ARCHER Single Node Optimisation

Vectorisation

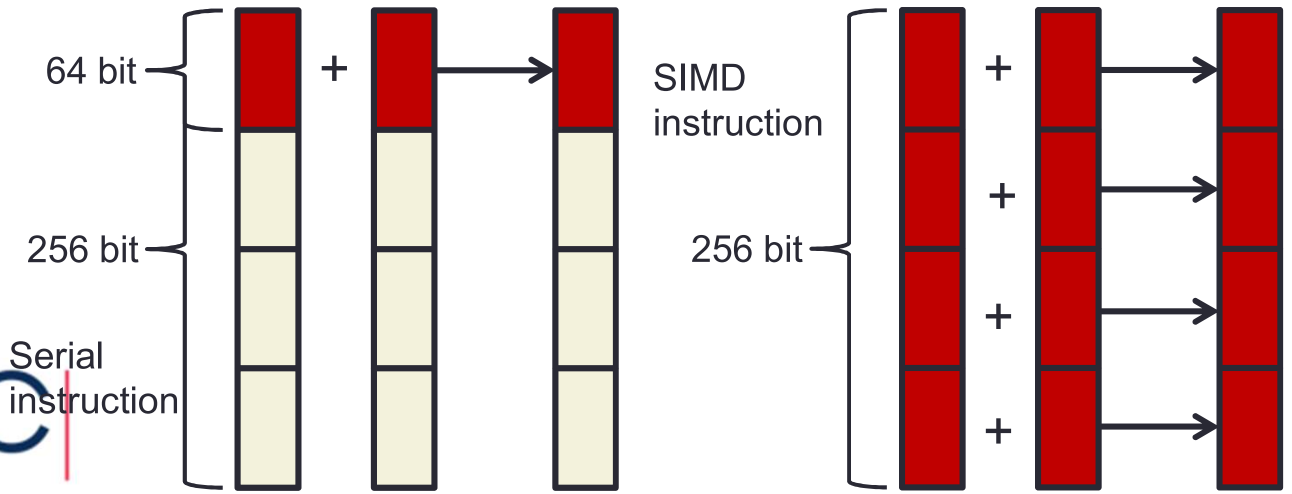
Vector Instructions (Vectorisation)

- 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

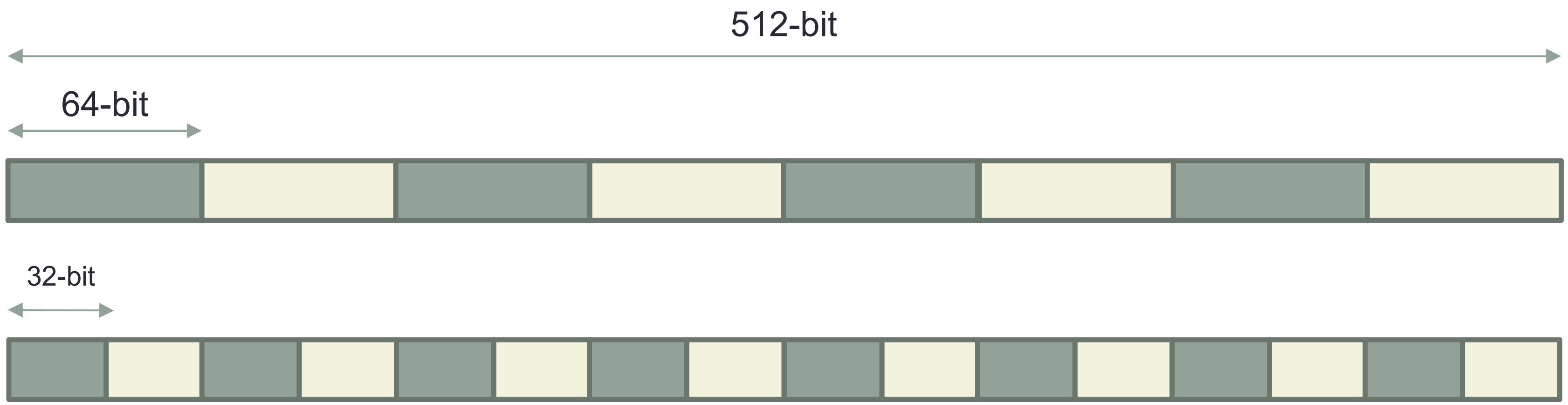
Vectorisation

- Same operation on multiple data items
  - Wide registers
  - SIMD needed to approach FLOP peak performance, but your code must be capable of vectorisation
- **x86 SIMD instruction sets:**
  - SSE: register width = 128 Bit
    - 2 double precision floating point operands
  - AVX: register width = 256 Bit
    - 4 double precision floating point operands

```
for(i=0;i<N;i++){
    a[i] = b[i] + c[i]
}
do i=1,N
    a(i) = b(i) + c(i)
end do
```



Intel AVX512



- Skylake processor has AVX512 vector units per core
  - Symmetrical units
    - Only one supports some of the legacy stuff (x87, MMX, some of the SSE stuff)
  - Vector instructions have a latency of 6 cycles

- 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...

When does the compiler vectorize

- What can be vectorized
  - Only loops
- Usually only one loop is vectorisable in loopnest
  - 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
- Check the compiler output listing and/or assembler listing
  - Look for packed AVX/AVX2/AVX512 instructions

  i.e. Instructions using registers `zmm0-zmm31` (512-bit) `ymm0-ymm31` (256-bit) `xmm0-xmm31` (128-bit)

  Instructions like `vaddps`, `vmulps`, etc...

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 no 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

Compilers

- Intel compiler requires
  - Optimisation enabled (generally is by default)
    - `-O2`
  - To know what hardware it's compiling for
    - `-xCORE-AVX512`
    - This is added automatically for you on ARCHER
  - Can disable vectorisation
    - `-no-vec`
    - Useful for checking performance
  - Intel compiler will provide vectorisation information
    - `-qopt-report=[n]` (i.e. `-qopt-report=5`)
  - Other compilers information
    - Cray: `-hlist=a`
    - GNU: `-fdump-tree-vect-all=<filename>`

Did my loop get vectorised?

- Always check the compiler output to see what it did
  - CCE: `-hlist=a`
  - GNU: `-fdump-tree-vect-all=<filename>`
  - Intel: `-opt-report3`
  - or (for the hard core) check the assembler generated
    - Look to see which registers are in use.
- 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

Did my loop get vectorised?

- GNU offers other options for checking:
- `-fopt-info`
- `-O3 -fopt-info-missed=missed.all`
- `-O2 -ftree-vectorize -fopt-info-vec-missed`
- `-fopt-info-loop-optimized`

Helping vectorisation

- Does the loop have dependencies?
  - information carried between iterations
    - e.g. `counter: total = total + a(i)`
  - No:
    - Tell the compiler that it is safe to vectorise
  - Yes:
    - Rewrite code to use algorithm without dependencies, e.g.
      - promote loop scalars to vectors (single dimension array)
      - use calculated values (based on loop index) rather than iterated counters, e.g.
        - Replace: `count = count + 2; a(count) = ...`
        - By: `a(2*i) = ...`
    - move `if` statements outside the inner loop
    - may need temporary vectors to do this (otherwise use masking operations)
- Is there a good reason for this?
  - There is an overhead in setting up vectorisation; maybe it's not worth it
    - Could you unroll inner (or outer) loop to provide more work?

Vectorisation example

- Compiler cannot easily vectorise:
  - Loops with pointers
  - Non-unit stride loops
  - Funny memory patterns
  - Unaligned data accesses
  - Conditionals/Function calls in loops
  - Data dependencies between loop iterations
  - ….

```
int *loop_size;
void problem_function(float *data1, float *data2, float *data3, int
*index){
  int i,j;
  for(i=0;i<*loop_size;i++){
    j = index[i];
    data1[j] = data2[i] * data3[i];
  }
}
```

Vectorisation example

- Can help compiler
  - Tell it loops are independent
    - `#pragma ivdep`
    - `!dir$ ivdep`
  - Tell it that variables or arrays are unique
    - `restrict`
  - Align arrays to cache line boundaries
  - Tell the compiler the arrays are aligned
  - Make loop sizes explicit to the compiler
    - Ensure loops are big enough to vectorise

```
int *loop_size;
void problem_function(float * restrict data1, float * restrict data2, float
* restrict data3, int * restrict index){
  int i,j,n;
  n = *loop_size;
  #pragma ivdep
  for(i=0;i<n;i++){
    j = index[i];
    data1[j] = data2[i] * data3[i];
  }
}
```

Vectorisation example

- This loop doesn't vectorise either:

```
do j = 1,N
  x = xinit
  do i = 1,N
    x = x + vexpr(i,j)
    y(i) = y(i) + x
  end do
end do
```

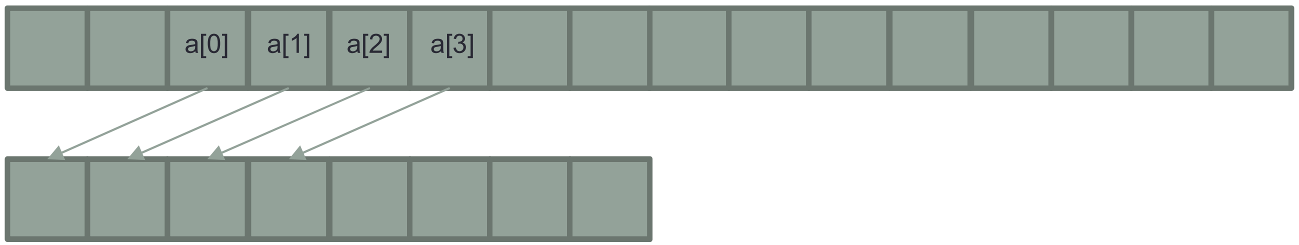
- Compiler will vectorise inner loop by default
  - Dependency on x between loop iterations

```
do j = 1,N
  x(j) = xinit
end do
do j = 1,N
  do i = 1,N
    x(i) = x(i) + vexpr(i,j)
    y(i) = y(i) + x(i)
  end do
end do
```

Data alignment

- When vectorising data aligned data is essential for performance

Cache line



Vector register

- Unaligned data
  - May require multiple data loads, multiple cache lines, multiple instructions
  - Will generate 3 different versions of a loop: peel, kernel, remainder
- Aligned data
  - Minimum number of data loads/cache lines/instructions
  - Will generate 2 different versions of a loop: kernel and remainder

Aligned data

- 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

Align data

- Align on allocate/create (dynamic)
  - `_mm_malloc, _mm_free`

  ```
  float *a = _mm_malloc(1024*sizeof(float),64);
  ```

  - `align attribute` (at definition, not allocation)

  ```
  real, allocatable :: A(1024)
  !dir$ attributes align : 64 :: a
  ```

- Align on definition (static)

```
float a[1024] __attribute__((aligned(64)));
real :: A(1024)
!dir$ attributes align : 64 :: a
```

- Common blocks in Fortran
  - It's not possible to use directives to align data inside a common block
  - Can align the start of a common block

  ```
  !DIR$ ATTRIBUTES ALIGN : 64 :: /common_name/
  ```

  - Up to you to pad elements inside common block
- Derived types
  - May need to use `SEQUENCE` keyword and manually pad to get correct alignment

Multi-dimensional alignment

- Need to be careful with multi-dimensional arrays and alignment
  - If you _mm_malloc each dimension then it should be fine
  - If you do a single dimension _mm_malloc there may be issues:

```
float* a = _mm_malloc(16*15(sizeof(float), 64);
for(i=0;i<16;i++){
#pragma vector aligned
   for(j=0;j<15;j++){
      a[i*15+j]++;
   }
}
```

Inform on alignment

- For non-static data, as well as aligning data, need to tell compiler it is aligned
- Number of different ways to do this
- Alignment of data inside a loop
  - Specify all data in the loop is aligned

```
#pragma vector aligned
!dir$ vector aligned
```

- Alignment of an array
  - Specify, for code after the alignment statement, a specific array is aligned

```
__assume_aligned(a, 64);
!dir$ assume_aligned a: 64
```

- May also need to define to properties of loop scalars

```
__assume(n1%16==0);
for(i=0;i<n;i++){
  x[i] = a[i] + a[i-n1] + a[i+n1];
}
!dir$ assume(mod(n1,16).eq.0)
```

- Also can use OpenMP simd clause
  - Specify array is aligned for simd loop

```
#pragma omp simd aligned(a:64)
!omp$ simd aligned(a:64)
```

Fortran data

- Different ways of passing data to subroutines can affect performance
- Explicit arrays

```
subroutine vec_add_mult(A, B, C)
real, intent(inout), dimension(1024) :: A
real, intent(in), dimension(1024) :: B, C
```

  - Compiler generates subroutine code based on contiguous data
    - Packing/unpacking required to do this is done by the compiler at caller level
    - May be overhead associated with this
  - Need to tell the compiler the arrays are aligned (i.e. `!dir$ assume_aligned` or `!dir$ vector aligned`)
  - Same for arrays where array size is passed as an argument to the routine

Fortran data

- Assumed size arrays

```
subroutine vec_add_mult(A, B, C)
real, intent(inout), dimension(:) :: A
real, intent(in), dimension(:) :: B, C
```

  - Compiler will generate different versions of the code, with and without contiguous functionality
    - Different versions may show up in the vector reports from the compiler
    - If there are too many different potential versions not all of them will necessarily be generated
      - The fall back version (none unit stride, not vectorised) will be used in this case for inputs that don't match any of the other versions
  - Choice which is used made at runtime
  - Still need to tell the compiler the arrays are aligned

Fortran data

- Assumed shape arrays

```
subroutine vec_add_mult(A, B, C)
real, intent(inout), dimension(*) :: A
real, intent(in), dimension(*) :: B, C
```

  - Compiler generates subroutine code based on contiguous data
    - Packing/unpacking required to do this is done by the compiler at caller level
    - May be overhead associated with this
  - Still need to tell the compiler the arrays are aligned

Fortran Indirect addressing

- Indirect addressing code can have some strange affects on vectorisation

```
subroutine vec_add_mult(A, B, C, index)
real, intent(inout), dimension(1024) :: A
real, intent(in), dimension(1024) :: B, C
integer, intent(in), dimension(1024) :: index
integer :: I
```

  - Following has flow dependency (needs `ivdep` directive)

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

  - Uses gather and scatter operations to pack/unpack indexed locations
  - Following creates array temporary for right hand side evaluation

```
a(index(:)) = a(index(:)) + b(index(:)) * c(index(:))
```

  - Ends up creating 2 loops

```
temp(:) = a(index(:)) + b(index(:)) * c(index(:))
a(index(:)) = temp(:)
```

  - Uses gather/scatter in both loops

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

Portable SIMD directives

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

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

or

```
!$omp simd [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`