Program Optimization Through Loop Vectorization

María Garzarán, Saeed Maleki
William Gropp and David Padua

Department of Computer Science
University of Illinois at Urbana-Champaign

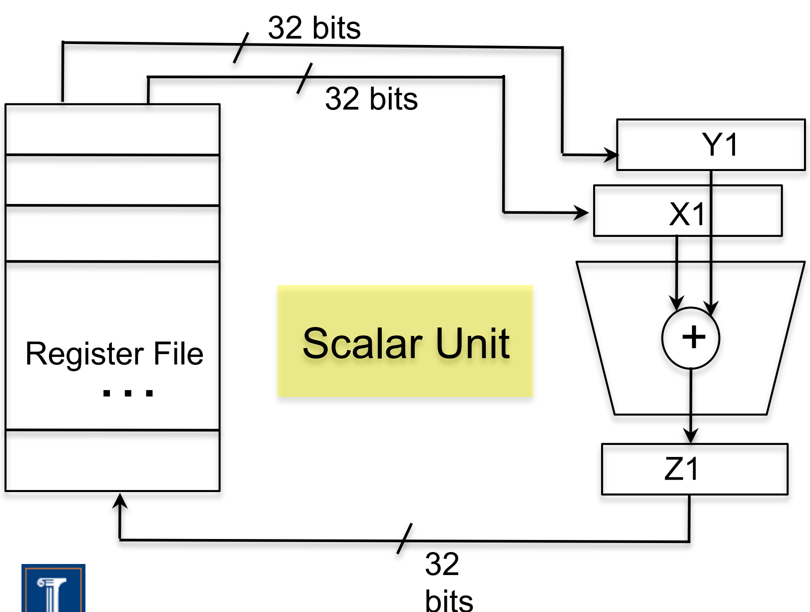
Simple Example

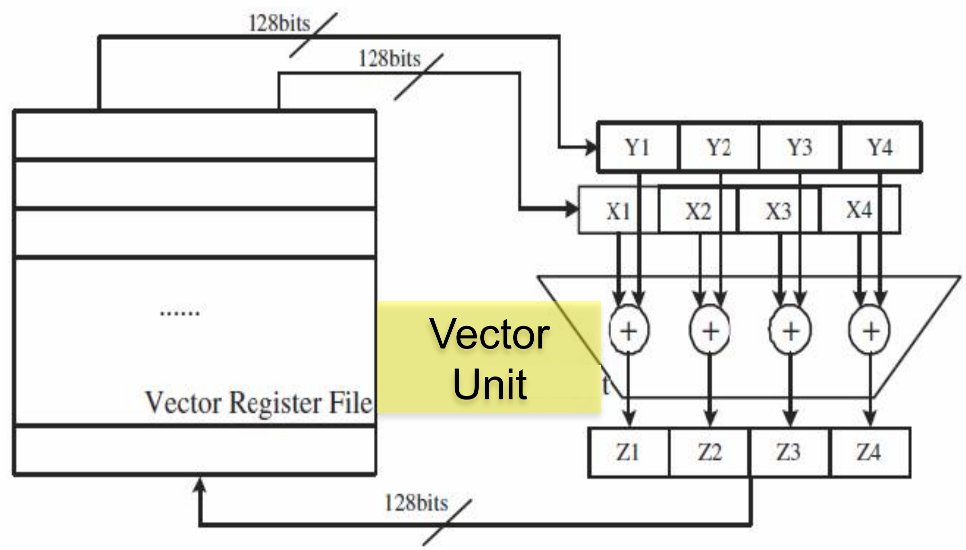
- Loop vectorization transforms a program so that the same operation is performed at the same time on several vector elements

n times
```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3
```

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

n/4 times
```
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3, addr3
```





SIMD Vectorization

- The use of SIMD units can speed up the program.
- Intel SSE and IBM Altivec have 128-bit vector registers and functional units
  - 4 32-bit single precision floating point numbers
  - 2 64-bit double precision floating point numbers
  - 4 32-bit integer numbers
  - 2 64 bit integer
  - 8 16-bit integer or shorts
  - 16 8-bit bytes or chars

- Assuming a single ALU, these SIMD units can execute 4 single precision floating point number or 2 double precision operations in the time it takes to do only one of these operations by a scalar unit.

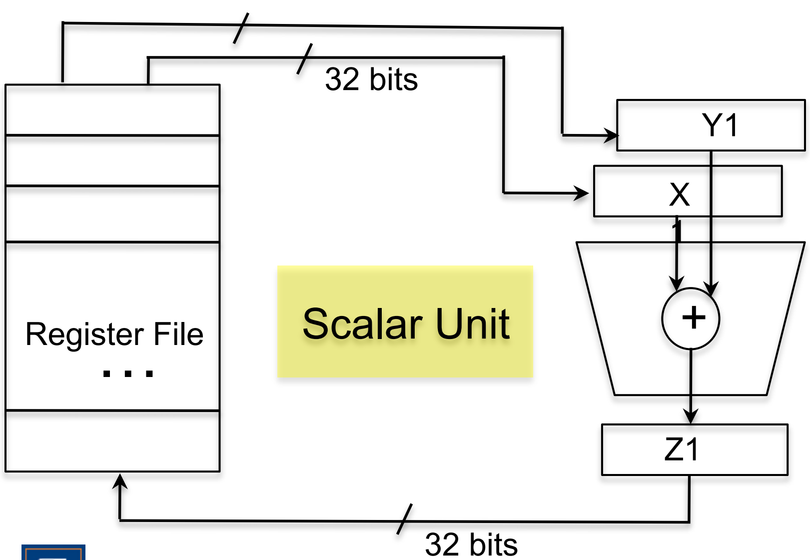
Executing Our Simple Example

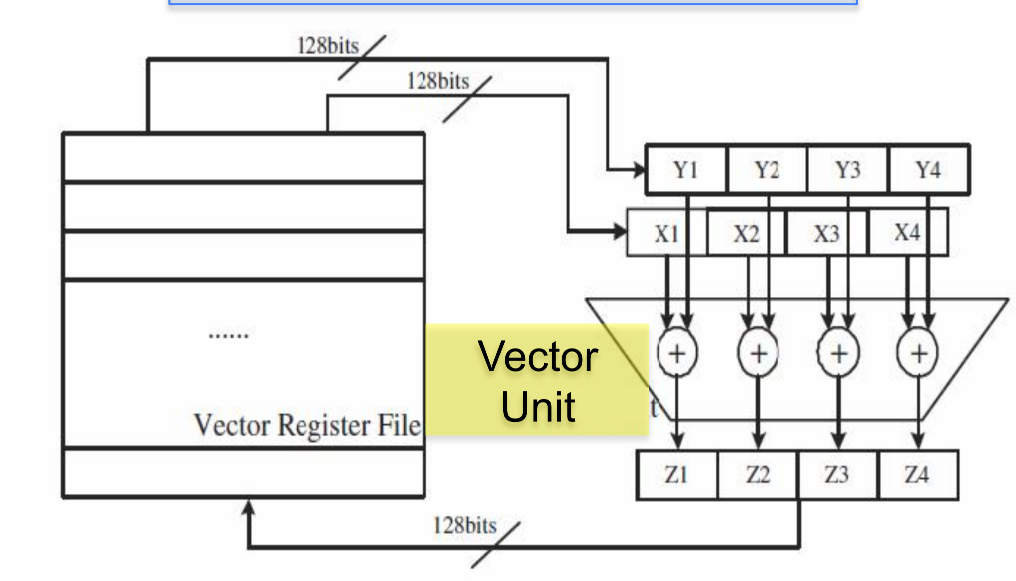
S000

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

Intel Nehalem
Exec. Time scalar code: 6.1
Exec. Time vector code: 3.2
Speedup: 1.8

IBM Power 7
Exec. Time scalar code: 2.1
Exec. Time vector code: 1.0
Speedup: 2.1





How do we access the SIMD units?

- Three choices
  1. C code and a vectorizing compiler

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

  2. Macros or Vector Intrinsics

```
void example(){
__m128 rA, rB, rC;
 for (int i = 0; i <LEN; i+=4){
     rA = _mm_load_ps(&a[i]);
     rB = _mm_load_ps(&b[i]);
     rC = _mm_add_ps(rA,rB);
    _mm_store_ps(&C[i], rC);
}}
```

  3. Assembly Language

```
..B8.5
   movaps     a(,%rdx,4), %xmm0
   addps      b(,%rdx,4), %xmm0
   movaps     %xmm0, c(,%rdx,4)
   addq       $4, %rdx
   cmpq       $rdi, %rdx
   jl         ..B8.5
```

Compiler directives

S1111

```
void test(float* A, float* B, float*
C, float* D, float* E)
{
  for (int i = 0; i <LEN; i++){
   A[i]=B[i]+C[i]+D[i]+E[i];
  }
}
```

S1111

Intel Nehalem
Compiler report: Loop was not vectorized.
Exec. Time scalar code: 5.6
Exec. Time vector code: --
Speedup: --

S1111

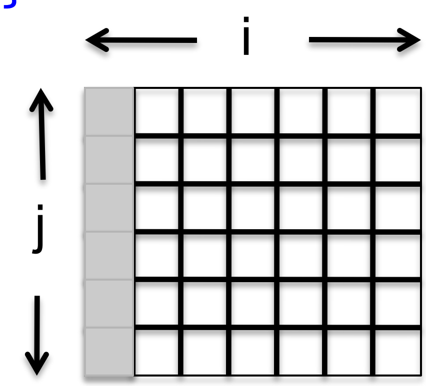
```
void test(float* __restrict__ A,
float* __restrict__ B,
float* __restrict__ C,
float* __restrict__ D,
float* __restrict__ E)
{
  for (int i = 0; i <LEN; i++){
   A[i]=B[i]+C[i]+D[i]+E[i];
  }
}
```

S1111

Intel Nehalem
Compiler report: Loop was vectorized.
Exec. Time scalar code: 5.6
Exec. Time vector code: 2.2
Speedup: 2.5

Loop Transformations

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




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

Loop Transformations

S136

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

S136_1

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

S136_2

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

S136

Intel Nehalem
Compiler report: Loop was not vectorized. Vectorization possible but seems inefficient
Exec. Time scalar code: 3.7
Exec. Time vector code: --
Speedup: --

S136_1

Intel Nehalem
report: Permuted loop was vectorized.
scalar code: 1.6
vector code: 0.6
Speedup: 2.6

S136_2

Intel Nehalem
report: Permuted loop was vectorized.
scalar code: 1.6
vector code: 0.6
Speedup: 2.6

Stripmining

- Stripmining is a simple transformation.

```
for (i=1; i<n; i++){
    …
}
```

```
/* n is a multiple of q */
for (k=1; k<n; k+=q){
   for (i=k; i<k+q-1; i++){
   …
   }
}
```

- It is typically used to improve locality.

Stripmining (cont.)

- Stripmining is often used when vectorizing

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

stripmine

```
for (k=1; k<n; k+=q){
/* q could be size of vector register */
  for (i=k; i < k+q; i++){
     a[i] = b[i] + 1;
     c[i] = a[i-1] + 2;
  }
}
```

vectorize

```
for (i=1; i<n; i+=q){
  a[i:i+q-1] = b[i:i+q-1] + 1;
  c[i:i+q-1] = a[i:i+q] + 2;
}
```

Loop Vectorization

- Loop Vectorization is not always a legal and profitable transformation.
- Compiler needs:
  - Compute the dependences
    - The compiler figures out dependences by
      - Solving a system of (integer) equations (with constraints)
      - Demonstrating that there is no solution to the system of equations
  - Remove cycles in the dependence graph
  - Determine data alignment
  - Vectorization is profitable

Simple Example

- Loop vectorization transforms a program so that the same operation is performed at the same time on several of the elements of the vectors

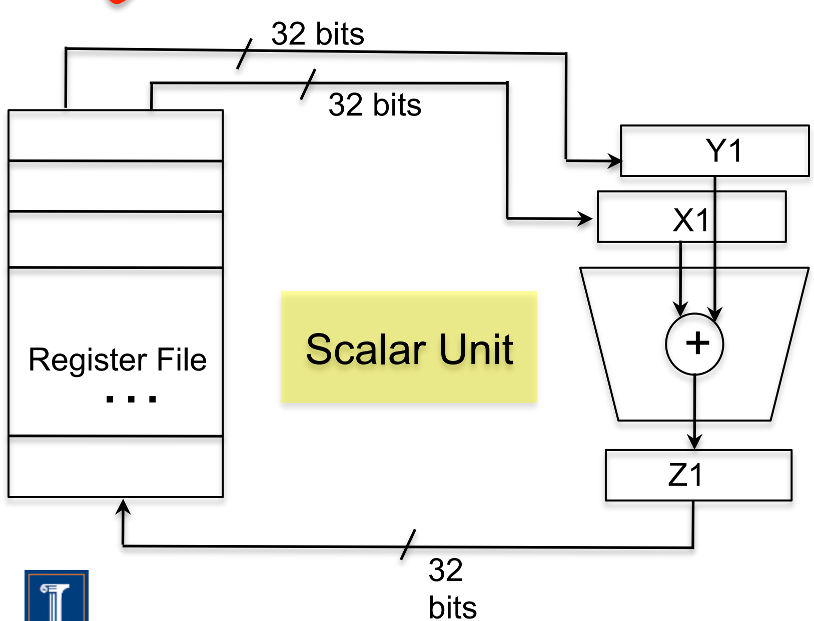
n times

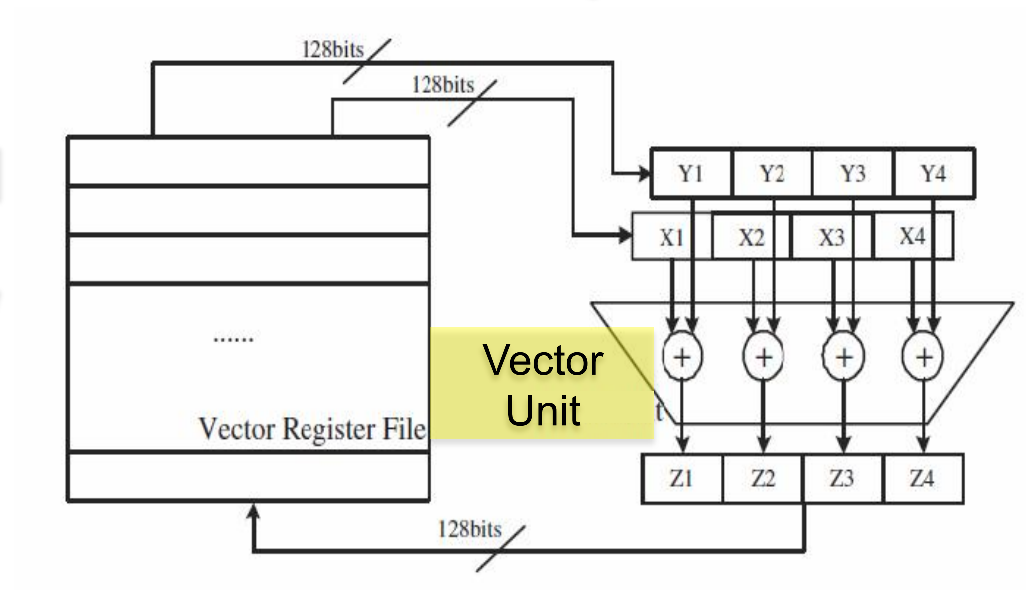
```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3
```

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

n/4 times

```
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3m addr3
```





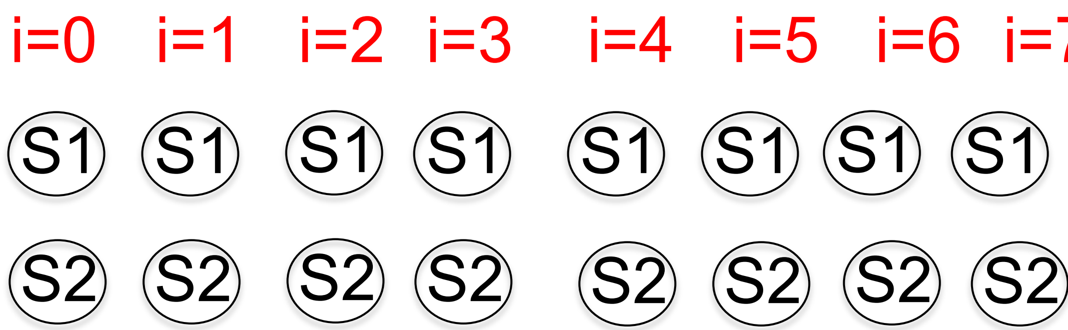
Loop Vectorization

- When vectorizing a loop with several statements the compiler need to strip-mine the loop and then apply loop distribution

```
 for (i=0; i<LEN; i++){
S1 a[i]=b[i]+(float)1.0;
S2 c[i]=b[i]+(float)2.0;
 }
```

→

```
for (i=0; i<LEN; i+=strip_size){
 for (j=i; j<i+strip_size; j++)
   a[j]=b[j]+(float)1.0;
 for (j=i; j<i+strip_size; j++)
   c[j]=b[j]+(float)2.0;
}
```



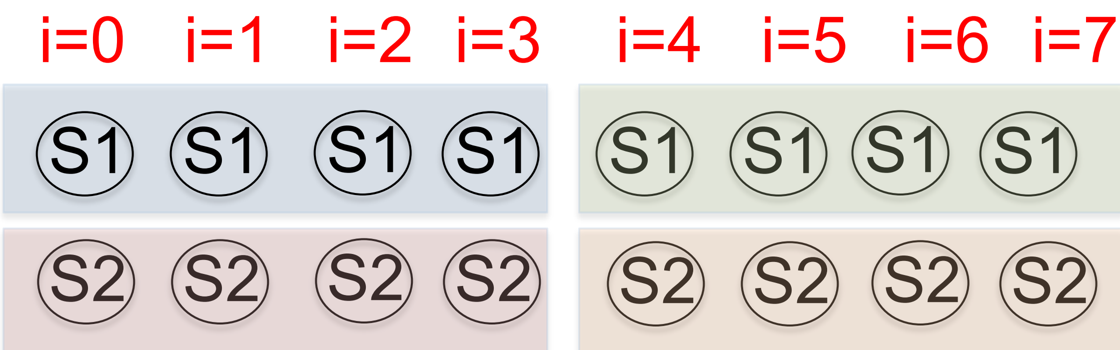
Loop Vectorization

- When vectorizing a loop with several statements the compiler need to strip-mine the loop and then apply loop distribution

```
 for (i=0; i<LEN; i++){
S1 a[i]=b[i]+(float)1.0;
S2 c[i]=b[i]+(float)2.0;
 }
```

→

```
for (i=0; i<LEN; i+=strip_size){
 for (j=i; j<i+strip_size; j++)
   a[j]=b[j]+(float)1.0;
 for (j=i; j<i+strip_size; j++)
   c[j]=b[j]+(float)2.0;
}
```



Loop Transformations

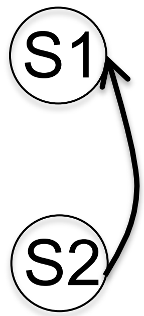
- Compiler Directives
- Loop Distribution or loop fission
- Reordering Statements
- Node Splitting
- Scalar expansion
- Loop Peeling
- Loop Fusion
- Loop Unrolling
- Loop Interchanging

Acyclic Dependenden Graphs Backward Dependences (II)

S214

```
for (int i=1;i<LEN;i++) {
  a[i]=d[i-1]+(float)sqrt(c[i]);
  d[i]=b[i]+(float)sqrt(e[i]);
}
```

S114

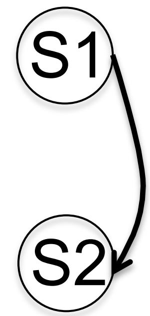


Intel Nehalem
Compiler report: Loop was not vectorized. Existence of vector dependence
Exec. Time scalar code: 7.6
Exec. Time vector code: --
Speedup: --

S214_1

```
for (int i=1;i<LEN;i++) {
  d[i]=b[i]+(float)sqrt(e[i]);
  a[i]=d[i-1]+(float)sqrt(c[i]);
}
```

S114_1



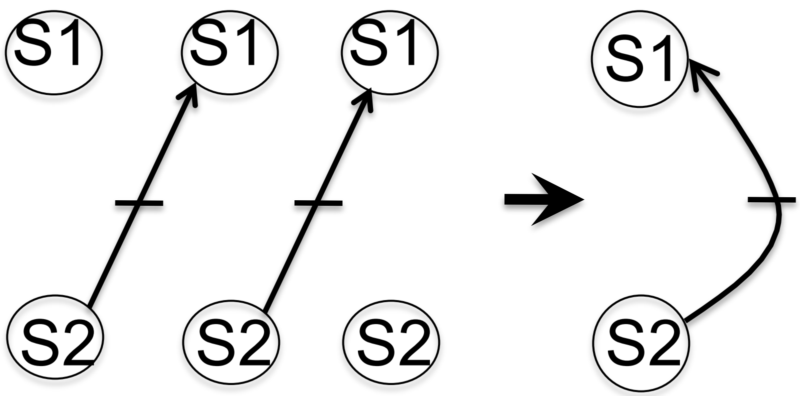
Intel Nehalem
Compiler report: Loop was vectorized
Exec. Time scalar code: 7.6
Exec. Time vector code: 3.8
Speedup: 2.0

Acyclic Dependenden Graphs Backward Dependences (I)

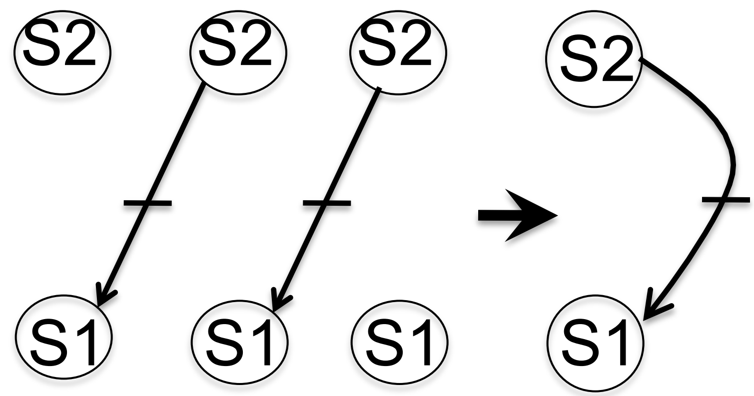
Reorder of statements

```
  for (i=0; i<LEN; i++) {
S1  a[i]= b[i] + c[i]
S2  d[i] = a[i+1] + (float) 1.0;
  }
```

```
  for (i=0; i<LEN; i++) {
S2  d[i] = a[i+1]+(float)1.0;
S1  a[i]= b[i] + c[i];
  }
```



backward
depedence



forward
depedence

Conditional Statements – I

- Loops with conditions need `#pragma vector always`
  - Since the compiler does not know if vectorization will be profitable
  - The condition may prevent from an exception

```
#pragma vector always
for (int i = 0; i < LEN; i++){
  if (c[i] < (float) 0.0)
    a[i] = a[i] * b[i] + d[i];
}
```

Conditional Statements – I

S137

```
for (int i = 0; i < LEN; i++){
  if (c[i] < (float) 0.0)
    a[i] = a[i] * b[i] + d[i];
}
```

S137_1

```
#pragma vector always
for (int i = 0; i < LEN; i++){
  if (c[i] < (float) 0.0)
    a[i] = a[i] * b[i] + d[i];
}
```

S137

Intel Nehalem
Compiler report: Loop was not vectorized. Condition may protect exception
Exec. Time scalar code: 10.4
Exec. Time vector code: --
Speedup: --

S137_1

Intel Nehalem
Compiler report: Loop was vectorized.
Exec. Time scalar code: 10.4
Exec. Time vector code: 5.0
Speedup: 2.0

Conditional Statements

- Compiler removes *if conditions* when generating vector code

```
for (int i = 0; i < LEN; i++){
  if (c[i] < (float) 0.0)
    a[i] = a[i] * b[i] + d[i];
}
```

Compiler Directives

- Compiler vectorizes many loops, but many more can be vectorized if the appropriate directives are used

Compiler Hints for Intel ICC	Semantics
#pragma ivdep	Ignore assume data dependences
#pragma vector always	override efficiency heuristics
#pragma novector	disable vectorization
__restrict__	assert exclusive access through pointer
__attribute__ ((aligned(int-val)))	request memory alignment
memalign(int-val,size);	malloc aligned memory
__assume_aligned(exp, int-val)	assert alignment property