Vectorisation

JOACHIM HEIN, LUNARC & CENTRE OF MATHEMATICAL SCIENCES

PEDRO OJEDA MAY, HPC2N, UMEÅ UNIVERSITY

LUNARC

HPC2N, UmU

Overview

- Vector registers
- SIMD construct
- Declare SIMD construct to vectorise functions

Vectorisation

HPC2N, UmU

Modern hardware has wide registers Overview on x86 system

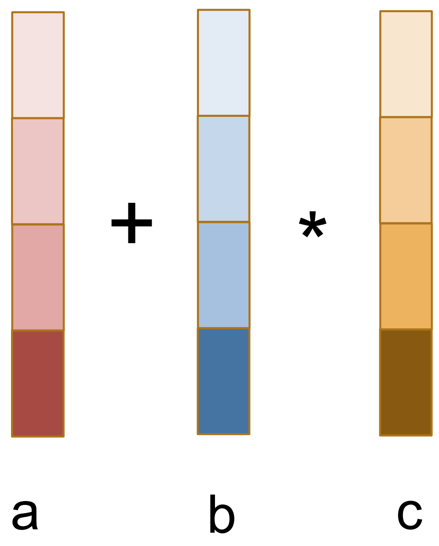
Instruction set	Register width	Single prec. words	Double prec. words	Typical hardware
SSE, SSE2	128 bit	4	2	modern x86
AVX, AVX2	256 bit	8	4	x86 since 2011
AVX-512	512 bit	16	8	Skylake Knights Landing

- Concept also exists in non-x86 hardware, examples:
  - ARM: NEON
  - IBM Power: VSX

Example: AVX2 FMA instruction

- AVX: 256 bit registers - 4 doubles
- Single instruction - 8 flops:

$$a_1 + b_1 * c_1$$
$$a_2 + b_2 * c_2$$
$$a_3 + b_3 * c_3$$
$$a_4 + b_4 * c_4$$



- Enable via compiler option:
  - Without cross compilation:
    - » GCC: `-march=native -O3`
    - » Intel: `-xHost -O3`
  - Cross compilation: explicit specification

LUND UNIVERSITY

Basic example for SIMD deployment

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

- Execute multiple loop iterations simultaneously
- Reduce loop count accordingly
- Iterations need to be independent

LUND UNIVERSITY

What needs to be done for SIMD (Simplified)

```
do i=1, n, 4
   a(i  ) = b(i  ) + c(i  )
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
enddo
```

- Execute multiple loop iterations simultaneously
- Iterations need to be independent
- Compiler might need to add a peel

LUND UNIVERSITY

Basic example for SIMD deployment

```
for (i=1; i<n; i++)
  {
   a[i] = b[i] + c[i]
  }
```

- Execute multiple loop iterations simultaneously
- Reduce loop count accordingly
- Iterations need to be independent

What needs to be done for SIMD (Simplified)

```
for (i=0; i<n; i+=4)
  {
   a[i  ] = b[i  ] + c[i  ];
   a[i+1] = b[i+1] + c[i+1];
   a[i+2] = b[i+2] + c[i+2];
   a[i+3] = b[i+3] + c[i+3];
  }
```

- Execute multiple loop iterations simultaneously
- Reduce loop count accordingly
- Compiler might need to add a peel

LUND UNIVERSITY

Automatic vectorisation

- Modern compilers vectorise many loops automatically
  - Choose right instruction set and optimisation level
    - GNU: `-O3 -march=native`
    - Intel: `-O3 –xHost`
  - Compilers can report on vectorisation
    - GNU: `-fopt-info-vec -fopt-info-vec-missed`
    - Intel: `-qopt-report -qopt-report-phase=vec`
- Compiler needs help in complex situations
  - OpenMP SIMD construct: portable way to help

SIMD construct

HPC2N, UmU

simd construct (Fortran)

- Assure compiler that the following loop can be vectorised

```
!$omp simd
do i = 1, n
   a(i) = a(i) + b(i)
enddo
```

LUND UNIVERSITY

simd construct in C

- Assure compiler that the following loop can be vectorised

```
#pragma omp simd
for (i = 1; i < n; i++)
  {
    a[i] = a[i] + b[i];
  }
```

Clauses for `simd` construct

- Data sharing:

  `private, lastprivate, reduction`

- The is no `default(none)` here!

- Number of loops associated with construct

  `collapse(n)`

Clauses for `simd` construct (cont.)

- Clause `safelen` allows vectorisation of certain dependencies

```
!$omp simd safelen(7)
do i = 1, n
   a(i) = a(i) + a(i+7)
enddo
```

- Allowed to load up to 7 values in the register
- This would be difficult to parallelise

LUND UNIVERSITY

Clauses for `simd` construct (cont.)

- Clause `safelen` allows vectorisation of certain dependencies

```
#pragma omp simd safelen(7)
for (i = 1; i < n; i++)
   a[i] = a[i] + a[i+7];
```

- Allowed to load up to 7 values in the register
- This would be difficult to parallelise

LUND UNIVERSITY

Clauses `simdlen`

- Clause `simdlen`: <u>preferred</u> number of consecutive iterations:

```
!$omp simd simdlen(4)
do i = 1, n
  a(i) = a(i) + b(i)
enddo
```

- This will suggests to do 4 iterations simultaneously

Clauses `simdlen`

- Clause `simdlen`: preferred number of consecutive iterations:

```
#pragma omp simd simdlen(4)
for (i = 1; i < n; i++)
  {
    a[i] = a[i] + b[i];
  }
```

- This suggests to do 4 iterations simultaneously

Clause `linear`

- Declare a linear relationship between iteration (≠ loop index) and a variable

```
j=0
!$omp simd linear(j:2)
do i = 1, N, 3
  j = j + 2
  a(i) = b(j)
enddo
```

- Data sharing clause, `j` is now private

LUND UNIVERSITY

Clause `linear`

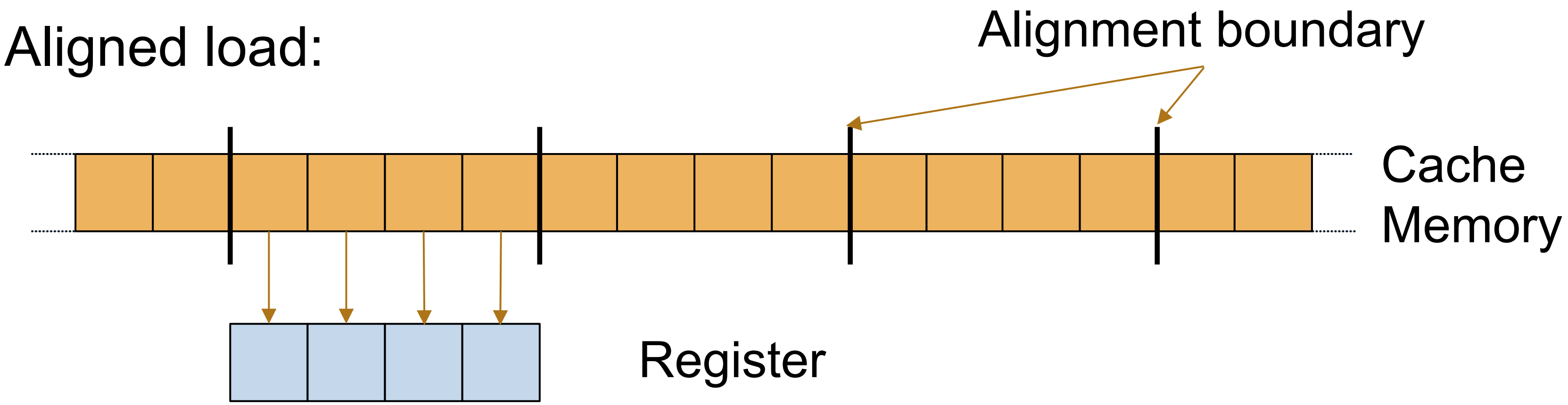
- Declare a linear relationship between iteration (≠ loop index) and a variable

```
j=0;
#pragma omp simd linear(j:2)
for (i = 1; i < n; i++)
  {
     j = j + 2;
     a[i] = b[j];
  }
```

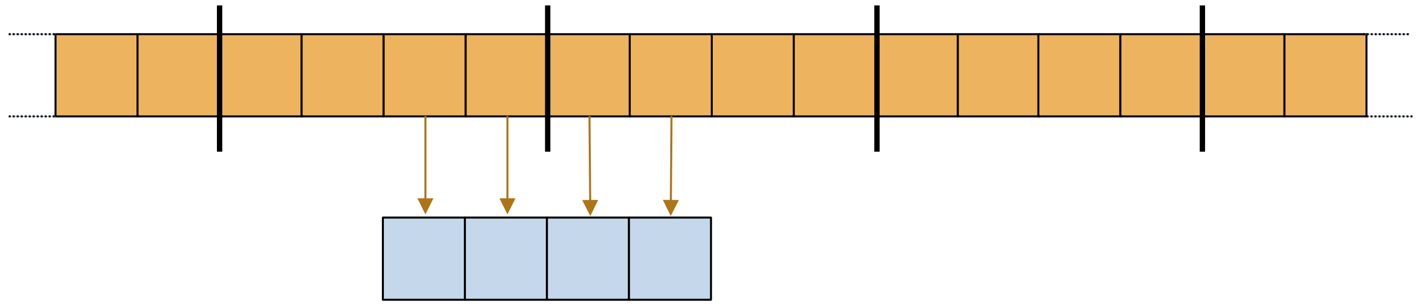
- Data sharing clause, `j` is now private

SIMD and data alignment

- Aligned load:



- Un-aligned load (typically not supported)



SIMD and data alignment

- Byte alignment of data
  - An address is e.g. 32-byte-aligned if:

    byte-address is divisible by 32
- SIMD loads/stores typically have alignment requirements
  - Unaligned data needs either
    - » Peel loops – load individual number until boundary
    - » Crossload
  - Checking requires extra code – performance impact

Aligning data

- There is no function in OpenMP ☹
- In C one might want to lib-functions from stdlib:

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc( size_t alignment, size_t size );
```

- In Fortran
  - Wrap `posix_memalign` or `aligned_alloc`
  - Compiler specific tools:
    - » Example: Intel compiler

      ```
      ifort -align array32byte
      ```

      aligns all arrays at 32-byte boundaries

Advanced optimisation: `aligned`

- If you understand/control your data alignment
- Declare it to the system (e.g. 32 byte alignment)

```
!$omp simd aligned(x,32)
Do i = 1, N
   x(i) = 2.0D0 * x(i)
enddo
```

- If correct, this will reduce overheads (peeled loops)
- If false, illegal instructions
- No optional parameter – implementation default alignment

LUND UNIVERSITY

Advanced optimisation: `aligned`

- If you understand/control your data alignment
- Declare it to the system (e.g. 32 byte alignment)

```
#pragma omp simd aligned(x,32)
for (i = 1; i<N; i++)
   x[i] = 2.0D0 * x[i];
```

- If correct, this will reduce overheads (peeled loops)
- If false, illegal instructions
- No optional parameter – implementation default alignment

Vectorisation of functions and subroutine

LUND UNIVERSITY

LUNARC

HPC2N, UmU

Function/subroutine calls in loop

- Try to avoid calls in loop, due to performance impact
- Use `declare simd` to create vector versions of functions and subroutines

```
function addfunc(a,b)
!$omp declare simd(addfunc)
   implicit none
   double precision :: a, b, addfunc
   addfunc = a + b
end function addfunc
```

Using a simd-ised function

- The function can be used in a `simd` loop

```
!$omp simd
do i = 1, N
   c(i) = addfunc(a(i), b(i))
enddo
```

LUND UNIVERSITY

Function/subroutine calls in loop

- Try to avoid calls in loop, due to performance impact
- Use `declare simd` to create vector versions
- Required in the header **and** source file!

```
#pragma omp declare simd
double addfunc(double a, double b) {
   double r;
   r = a + b;
   return r;
}
```

Using a simd-ised function

- The function can be used in a `simd` loop

```
#pragma omp simd
for (i = 1; i < N; i++)  {
   c[i] = addfunc(a[i], b[i])
}
```

Clauses for `declare simd`

- Discussed before:
  - `simdlen(`*`length`*`)`
  - `linear(`*`linear-list[ : linear-step]`*`)`
  - `aligned(`*`arg-list[ : alignment]`*`)`
- Specific clauses
  - `uniform(`*`argument-list`*`)`

    value invariant for all invocations
  - `inbranch`

    always called inside a conditional statement
  - `notinbranch`

    never called inside a conditional statement

LUND UNIVERSITY

Example 1

```
function vecop(a, b, i, offset)
!$omp declare simd(vecop) uniform(a,b,offset) &
!$omp linear(i:1)
   integer :: i
   double precision :: a(*), b(*), offset, vecop
   vecop = a(i) + b(i) + offset
end function vecop
```

- GCC 6.3, 8.3, 9.2 and clang 9.0.1 will not compile this!
  - Complain about a and b in `uniform`
  - Possible workaround: scalar code
- Intel does compile this

Using function vecop

```
double precision :: a(N), b(N)
...
!$omp simd
Do i = 1, N
  a(i) = vecop(a, b, i, 3.1d0)
enddo
```

Example 2

```
function cube(x)
!$omp declare simd inbranch
   implicit none
   double precision :: cube, x
   cube = x*x*x
end function cube
```

- This will generate code with a mask and suppress the unmasked version

Using the cube function

```
double precision :: x(N), y(N)
...
!$omp simd simdlen(4)
do i = 1, N
  if( y(i).gt.0.0d0 ) then
     y(i) = cube(x(i))
  endif
enddo
```

- Will operate on 4 long vectors, but not do the operation on lane, where condition is false (applying masking)

LUND UNIVERSITY

Combined contructs

- Distribute a loop and simd-ise it:

```
!$omp do simd
```

- Start parallel region, distribute a loop and simd-ise it:

```
!$omp parallel do simd
```

- Modified schedule

```
!$omp do simd schedule(simd:static, 11)
```

  – new chunksize = (chunksize/simdlen) * simdlen

  – always a multiple of the simdlen

LUND UNIVERSITY

Summary

- Using the `SIMD` construct to assist in loop vectorisation

- `declare SIMD` construct to allow vectorised function calls

- Some compilers allow for SIMD part of OpenMP only
  - Intel: `-qopenmp-simd`

LUND UNIVERSITY