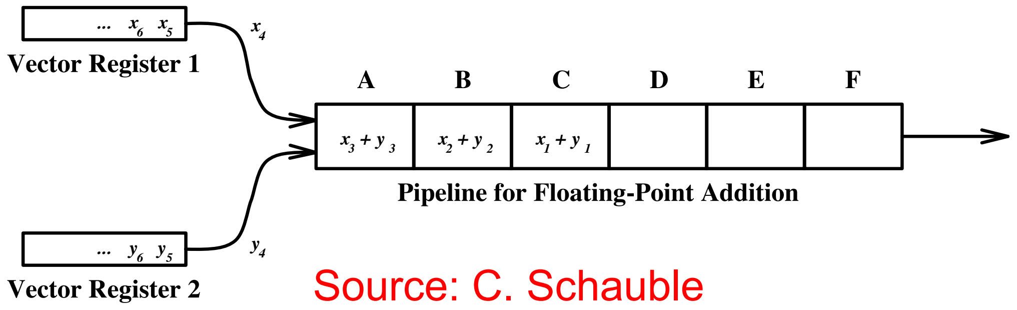
Why Vectorization?

- What is vectorization?
  - It is the expression of a computation in terms of identical operations on vectors of data
- Initial motivation:
  - Enhanced performance with pipelined functional units in early supercomputers: independent operations on components of vector to feed the units
- Today: Vector-SIMD ISAs (SSE, AVX, AVX-512,..) and SIMD functional units in GPUs enable both energy efficiency and high performance with vectorizable computations

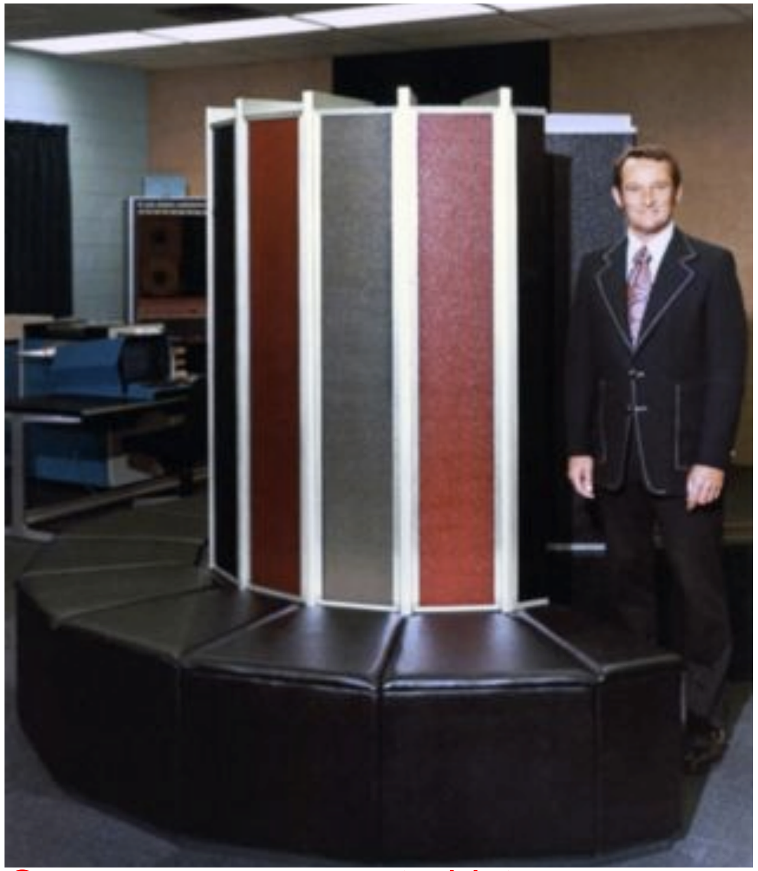
Vector Supercomputers



Source: C. Schauble

Step	Time ⟶ τ	2τ	3τ	4τ	5τ	6τ	7τ	8τ
A	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$	$x_7 + y_7$	$x_8 + y_8$
B		$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$	$x_7 + y_7$
C			$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$
D				$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$
E					$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$
F						$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$

- First effective vector supercomputer: Cray-1
  - 64-element vector registers
  - Pipelined execution of operations
  - Vectorizing FORTRAN compiler



Source: www.computerhistory.org

Vector Machines

Machine	Year	Clock	Regs	Elements	FUs	LSUs
Cray 1	1976	80 MHz	8	64	6	1
Cray XMP	1983	120 MHz	8	64	8	2L, 1S
Cray YMP	1988	166 MHz	8	64	8	2L, 1S
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1996	455 MHz	8	128	8	4
Conv. C-1	1984	10 MHz	8	128	4	1
Conv. C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP300	1996	100 MHz	8-256	32-1024	3	2
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/3	1995	400 MHz	8+8K	256+var	16	8

Source: F. Chong

Vectors in Commodity Systems

- Initial use of vectors was only in high-end supercomputers
- By mid 90’s, desktops became powerful enough for real-time gaming and video
- Performance demands led to vector-SIMD instruction set architectures (ISAs) on commodity processors

Year	Vendor	ISA
1995	Sun	VIS
1996	Intel	MMX
1997	IBM/Motorola	Altivec
1998	AMD	3DNow!
1999	Intel	SSE
2001	Intel	SSE2
2004	Intel	SSE3
2006	Intel	SSSE3
2006	Intel	SSE4
2008	Intel	AVX
2011	ARM	NEON
2015	Intel	AVX-512

Vector-SIMD Model

n times

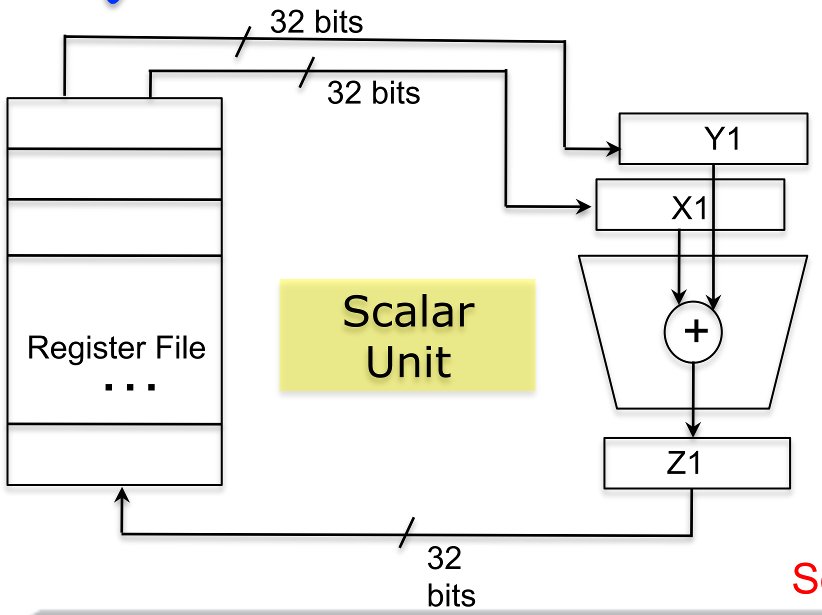
```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3
...
// Add 4 to addr1
// addr2, addr3
...
```

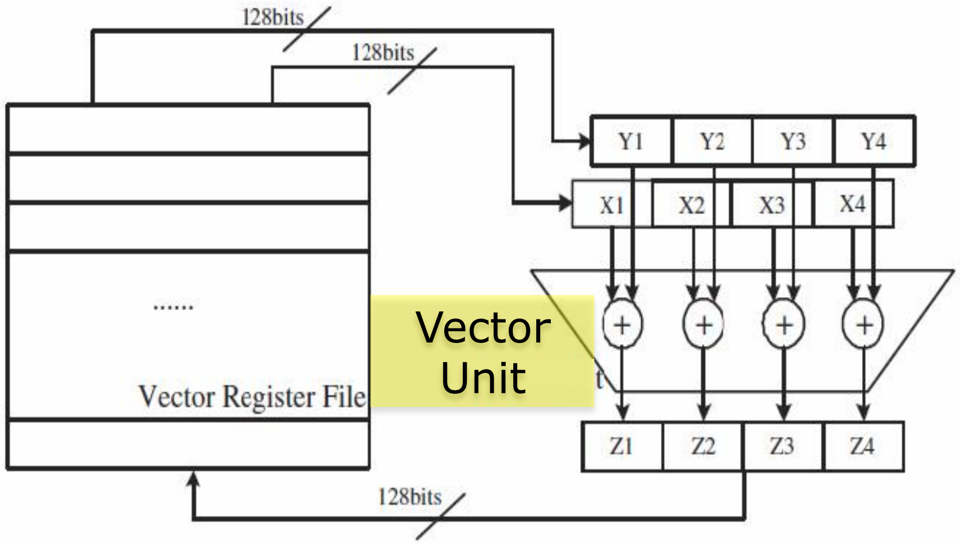
```
for (i=0; i<n; i++)
  c[i]=a[i]+b[i];
```

```
for (i=0; i<n; i+=4)
  c[i:i+3]=a[i:i+3]+b[i:i+3];
```

n/4 times

```
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3, addr3
...
// Add 16 to addr1
// addr2, addr3
...
```





Source: Maria Garzaran and David Padua

Vector-SIMD Characteristics

- SIMD: Single Instruction Multiple Data
  - Synchronized lock-step execution by multiple functional units
  - Data generally contiguous in memory
- SIMD functional units in most CPUs and GPUs
- High performance and energy efficiency
- But, greater burden on programmer, unless automatically vectorized by compiler

Utilizing Vector-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);
}}
```

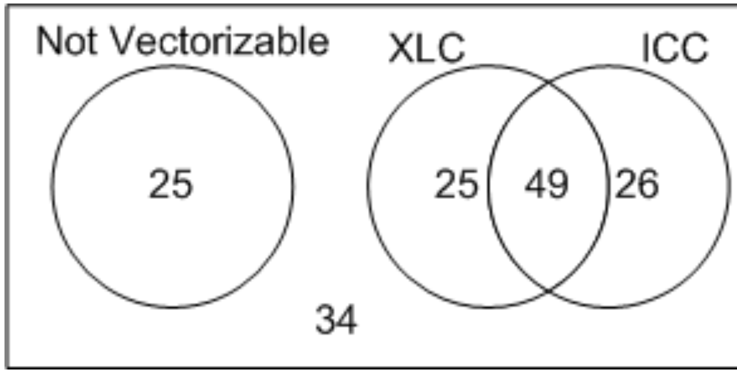
3. Assembly Language

```
..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
```

Source: Maria Garzaran and David Padua

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

By adding manual vectorization the average speedup was 3.78 (versus 1.73 obtained by the XLC compiler)

Source: Maria Garzaran and David Padua

Example: Stride-1, Independent Ops

```
for (i=0; i<N; i++) A[i] = A[i]+1;
```

- Stride-1 accesses to Array A
- Loop has independent operations (no loop carried dependences)
- Data is resident in L1 cache
- SSE: 128-bit vector = 2 doubles = 4 floats/ints = 16 chars
- AVX: 256-bit vector = 4 doubles = 8 floats/ints = 32 chars
- Intel icc v.13.1.3; compiled –fast; enable/disable vectorization

	char	int	float	dble
No-Vector	0.5	0.5	0.5	0.67
Vector	9.7	2.5	2.5	1.36
Speedup	19.4	5.0	5.0	2.0

Ops/cycle
Nehalem (Core i7-920 @2.67 Ghz)

	char	int	float	dble
No-Vector	0.88	0.9	0.95	0.87
Vector	9.5	3.8	7.2	3.8
Speedup	10.8	4.2	7.6	4.4

Ops/cycle
Haswell (Core i7-4770K @3.49 Ghz)

Example: Stride-16, Independent Ops

```
for (i=0; i<N; i+=16) A[i] = A[i]+1;
```

- Stride-16 accesses to Array A
- No performance gain; often loss of performance from vectorization
- Unlike past vector machines like the Cray-1, current vector-SIMD ISAs do not support efficient strided data access from memory

	char	int	float	dble
No-Vec	0.65	0.66	0.5	0.5
Vector	0.52	0.56	0.4	0.65
Speedup	0.80	0.85	0.80	1.30

Ops/cycle
Nehalem (Core i7-920 @2.67 Ghz)

	char	int	float	dble
No-Vec	0.76	0.81	0.69	0.69
Vector	0.54	0.66	0.51	1.0
Speedup	0.71	0.81	0.74	1.45

Ops/cycle
Haswell (Core i7-4770K @3.49 Ghz)

Example: Stride-1, Dependent Ops

```
for (i=1; i<N; i++) A[i] = A[i-1]+1;
```

- Stride-1 accesses to Array A
- Loop has loop-carried dependence
- Cannot use vector instructions since independent operations are not available

	char	int	float	dble
No-Vec	0.66	0.66	0.33	0.33
Vector	0.66	0.66	0.33	0.33
Speedup	1.0	1.0	1.0	1.0

Ops/cycle
Nehalem (Core i7-920 @2.67 Ghz)

	char	int	float	dble
No-Vec	1.0	1.0	0.33	0.33
Vector	1.0	1.0	0.33	0.33
Speedup	1.0	1.0	1.0	1.0

Ops/cycle
Haswell (Core i7-4770K @3.49 Ghz)

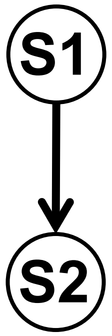
Vectorization of Loop Programs

1. Independent operations (usually in innermost loops) can be vectorized
2. For short-vector SIMD ISAs, the operands must also have unit stride w.r.t. inner loop (or stride 0, i.e., be loop invariant)

- For simple single-statement, single-loop programs, loop carried dependences can inhibit vectorization
- For multi-statement and/or multi-loop code:
  - Data dependence graph analysis reveals vectorizability
  - Loop transformations may enhance vectorizability
- Feedback from compiler shows which loops were/ weren't vectorized and why
  - gcc –O3 -ftree-vectorizer-verbose=n (n=1 or higher)

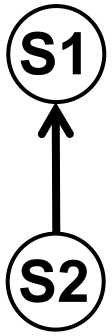
Examples: Acyclic Dependences

```
for (i=0; i<N-1; i++) {
   w[i+1] = x[i]+1;          S1
   y[i] = 2*w[i];            S2
}
```



```
w[1:N-1] = x[0:N-2]+1;   S1
y[0:N-2] = 2*w[0:N-2];   S2
```

```
for (i=0; i<N-1; i++) {
   w[i] = y[i]+1;            S1
   y[i+1] = 2*x[i];          S2
}
```

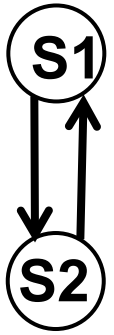


```
y[1:N-1] = 2*x[0:N-2];   S2
w[0:N-2] = y[0:N-2]+1;   S1
```

- Loop Distribution is needed to vectorize multi-statement loops

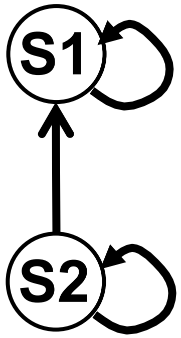
Examples: Cyclic Dependences

```
for (i=0; i<N-1; i++) {
   x[i+1] = y[i]+1;        S1
   y[i+1] = 2*x[i];        S2
}
```



Not vectorizable

```
for (i=0; i<N-1; i++) {
   x[i+1] = y[i]+x[i];      S1
   y[i+1] = 2*y[i]+z[i];  S2
}
```

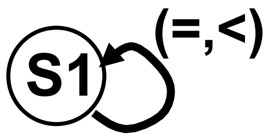


Not vectorizable

- Cycles in dependence graph prevent vectorization

Example: Vectorizable After Transform

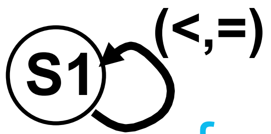
```
for (j=0; j<N; j++)
  for(i=1; i<N; i++)
    A[i][j] = A[i-1][j]+1; S1
```



Not vectorizable

$\downarrow$ Loop Permutation

```
for (i=1; i<N; i++)
  for(j=0; j<N; j++)
    A[i][j] = A[i-1][j]+1; S1
```



```
for (i=1; i<N; i++)
  A[i][0:N-1] = A[i-1][0:N-1]+1; S1
```

- Can vectorize if dependence is carried at an outer loop

Example: Vectorizable After Transform

```
for (i=0; i<N; i++) {
  sum = 0.0;
  for(j=0; j<N; j++)
    sum += A[j][i]*A[j][i];
 x[i] = sum; }
```

$\Rightarrow$

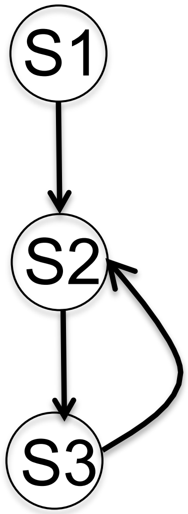
```
for (i=0; i<N; i++) sum[i]=0;
for (i=0; i<N; i++) {
  for(j=0; j<N; j++)
    sum[j] += A[i][j]*A[i][j]; }
for (i=0; i<N; i++) x[i] = sum[i];
```

- Anti-dependences in loops may be removable via scalar expansion

Example: Cyclic Dependence

A loop can be partially vectorized

```
   for (int i=1;i<LEN;i++){
S1   a[i] = b[i] + c[i];
S2   d[i] = a[i] + e[i-1];
S3   e[i] = d[i] + c[i];
   }
```


$$\begin{tikzcd} S1 \arrow[d] \\ S2 \arrow[d] \\ S3 \arrow[u, bend right] \end{tikzcd}$$

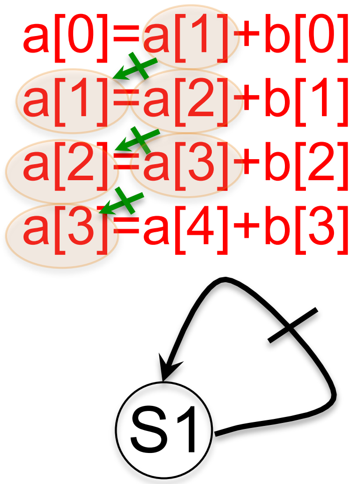
S1 can be vectorized
S2 and S3 cannot be vectorized (as they are)

Source: Maria Garzaran and David Padua

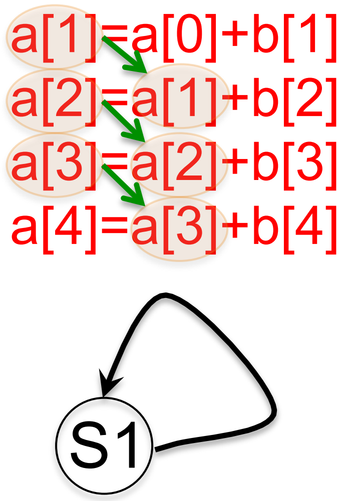
Example: Cyclic Dependence

```
   for (int i=0;i<LEN-1;i++){
S1   a[i]=a[i+1]+b[i];
   }
```

```
   for (int i=1;i<LEN;i++){
S1   a[i]=a[i-1]+b[i];
   }
```



Self-antidependence can be vectorized

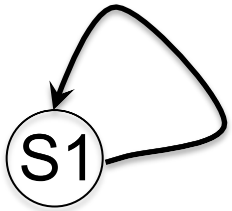


Self true-dependence can not vectorized (as it is)

Source: Maria Garzaran and David Padua

Example: Cyclic Dependence

```
      for (int i = 0; i < LEN-1; i++) {
       for (int j = 0; j < LEN; j++)
S1       a[i+1][j] = a[i][j] + b;
      }
```



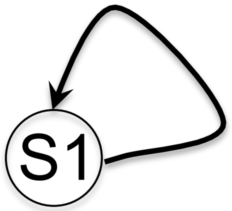
Can this loop be vectorized?

```
i=0, j=0:  a[1][0] = a[0][0] + b
     j=1:  a[1][1] = a[0][1] + b
     j=2:  a[1][2] = a[0][2] + b
i=1  j=0:  a[2][0] = a[1][0] + b
     j=1:  a[2][1] = a[1][1] + b
     j=2:  a[2][2] = a[1][2] + b
```

Source: Maria Garzaran and David Padua

Example: Cyclic Dependence

```
     for (int i = 0; i < LEN-1; i++) {
      for (int j = 0; j < LEN; j++)
S1      a[i+1][j] = a[i][j] + (float) 1.0;
     }
```



Can this loop be vectorized?

```
i=0, j=0:  a[1][0] = a[0][0] + 1
     j=1:  a[1][1] = a[0][1] + 1
     j=2:  a[1][2] = a[0][2] + 1
i=1  j=0:  a[2][0] = a[1][0] + 1
     j=1:  a[2][1] = a[1][1] + 1
     j=2:  a[2][2] = a[1][2] + 1
```

Dependences occur in the outermost loop.
- outer loop runs serially
- inner loop can be vectorized

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

Source: Maria Garzaran and David Padua

AoS versus SoA

- Array-of-Structures often prevents vectorization

```
typedef struct
{ float x;float y; float z;
  float dsquared;
} coords;
```

```
coords nbodies_aos[N];
```

```
for(i=0;i<N;i++)
 nbodies_aos[i].dsquared += nbodies_aos[i].x*nbodies_aos[i].x
                          + nbodies_aos[i].y*nbodies_aos[i].y
                          + nbodies_aos[i].z*nbodies_aos[i].z;
```

AoS versus SoA

- Converting to Structure-of-Arrays can enable vectorization

```
typedef struct
{
  float x[N]; float y[N]; float z[N];
  float dsquared[N];
} coords_arr;
```

```
coords_arr nbodies_soa;
```

```
for(i=0;i<N;i++)
 nbodies_soa.dsquared[i] += nbodies_soa.x[i]*nbodies_soa.x[i]
                            + nbodies_soa.y[i]*nbodies_soa.y[i]
                            + nbodies_soa.z[i]*nbodies_soa.z[i];
```

Summary

- Vectorization requires independent operations with stride 0/1 operands
- Innermost loop vectorization is most common; outerloop vectorization is also possible
- Loop permutation can enable vectorization by 1) making innermost loop parallel, and 2) make stride 0/1
- Anti-dependences due to scalars can inhibit vectorization; scalar expansion can enable vectorization
- Loop distribution enables vectorization of multi-statement loops
- Cycle of flow-dependences in the dependence graph prevents vectorization
- Conditional statements in innermost loop prevents vectorization
- Array-of-Structures typically inhibits vectorization; converting to Structure-of-Arrays can enable vectorization