



EPSRC

# ARCHER Single Node Optimisation

---

Vectorisation

Slides contributed by Cray and EPCC



# Vector Instructions (Vectorisation)

- Modern CPUs can perform multiple operations each cycle
  - Use special SIMD (Single Instruction Multiple Data) instructions
    - e.g. SSE, AVX
  - Operate on a "vector" of data
    - typically 2 or 4 double precision floats (on Ivy Bridge)
  - Potentially gives speedup in floating point operations
  - Usually only one loop is vectorisable in loop nest
    - And most compilers only consider inner loop



- Optimising compilers will use vector instructions
  - Relies on code being vectorisable
  - ...or in a form that the compiler can convert to be vectorisable
  - Some compilers are better at this than others
  - But there are some general guidelines about what is likely to work...



# Requirements for vectorisation

- Loops must have determinable (at run time) trip count
  - rules out most while loops
- Loops must not contain function/subroutine calls
  - unless the call can be inlined by the compiler
  - maths library functions usually OK
- Loops must not contain branches or jumps
  - guarded assignments may be OK
  - e.g. `if (a[i] != 0.0) b[i] = c * a[i];`
- Loop trip counts needs to be long, or else a multiple of the vector length



- Loops must not have dependencies between iterations
  - reductions usually OK, e.g. `sum += a[i];`
  - avoid induction variables e.g. `indx += 3;`
  - use `restrict`
  - may need to tell the compiler if it can't work it out for itself
- Aligned data is best
  - e.g. AVX vector loads/stores operate most effectively on 32-bytes aligned address
  - need to either let the compiler align the data....
  - ..or tell it what the alignment is
- Unit stride through memory is best



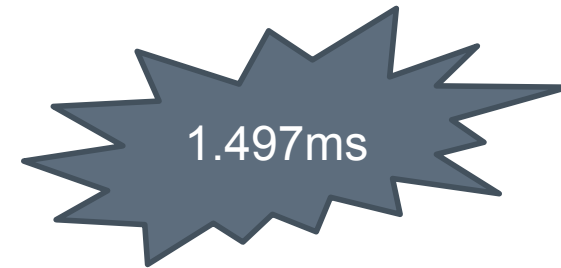
# Did my loop get vectorised?

- Always check the compiler output to see what it did
  - CCE: `-hlist=a`
  - GNU: `-ftree-vectorizer-verbose=1`
  - PGI: `-Minfo`
  - or (for the hard core) check the assembler generated
- Clues from CrayPAT's HWPC measurements
  - `export PAT_RT_HWPC=13 or 14 # Floating point operations SP,DP`
  - Complicated, but look for ratio of operations/instructions > 1
    - expect 4 for pure AVX with double precision floats



# Example

```
16. + 1-----<   do j = 1,N
17.   1             x = xinit
18. + 1 r4-----<   do i = 1,N
19.   1 r4         x = x + vexpr(i,j)
20.   1 r4         y(i) = y(i) + x
21.   1 r4----->   end do
22.   1----->     end do
```



1.497ms

**ftn-6254** ftn: VECTOR File = bufpack.F90, Line = 16

A loop starting at line 16 was **not vectorized** because a recurrence was found on "y" at line 20.

**ftn-6005** ftn: SCALAR File = bufpack.F90, Line = 18

A loop starting at line 18 was **unrolled 4 times**.

**ftn-6254** ftn: VECTOR File = bufpack.F90, Line = 18

A loop starting at line 18 was not vectorized because a recurrence was found on "x" at line 19.



```

38.   Vf-----<   do i = 1,N
39.   Vf              x(i) = xinit
40.   Vf----->   end do
41.
42.   ir4-----<   do j = 1,N
43.   ir4 if--<     do i = 1,N
44.   ir4 if          x(i) = x(i) + vexpr(i,j)
45.   ir4 if          y(i) = y(i) + x(i)
46.   ir4 if-->     end do
47.   ir4----->   end do

```

**x promoted to vector:**  
trade slightly more memory  
for better performance

1.089ms

-37%

**ftn-6007** ftn: SCALAR File = bufpack.F90, Line = 42

A loop starting at line 42 was **interchanged** with the loop starting at line 43.

**ftn-6004** ftn: SCALAR File = bufpack.F90, Line = 43

A loop starting at line 43 was **fused** with the loop starting at line 38.

**ftn-6204** ftn: VECTOR File = bufpack.F90, Line = 38

A loop starting at line 38 was **vectorized**.

**ftn-6208** ftn: VECTOR File = bufpack.F90, Line = 42

A loop starting at line 42 was **vectorized** as part of the loop starting at line 38.

**ftn-6005** ftn: SCALAR File = bufpack.F90, Line = 42

A loop starting at line 42 was **unrolled 4 times**.



# OpenMP 4.0 SIMD directives

- Many compilers support their own sets of directives to assist the compiler to vectorise loops.
  - useful but not portable
- OpenMP 4.0 contains a standardised set of directives
- Currently supported by Intel and GNU compilers on ARCHER
  - Cray coming soon.....



# Portable SIMD directives

- Use **simd** directive to indicate a loop should be vectorised

```
#pragma omp simd [clauses]
```

or

```
!$omp side [clauses]
```

- Executes iterations of following loop in SIMD chunks
- Loop is *not* divided across threads
- SIMD chunk is set of iterations executed concurrently by SIMD lanes
- Not a hint! Programmer is asserting independence of iterations.



- Clauses control data environment, how loop is partitioned
- **safelen(length)** limits the number of iterations in a SIMD chunk.
- **linear** lists variables with a linear relationship to the iteration space (induction variables)
- **aligned** specifies byte alignments of a list of variables
- **private**, **lastprivate**, **reduction** and **collapse** have usual meanings.
- Also **declare simd** directive to generate SIMDised versions of functions.
- Can be combined with loop constructs (parallelise and vectorise)
  - **#pragma omp for simd**

