)}+{C,C}
```

Is it vectorizable?

Case 1: $X < 0$ and $|X| < VL$
read-after-write dependence
not/partially vectorizable

Case 2: $X > 0$ and $X < VL$
write-after-read dependence
vectorizable

Case 3: $|X| \geq VL$
no real dependence
vectorizable

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

What can be vectorized?

Read-after-write dependence (example cnt.): `X` = -1, `C` = 1

```
do i=2,N
  a(i)=a(i-1)+1
```

vector version: VL 2

```
do i=2,N,2
  {a(i),a(i+1)}={a(i-1),a(i)}+{1,1}
```

The old a(i) value is assigned here.
FLOW DEPENDENCE (read-after-write)

n	a(n) input	a(n) scalar	a(n) vector
1	2	2	2
2	5	3	3
3	4	4	6
4	2	5	7
5	9	6	3
…	…	…	…

Compiler will not vectorize this loop automatically:

```
…remark: loop was not vectorized: existence of vector dependence
…remark: vector dependence: assumed FLOW dependence…
```

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

What can be vectorized?

Write-after-write dependency:

```
do i=2,N
   a(i-1)=x(i)
   …
   a(i)=3.0*i
```

vector version: VL 2

```
do i=2,N,2
   {a(i-1),a(i)}={x(i),a(i+1)}
   …
   {a(i),a(i+1)}={3.0*i,3.0*(i+1)}
```

n	a(n) input	a(n) scalar	a(n) vector
1	2	x(2)	x(2)
2	5	x(3)	3.0*2
3	4	x(4)	x(4)
4	7	x(5)	3.0*4
…	…	…	…
N-1	5	x(N)	3.0*(N-1)
N	9	3.0*N	3.0*N

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

Explicit vectorization with OpenMP 4.0

SIMD loop

- `safelen(length)`: Maximum distance of successive SIMD iterations.
  Consider this loop on an AVX system with `a(:)` being of type `real*8`:

```
!$omp simd
do i=4,n
  a(i)=2.0*a(i-3)
```

Wrong results!

```
!$omp simd safelen(3)
 do i=4,n
   a(i)=2.0*a(i-3)
```

Correct results!

- `aligned(list[:alignment])`: Additional information for the compiler regarding alignment of data.

SIMD function

- `simdlen(length)`: Number of concurrent elements packed for vector arguments.
- `linear(list[:linear-step])`: Linear relationship w.r.t. iteration space for a list of variables.
  The parameter referenced in a linear-step must be subject of a `uniform` clause.

```
!$omp simd
do i=1, n
  call foo(x(1),y(1),i)
```

```
subroutine foo(x,y,i)
!$omp declare simd(foo) simdlen(4) uniform(x,y) linear(i:1)
  real*8::x(*),y(*)
  integer::i
  y(i)=2.0*x(i)
```

Xinmin Tian, Bronis R. de Supinski, *Explicit Vector Programming with OpenMP 4.0 SIMD Extensions*, Primeur Magazine, 2014