VECTORISATION

Adrian Jackson

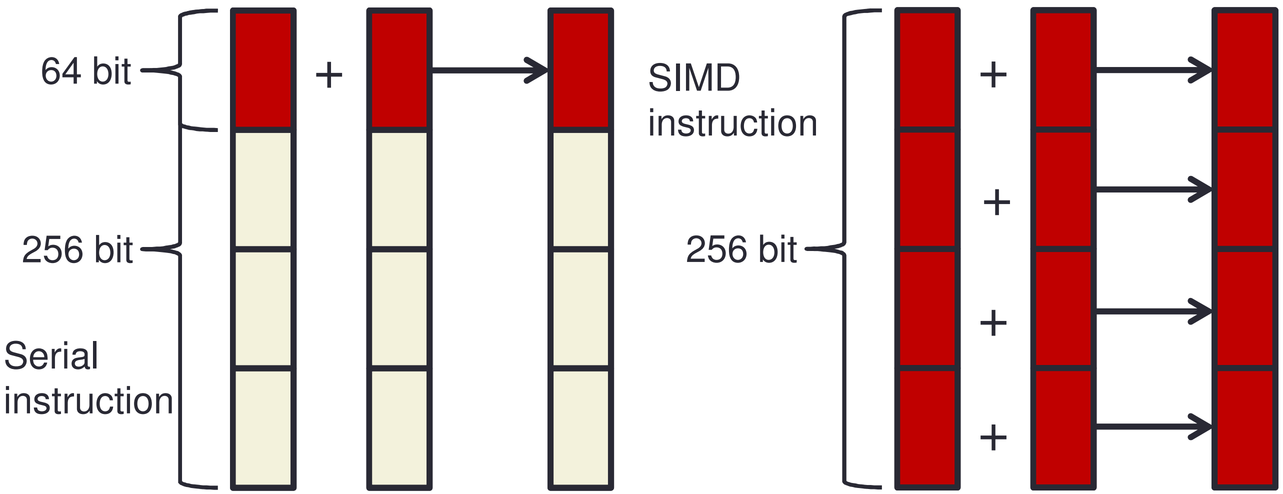
adrianj@epcc.ed.ac.uk

@adrianjhpc

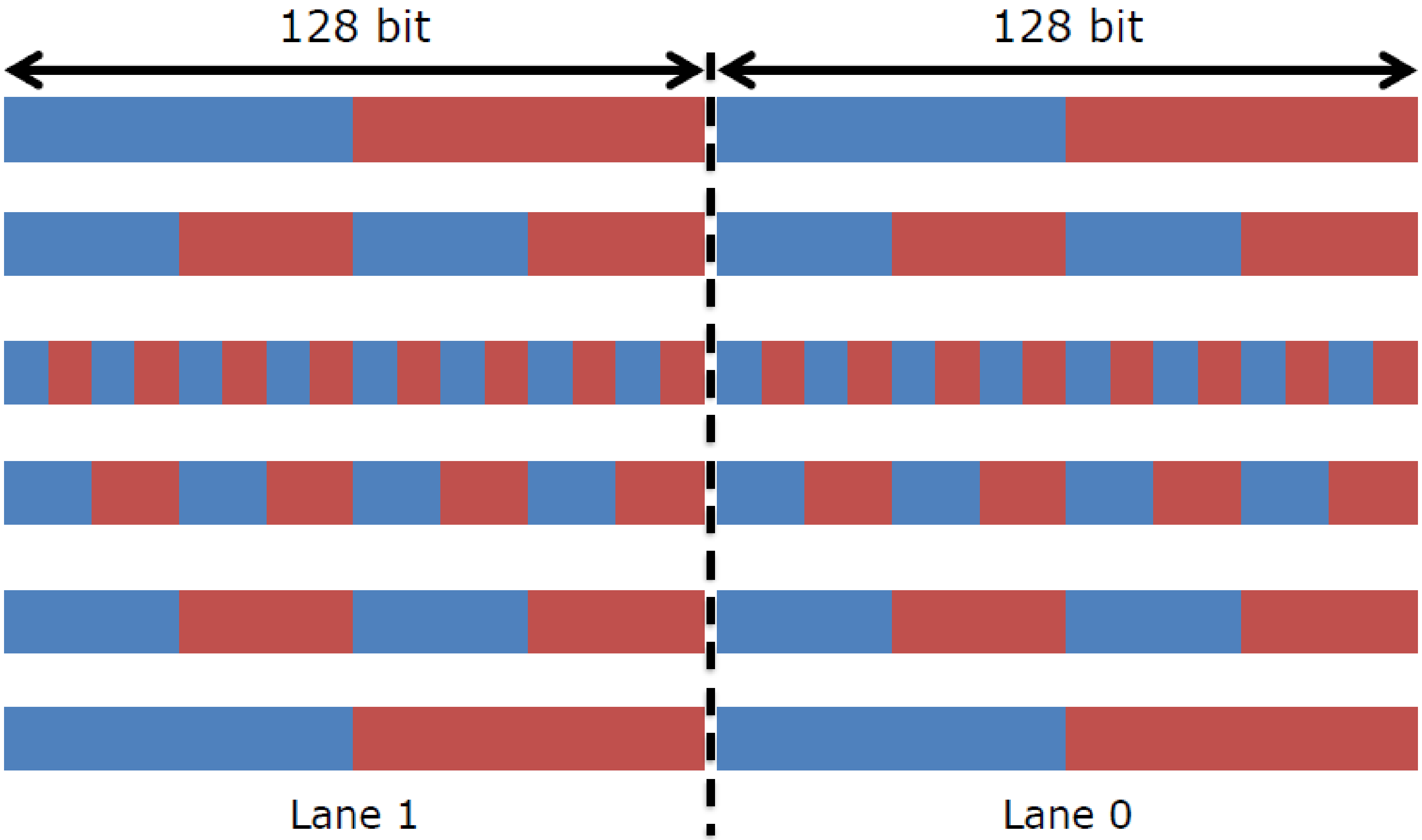
Vectorisation

- 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

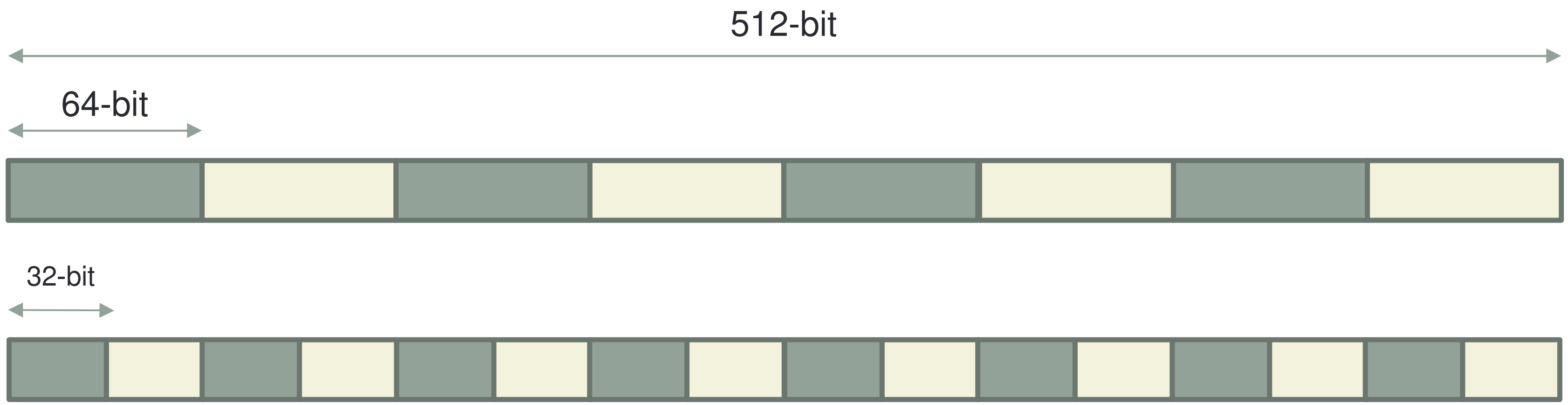
```
for(i=0;i<N;i++){
    a[i] = b[i] + c[i]
}
do i=1,N
    a(i) = b(i) + c(i)
end do
```



Intel AVX/AVX2



Intel AVX512



- KNL processor has 2 x 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 instructions

KNL AVX-512

- AVX512 has extensions to help with vectorisation
- Conflict detection (AVX-512CD)
  - Should improve vectorisation of loops that have dependencies

  `vpconflict` instructions

  - If loops don't have dependencies telling the compile will still improve performance (i.e. `#pragma ivdep`)
- Exponential and reciprocal functions (AVX-512ER)
  - Fast maths functions for transcendental sequences
- Prefetch (AVX-512PF)
  - Gather/Scatter sparse vectors prior to calculation
  - Pack/Unpack

Compiler vs explicit vectorisation

- Compilers will automatically try to vectorise code
  - Implicit vectorisation
  - Can help them to do this
    - Compiler always chooses correctness rather than performance
  - Will often make an automatic decision about when to vectorise

- There are programming constructs/features that let you write explicit vector code
  - Can be less portable/more machine specific
  - Defined code will always be vectorised (even if slower)

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…

Intel compiler

- Intel compiler requires
  - Optimisation enabled (generally is by default)
    - `-O2`
  - To know what hardware it's compiling for
    - `-xMIC-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`)

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
  - None-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:

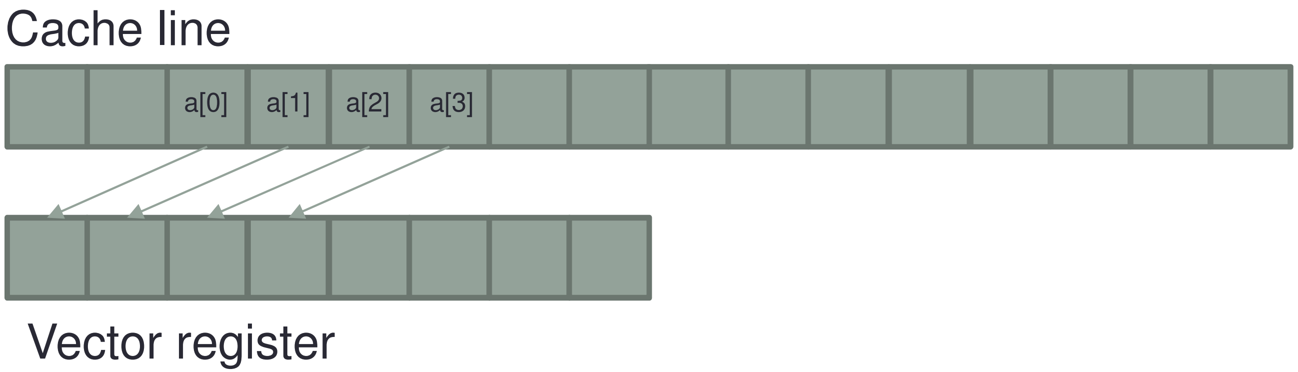
```
do j = 1,N
  x = xinit
  do i = 1,N
    x = x + vexpr(i,j)
    y(i) = y(i) + x
  end do
end do
```

- 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



- 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

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(1024) :: A
real, intent(in), dimension(1024) :: 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

Gathers and Scatters

- If data not accessed in unit stride, may still be vectorised
  - Using vector gather and scatter instructions
  - Pack and unpack registers
- KNL has specialised gather and scatter instructions
  - Improve performance of data load/store (compared to older vectorisation functionality)
  - Still cost more than aligned data
- Vector scalar
  - Possible to vectorise and still be doing scalar calculations
  - Vector operation on a single valid element
  - Compiler reports vectorisation, performance doesn't change

Masking

- Vectorisation is disrupted by conditional statements in loops
- Mask instructions can be used to protect elements that shouldn't be updated based on an if/else construct
  - mask is an integer array that can be compared for non-zero numbers
  - select the vector lanes to run or update

```
for (i = 0; i < N; i++) {
  if (Trigger[i] < Val) {
    A[i] = B[i] + 0.5;
  }
}
```

Blending

- Compiler will try and use more advanced techniques to avoid masking

```
for (i = 0; i < N; i++) {
  if (Trigger[i] < Val) {
    A[i] = B[i] + 0.5;
  }else{
    A[i] = B[i] - 0.5;
  }
}
```

```
for (i = 0; i < N; i+=16) {
  TmpB= B[i:i+15];
  Mask = Trigger[i:i+15] < Val
  TmpA1 = TmpB + 0.5;
  TmpA2 = TmpB - 0.5;
  TmpA = BLEND Mask, TmpA1, TmpA2
  A[i:i+15] = TmpA;
}
```

Explicit vectorisation

- Language features, intrinsics, extensions, etc… let you manually specify vectorisation
  - Override compiler, implement yourself
  - Forces compiler to do it
  - Up to you to make sure it's correct
- OpenMP SIMD directive
- Intel directives
- CilkPlus/Fortran array notation
- Vector intrinsics
  - Not recommend for KNL
    - At least, the intrinsics used for KNC are not expected to give good performance on KNL

OpenMP SIMD directives

- Many compilers support SIMD directives to aid vectorisation of loops.
  - compiler can struggle to generate SIMD code without these
- OpenMP 4.0 provides a standardised set
- Use **`simd`** directive to indicate a loop should be vectorised

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

`!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

OpenMP SIMD clauses

- Clauses control data environment and partitioning
- `safelen(length)` limits the number of iterations in a SIMD chunk.
- `linear(a1,a2,….)` lists variables with a linear relationship to the iteration space (loop variable)
- `aligned(a1:base,…)` specifies byte alignments of a list of variables
- `private`, `lastprivate`, `reduction` specify data scoping of functionality (as per the OpenMP standard)
- `collapse` will combine multiple perfectly nested loops below the directive to give a bigger loop space
- `declare simd` directive to generate SIMDised versions of functions.
- Can be combined with OpenMP loop constructs

SIMD example

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

SIMD function

- Can define functions that can be called from within a vectorised loop
  - Can specify things about the function arguments
- Fortran:

```
!$omp declare simd(name) [clause
[[,clause]…]
function name …
```

- C/C++

```
#pragma omp declare simd [clause
[[,clause]…]
function name ….
```

SIMD function clauses

- **`simdlen(length)`** defines the vector length to be used, must be power of 2
- **`linear(a1,a2,….)`** lists variables with a linear relationship to the iteration space (loop variable)
- **`aligned(a1:base,…)`** specifies byte alignments of a list of variables
- **`uniform(qdata,…)`** declares that arguments aren't vectors (so constant across SIMD lanes
- **`inbranch, notinbranch`** whether function is called in a branch or not

Cilk

- C/C++ extension
  - Provides array and array section operations
  - Similar to Fortran array syntax
- Specify array start, length, and stride

```
A[:]
A[start : length ]
A[start : length : stride ]
```

- `length` is number of elements in subarray, **not** maximum index in subarray

```
A[:] = B[:] + C[:]
```

Cilk

- Long form

```
A[0:N] = B[0:N] + C[0:N];
D[0:N] = A[0:N] * B[0:N];
```

  - Concise
- Short form

```
for(i=0;i<N;i=i+V){
  A[i:V] = B[i:V] + C[i:V];
  D[i:V] = A[i:V] * B[i:V];
}
```

  - Can be more efficient, loop blocking so should give better cache re-use.
    - Same true of Fortran array syntax

Fortran array syntax and elemental

- Standard Fortran array syntax should vectorise well

```
real, dimension(1024) :: a,b,c
!dir$ attributes align : 64 :: a
!dir$ attributes align : 64 :: b
!dir$ attributes align : 64 :: c
A=B+C
```

- Elemental functions should also allow loops containing them to be vectorised:

```
module test_mod
  implicit none
contains
  elemental real function square(x)
    real, intent(in) :: x
    square = x*x
  end function
end module
```

```
program test_prog
  use test_mod
  implicit none
  integer :: i
  real, dimension(4) :: x = (/ 1.0, 2.0, 3.0, 4.0 /)
  do i=1,4
    square(x(i))
  end do
end program
```

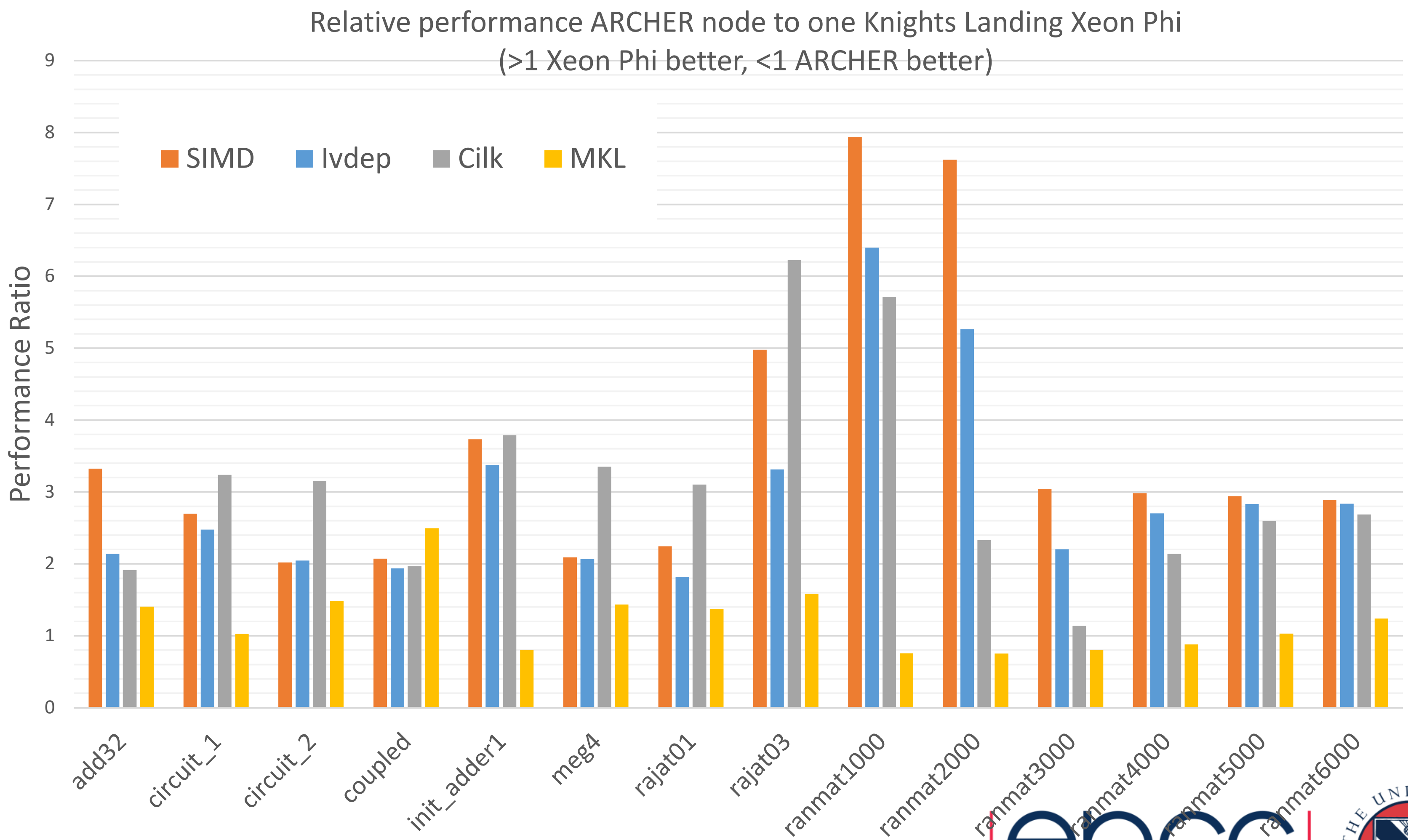
Explicit vector programming

- Can program with explicit vector instructions

```
double A[vec_width], B[vec_width];
__m512d A_vec = _mm512_load_pd(A);
__m512d B_vec = _mm512_load_pd(B);
A_vec = _mm512_add_pd(A_vec,B_vec);
_mm512_store_pd(A,A_vec);
```

- Not recommended as it limits portability
  - i.e. KNC instructions will not perform as efficiently on KNL
  - If want to do from Fortran can cross call between C and Fortran, write kernel in C

Comparing vectorisation performance



Summary

- Vectorisation key to performance on modern processors
  - With it, 32x performance boost for KNL
    - 16 DP vector operations x FMA
- Vector support for real world applications better with KNL
  - Vectorisation of gather/scatter/dependencies better supported
  - Still will have some performance impact
- Test your code with and without vectorisation
  - Manually turn it off at compile time
  - See what the performance difference is
- Look at the compiler vectorisation reports
  - Understand how well it (thinks it) is vectorising your code