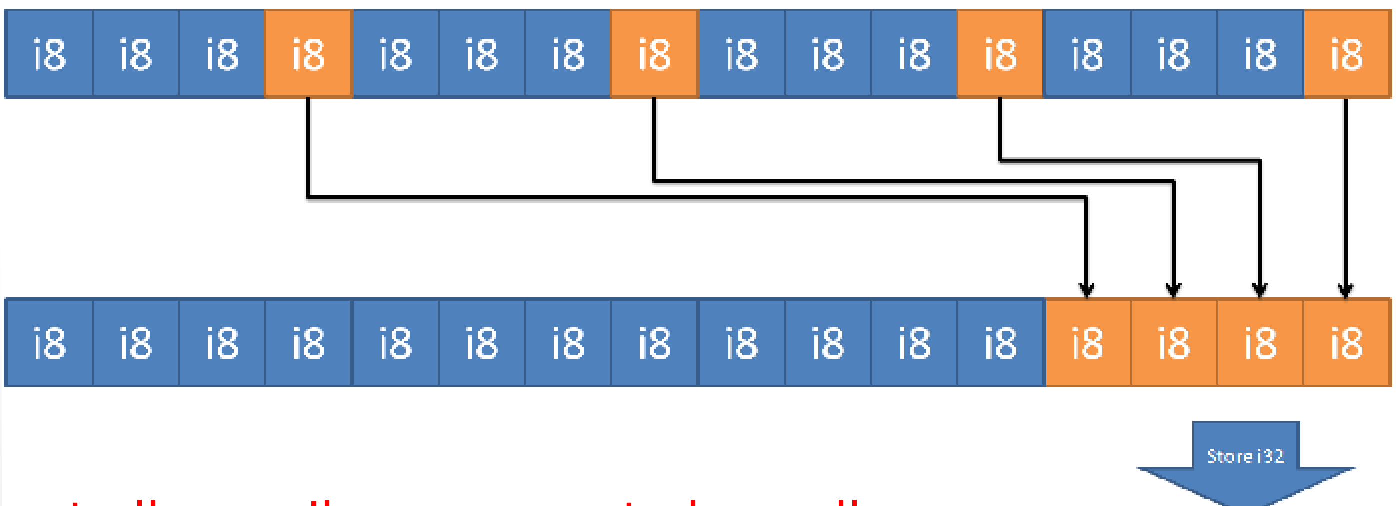
cse5441 - parallel computing

vectorizing compilers



not all compilers are created equally ...

https://i.ytimg.com/vi/LVV5LcZRnCc/maxresdefault.jpg
http://pngimg.com/uploads/darth_vader/darth_vader_PNG3.png
http://2.bp.blogspot.com/-iEM0ks4VajM/Tu8sGVUjBcI/AAAAAAAAEao/Hn4uRBVoK7s/s1600/PackedStore.png
https://upload.wikimedia.org/wikipedia/commons/thumb/5/52/Explosion-153710_icon.svg/833px-Explosion-153710_icon.svg.png

J. S. Jones

array programming

array programming languages (such as Fortran 90) provide operators which generalize scalar functions to higher dimensions

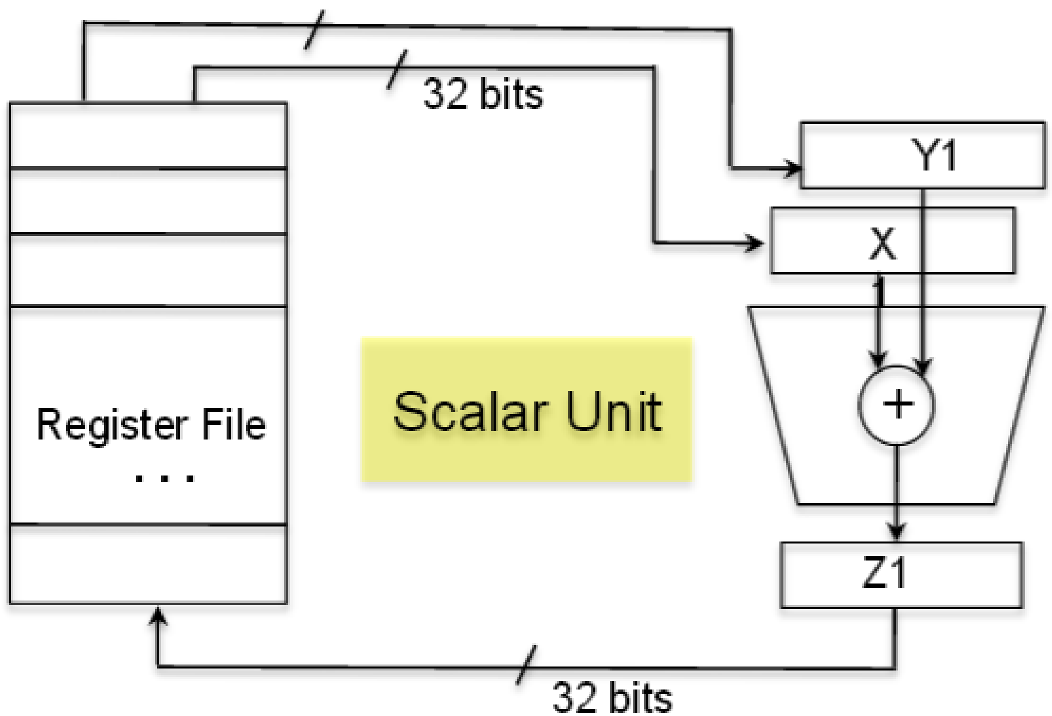
Fortran 77

```
do i=1,n
   do j=1,n
      C(j,i) = A(j,i) + B(j,i)
   enddo
enddo
```

Fortran 90

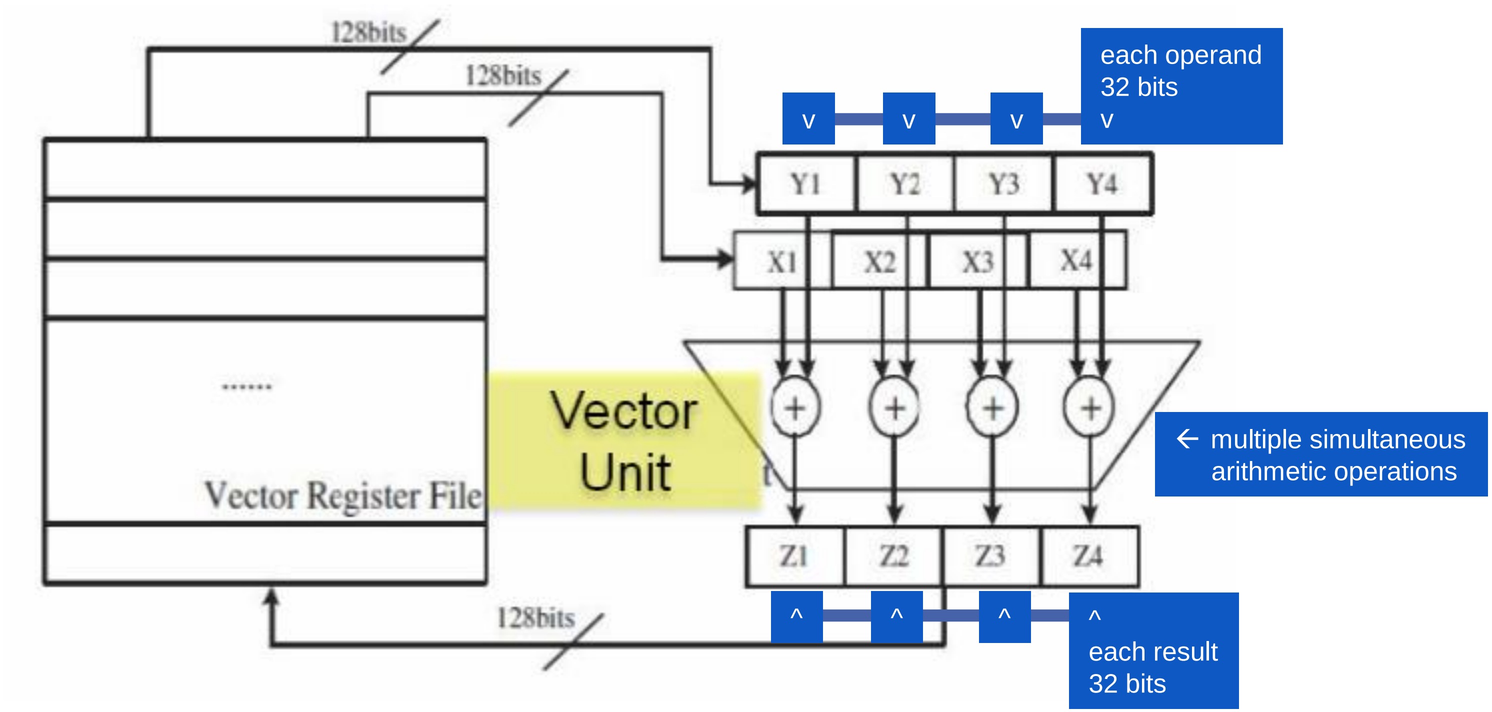
```
C = A + B
```

scalar arithmetic / logic unit (ALU)



David Padua. Department of Computer Science, University of Illinois at Urbana-Champaign

vector arithmetic / logic unit



David Padua. Department of Computer Science, University of Illinois at Urbana-Champaign

unordered data distribution

```
for (iband = 0; iband < nbands; iband++)
{
    for (idir = 0; idir < ndir; idir++)
    {
        for (icell = 0; icell < ncells; icell++)
        {
            for (iface = 0; iface < nfcell[icell]; iface++)
            {
                if ( bface[currf] == 0 )        //interior face
                {
                    do_stuff(interior);
                }
                else                            //boundary face
                {
                    if ( bctype[ibface] == ADIA )
                    {
                        do_stuff(ADIA);
                    }
                    else if ( bctype[ibface] == ISOT )
                    {
                        do_stuff(ISOT);
                    }
                }//end -- if interior or boundary case
                else

                    . . .


            }//end -- loop over cell faces
        }//end – cell loop
    }//end – dir loop
}//end – band loop
```

INPUT DATA (stylized)

INTER
INTER
ADIA
ISOT
PERM
INTER
PERM
ADIZ
ADIZ
INTER
PERM
ISOT
PERM
ADIA
ISOT
XPR
ISOT
PERM
ADIA
INTER
.
.
.

assume all inputs are of same type, and in these application categories

SIMD adaptation

UNORDERED INPUT DATA	PARTITIONED INPUT DATA
INTER	**ADIA**
INTER	**ADIA**
ADIA	**ADIA**
ISOT	**ADIA**
PERM	**ADIA**
INTER	**ADIA**
PERM	**ADIA**
ADIZ	**ADIA**
ADIZ	**ADIA**
INTER	**ADIA**
PERM	**ADIA**
ISOT	**ADIA**
PERM	**.**
ADIA	**.**
ISOT	**.**
XPR	INTER
ISOT	INTER
PERM	INTER
ADIA	INTER
INTER	INTER
.	.
.	.
.	.

SIMD adaptation

```
for (iband = 0; iband < nbands; iband++)
{
    for (idir = 0; idir < ndir; idir++)
    {
        for (icell = 0; icell < ncells; icell++)
        {
            for (iface = 0; iface < nfcell[icell]; iface++)
            {
                if ( bface[currf] == 0 )        //interior face
                {
                    do_stuff(interior);
                }
                else                            //boundary face
                {
                    if ( bctype[ibface] == ADIA )
                    {
                        do_stuff(ADIA);
                    }
                    else if ( bctype[ibface] == ISOT )
                    {
                        do_stuff(ISOT);
                    }
                }//end -- if interior or boundary case
            }//end -- loop over cell faces
        }//end – cell loop
    }//end – dir loop
}//end – band loop
```

```
for (iband = 0; iband < nbands; iband++)
{
    for (idir = 0; idir < ndir; idir++)
    {
        for (iface = 0; iface < nf_max; iface++)
        {
            //process interior cell faces
            for (indx = 0; indx < num_if_cells[iface]; indx++)
            {
                do_stuff(interior);
            }
            //process ISOT cell faces
            for (indx = ISOT_offset;
                 indx < ISOT_offset+num_isot_cells[iface];
                 indx++)
            {
                do_stuff(ISOT);
            }
            //process ADIA cell faces
            for (indx = ADIA_offset;
                 indx < ADIA_offset+num_adia_cells[iface];
                 indx++)
            {
                do_stuff(ADIA);
            }
        }//end – face loop
    }//end – dir loop
}//end – band loop
```

C/C++ vectorizing compilers

```
for (int i = 0;  i < n;  i++)
   c[ i ]  =  a[ i ] + b[ i ];
```

compilers automatically handle the simple cases

```
for (int i = 0;  i < n;  i++)
{
   sum  =  0.0;
   for (int j = 0;  j < n;  j++)
   {
      sum +=  A[ j ][ i ];
   }
   B[ i ]  =  sum;
}
```

what makes this loop more challenging?

- no stride-1 access
- sum creates a loop-carried dependency for i

C/C++ vectorizing compilers

example

```
for (int i = 0;  i < n;  i++)
{
   sum  =  0.0;
   for (int j = 0;  j < n;  j++)
   {
      sum +=  A[ j ][ i ];
   }
   B[ i ]  =  sum;
}
```

A

```
for (int i = 0;  i < n;  i++)
{
   sum[ i ]  =  0.0;
   for (int j = 0;  j < n;  j++)
   {
      sum[ i ] +=  A[ j ][ i ];
   }
   B[ i ]  =  sum[ i ];
}
```

B

scalar expansion:

eliminates loop dependency

```
for (int i = 0;  i < n;  i++)
{
   sum[ i ]  =  0.0;
}

for (int j = 0;  j < n;  j++)
{
   for (int i = 0;  i < n;  i++)
   {
      sum[ i ] +=  A[ j ][ i ];
   }
}

for (int i = 0;  i < n;  i++)
{
   B[ i ]  =  sum[ i ];
}
```

C

plus

loop reordering and distribution:

provides stride-1 access

C/C++ vectorizing compilers

example

```
for (int i = 0;  i < n;  i++)
{
   sum  =  0.0;
   for (int j = 0;  j < n;  j++)
   {
      sum +=  A[ j ][ i ];
   }
   B[ i ]  =  sum;
}
```

A

```
for (int i = 0;  i < n;  i++)
{
   sum[ i ]  =  0.0;
   for (int j = 0;  j < n;  j++)
   {
      sum[ i ] +=  A[ j ][ i ];
   }
   B[ i ]  =  sum[ i ];
}
```

B

Intel Nehalem

loop not vectorized

IBM Power 7

loop not vectorized

Intel Nehalem

loop vectorized

speedup: 2.6   (62% faster)

relative run-time 0.6

IBM Power 7

loop interchanged and vectorized

speedup: 2.0   (50% faster)

relative run-time 0.2

test results courtesy of UIUC

stripmine

step 1 of 2

ORIGINAL

```
for (int i = 0;  i < n;  i++)
{
S1 A[ i ]  =  B[ i ] + 1.0;
S2 C[ i ]  =  A[ i ] + 2.0;
}
```

STRIPMINE

```
for (int i = 0;  i < n;  i+= stripsize)
{
   for (int j = i;  j < i+stripsize;  j++)
   {
      A[ j ]  =  B[ j ] + 1.0;
      C[ j ]  =  A[ j ] + 2.0;
   }
}
```

stripmine - distribute

step 2 of 2

ORIGINAL

```
for (int i = 0;  i < n;  i++)
{
S1 A[ i ]  =  B[ i ] + 1.0;
S2 C[ i ]  =  A[ i ] + 2.0;
}
```

STRIPMINE

```
for (int i = 0;  i < n;  i+= stripsize)
{
   for (int j = i;  j < i+stripsize;  j++)
   {
      A[ j ]  =  B[ j ] + 1.0;
      C[ j ]  =  A[ j ] + 2.0;
   }
}
```

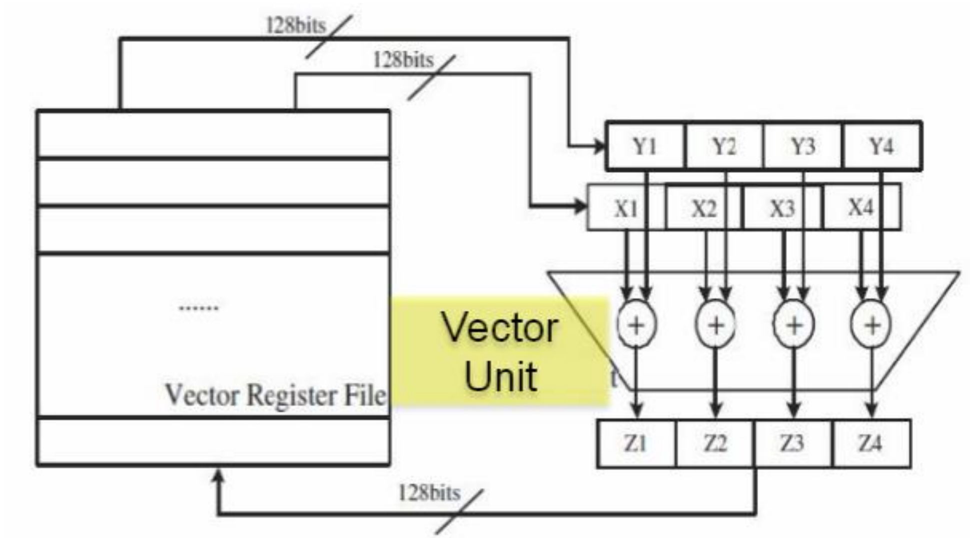
DISTRIBUTE

```
for (int i = 0;  i < n;  i+= stripsize)
{
   for (int j = i;  j < i+stripsize;  j++)
   {
      A[ j ]  =  B[ j ] + 1.0;
   }
   for (int j = i;  j < i+stripsize;  j++)
   {
      C[ j ]  =  A[ j ] + 2.0;
   }
}
```

VECTORIZED

```
   for (int  i = k;  i < n;  i+=q)
   {
      A[ i  :  i+q-1 ]  =  B[ i  :  i+q-1 ] + 1.0;
   }
 for (int  i = k;  i < n;  i+=q)
   {
      C[ i  :  i+q-1 ]  =  B[ i  :  i+q-1 ] + 2.0;
    }
}
```

vectorization result



scalar:

load	r1	&operand1
load	r2	&operand2
add	r3	r1, r2
store	r3	&result
load	r1	&operand1a
load	r2	&operand2a
add	r3	r1, r2
store	r3	&result

scalar:

load	r1	&operand1b
load	r2	&operand2b
add	r3	r1, r2
store	r3	&result
load	r1	&operand1c
load	r2	&operand2c
add	r3	r1, r2
store	r3	&result

vector:

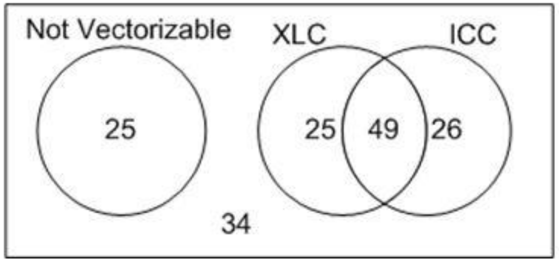
loadv	vr1	&operand1
loadv	vr2	&operand2
addv	vr3	vr1, vr2
storev	vr3	&result

David Padua. Department of Computer Science, University of Illinois at Urbana-Champaign

how well do compilers vectorize?

Loops \ Compiler	XLC	ICC	GCC
Total	159		
Vectorized	74	75	32
Not vectorized	85	84	127
Average Speed Up	1.73	1.85	1.30

Loops \ Compiler	XLC but not ICC	ICC but not XLC
Vectorized	25	26



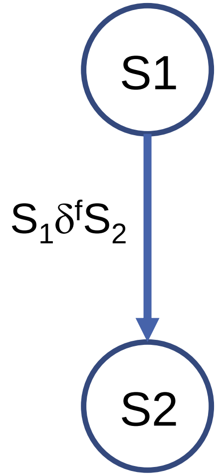
adding manual vectorization hints increased the average speedup (IBM) from 1.73 to 3.78

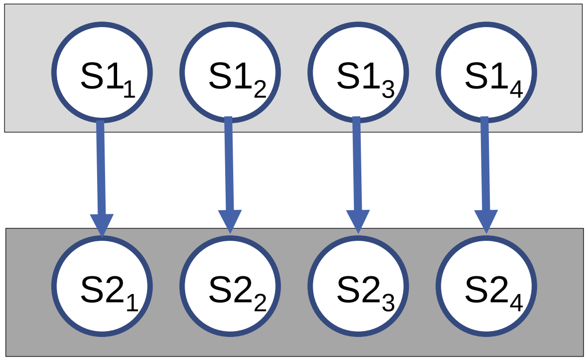
David Padua. Department of Computer Science, University of Illinois at Urbana-Champaign

acyclic dependence graphs

forward dependencies

```
for (int i = 0; i < max; i++)
{
S1  a[ i ] = b[ i ] + c[ i ];
S2  d[ i ] = a[ i ] + 1;
}
```





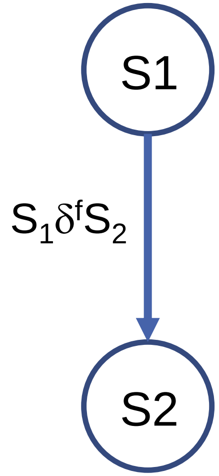
can we group all the $S1_n$ and follow with $S2_n$?

J. S. Jones

forward dependencies

example

```
for (int i = 0; i < max; i++)
{
S1  a[ i ] = b[ i ] + c[ i ];
S2  d[ i ] = a[ i ] + 1;
}
```

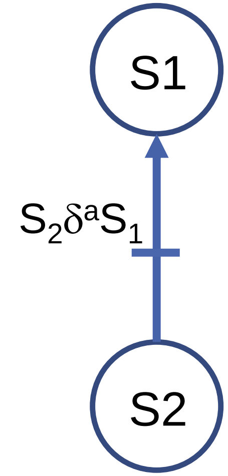


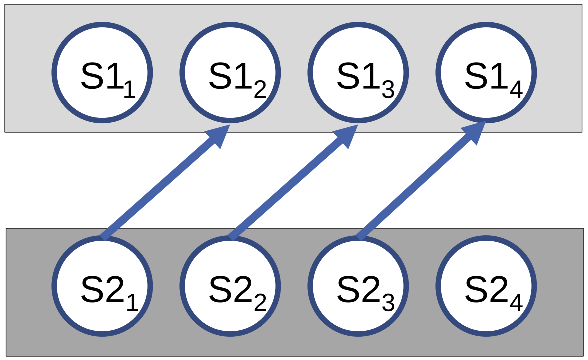
XLC: vectorized, speedup 2.0
ICC: vectorized, speedup 1.6

acyclic dependence graphs

backward dependencies

```
for (int i = 0; i < max; i++)
{
S1  a[ i ] = b[ i ] + c[ i ];
S2  d[ i ] = a[ i+1 ] + 1;
}
```





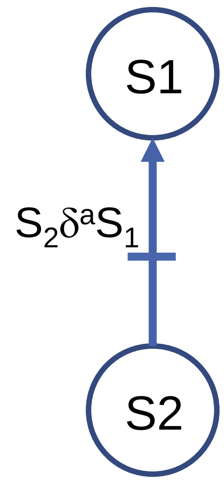
can we group all the $S2_n$ and follow with $S1_n$?

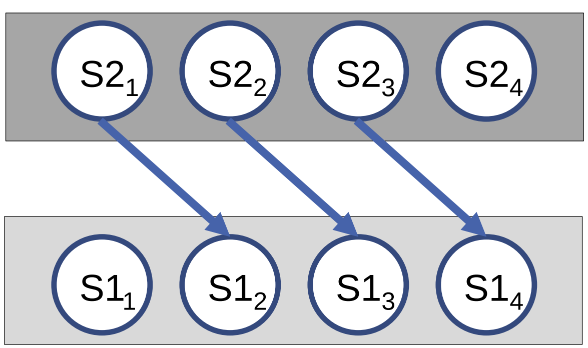
J. S. Jones

acyclic dependence graphs

backward dependencies

```
for (int i = 0; i < max; i++)
{
S2  d[ i ] = a[ i+1 ] + 1;
S1  a[ i ] = b[ i ] + c[ i ];
}
```





can we group all the $S2_n$ and follow with $S1_n$?

backward dependencies

example

original

```
for (int i = 0; i < max; i++)
{
    a[ i ] = b[ i ] + c[ i ];
    d[ i ] = a[ i+1 ] + 1;
}
```

re-ordered

```
for (int i = 0; i < max; i++)
{
    d[ i ] = a[ i+1 ] + 1;
    a[ i ] = b[ i ] + c[ i ];
}
```

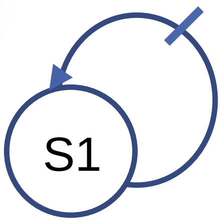
ICC time 12.6
non-vectorized

XLC time 0.6
vectorized

ICC time 9.4
vectorized

XLC time 0.6
vectorized

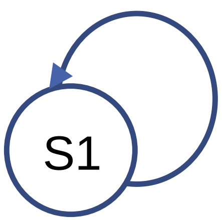
cyclic dependence graphs



$S_1 \delta^a S_1$

```
for (int i = 0; i < max; i++)
{
   a[ i ] = a[ i+1] + b[ i ];
}
```

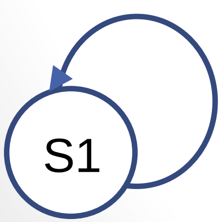
ICC
vectorized
XLC
vectorized



$S_1 \delta^f S_1$

```
for (int i = 0; i < max; i++)
{
   a[ i ] = a[ i-1] + b[ i ];
}
```

ICC
non-vectorized
XLC
non-vectorized



$S_1 \delta^f S_1$

```
for (int i = 0; i < max; i++)
{
   a[ i ] = a[ i-4] + b[ i ];
}
```

ICC
vectorized
XLC
vectorized

more on vectorizing compilers ...

references:

"Intel® Cilk™ Plus Support," [online] software.intel.com/en-us/articles/intel-cilk-plus-support

"A Guide to Auto-vectorization with Intel® C++ Compilers," [online]
software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers

"Quick-Reference Guide to Optimization with Intel® Compilers version 13," [online]
software.intel.com/sites/default/files/Compiler_QRG_2013.pdf

"Intel® C++ Intrinsics Reference," [online]
software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/intref_cls/common/intref_bk_intro.htm

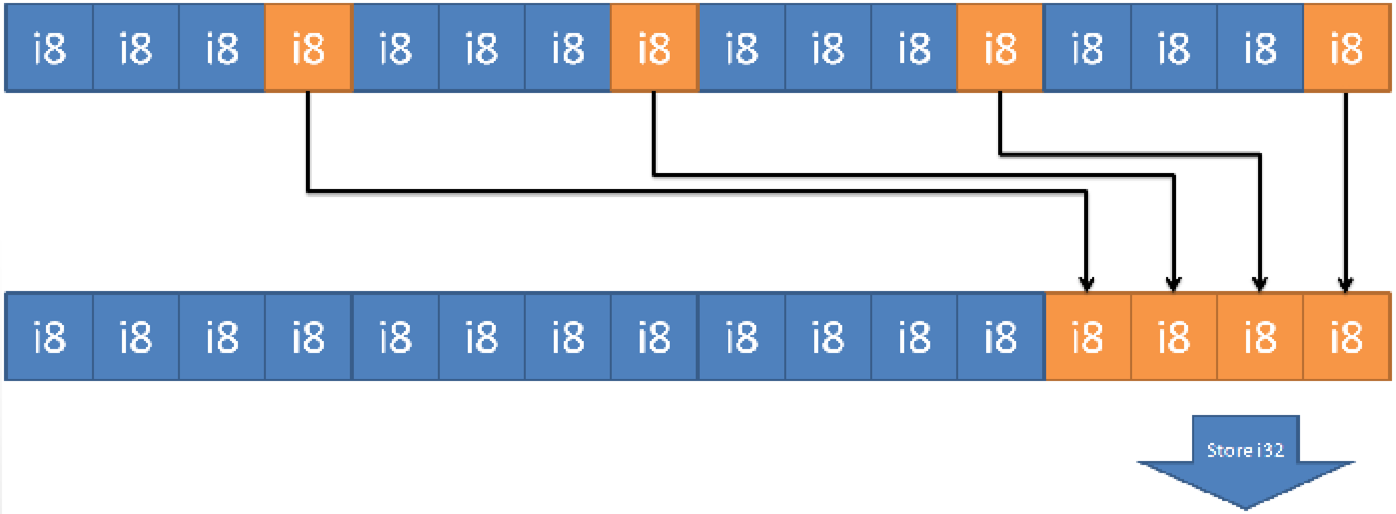
"Intel Advanced Vector Extensions (AVX)," [online] software.intel.com/en-us/avx

"Compiler Prefetching for the Intel® Xeon Phi™ coprocessor," [online]
software.intel.com/sites/default/files/managed/5d/f3/5.3-prefetching-on-mic-4.pdf

trade-offs

vectorization is not always profitable ...

vectorizing compilers



not all compilers are created equally …

http://2.bp.blogspot.com/-iEM0ks4VajM/Tu8sGVUjBcI/AAAAAAAAEao/Hn4uRBVoK7s/s1600/PackedStore.png