

CS698L: Vectorization

Swarnendu Biswas

Semester 2019-2020-I
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Material adapted from

M. Garzaran et al. Program Optimization Through Loop Vectorization.

M. Voss. Topics in Loop Vectorization.

Different Levels of Parallelism in Hardware

- Instruction-level Parallelism
 - Microarchitectural techniques like pipelining, OOO execution, and superscalar
- Vector-level Parallelism
 - Use Single Instruction Multiple Data (SIMD) vector processing instructions and units
- Thread-level Parallelism
 - Hyperthreading

Vectorization

- Process of transforming a scalar operation on single data elements at a time (SISD) to an operation on multiple data elements at once (SIMD)
- Loop vectorization transforms a program so that the same operation is performed at the same time on several vector elements

Vectorization

```
double *a, *b, *c;  
for (int i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

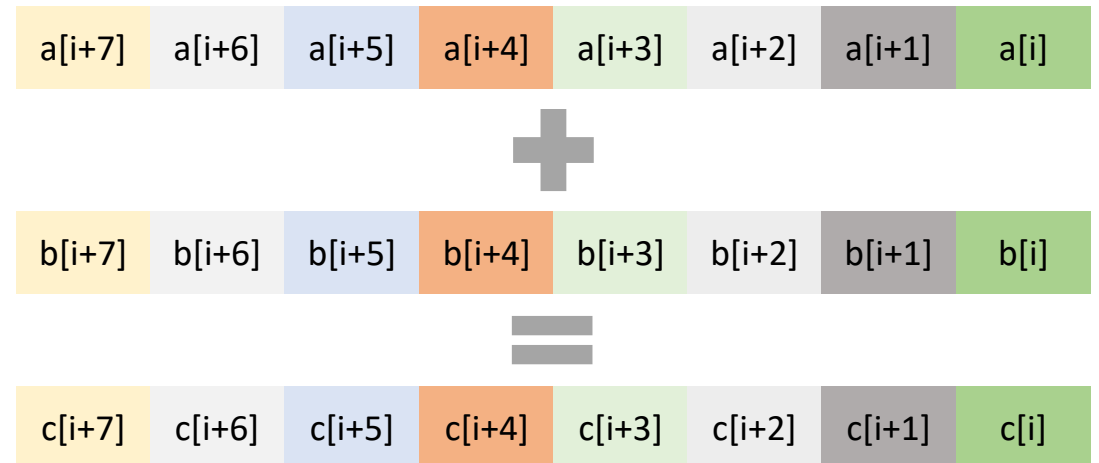
Scalar mode

- One instruction produces one result
 - vaddsd/vaddss



Vector mode

- One instruction can produce multiple results
 - vaddpd/vaddps



Vectorization

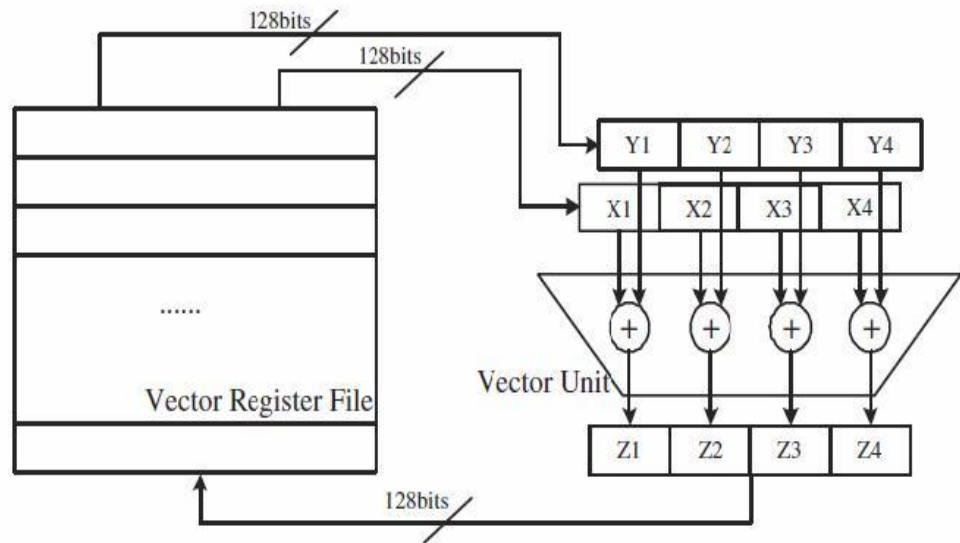
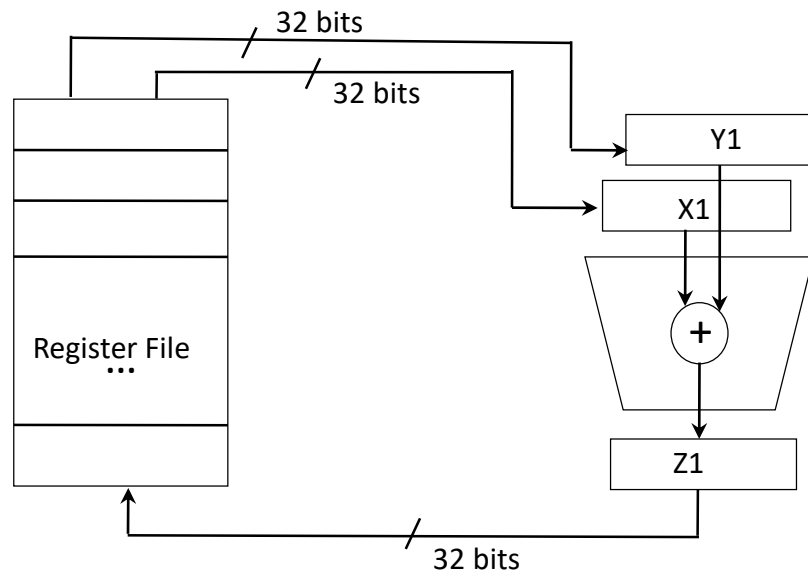
n
times

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

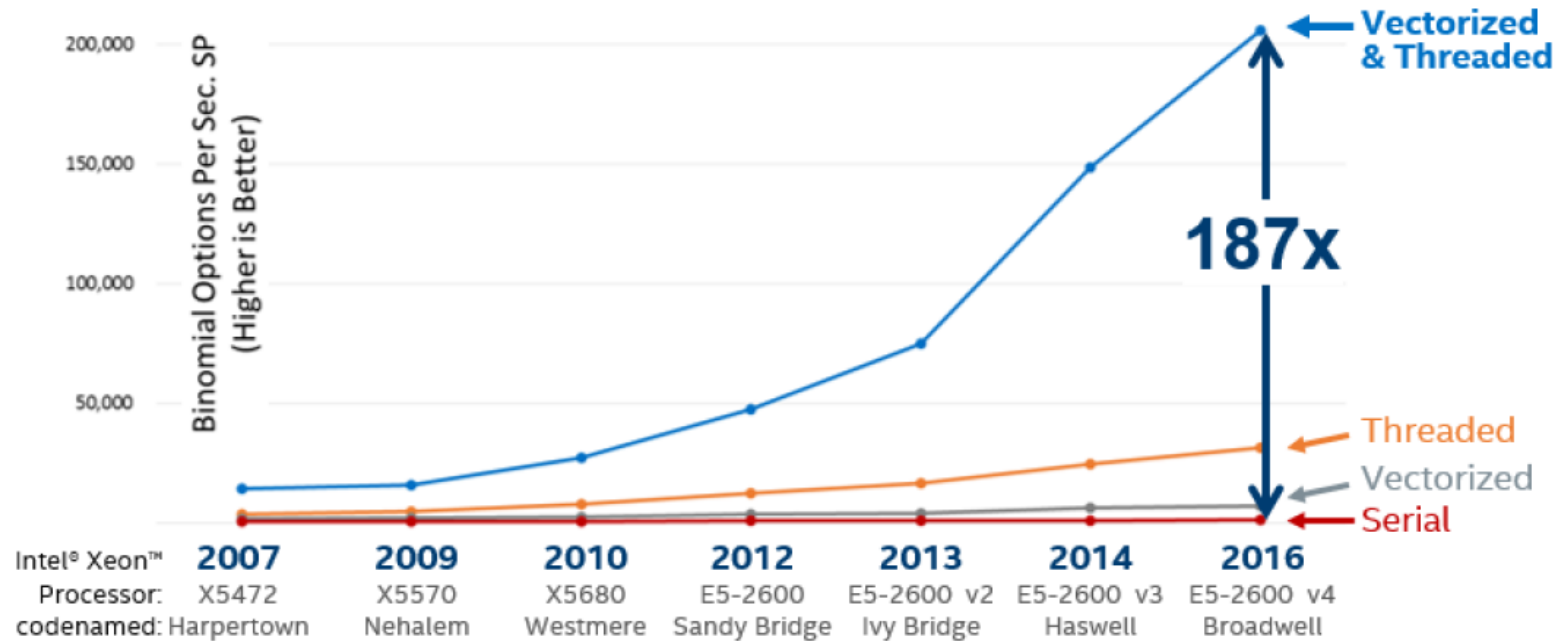
```
for (i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

n/4
times

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



The combined effect of vectorization and threading



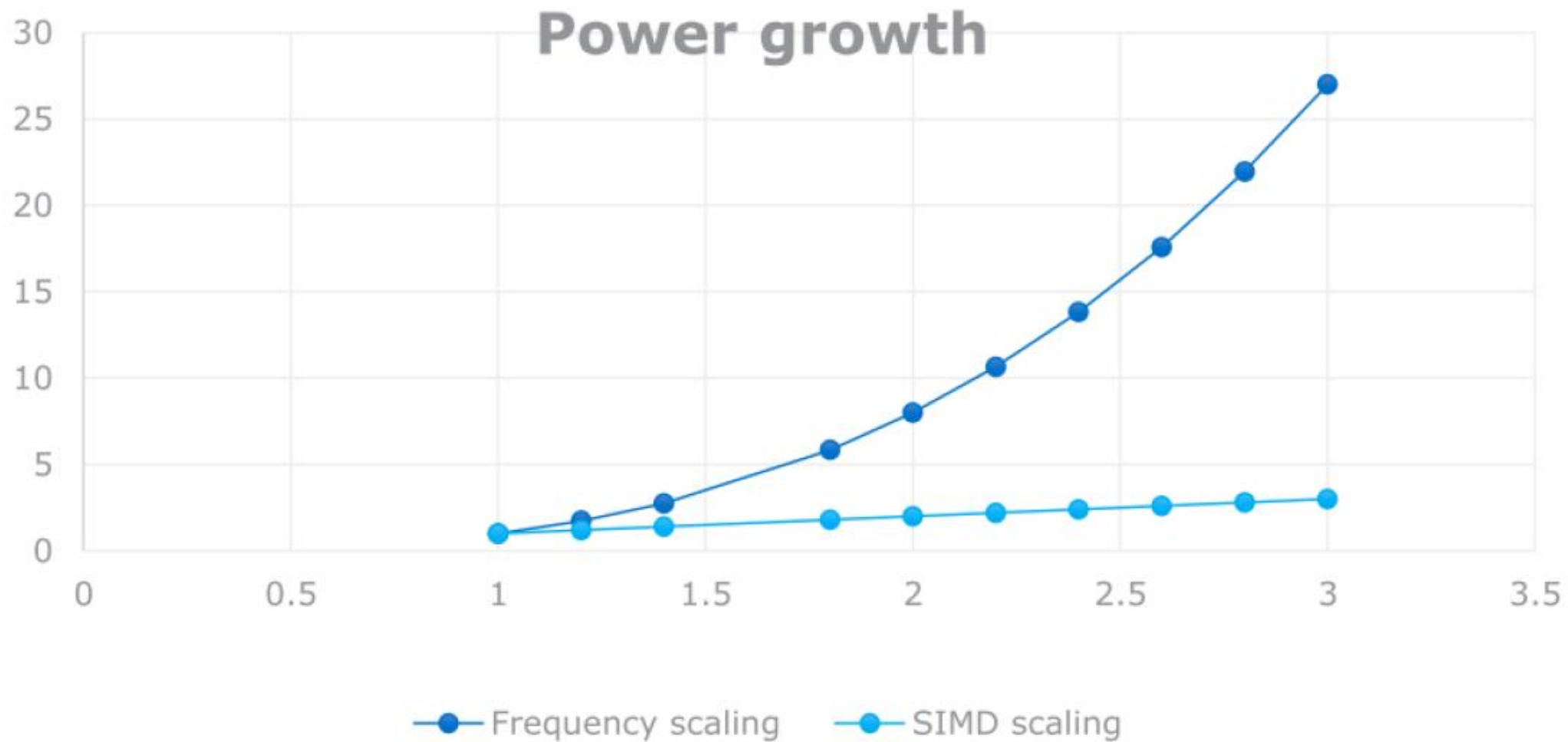
The Difference Is Growing With Each New Generation of Hardware

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to <http://www.intel.com/performance> [Configurations](#) at the end of this presentation.

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.





Intel-Supported SIMD Extensions

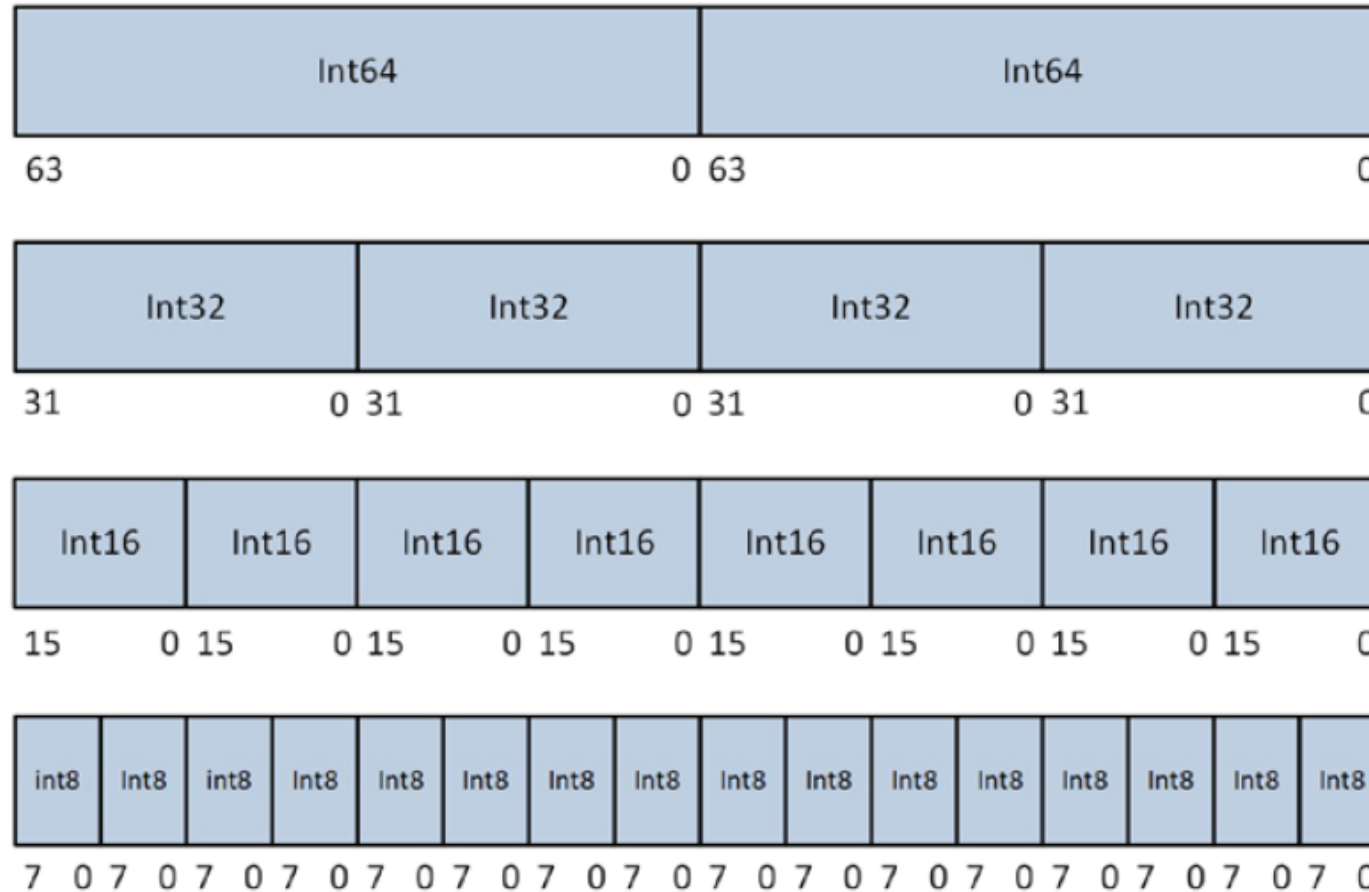
SIMD extensions	Width (bits)	Dual precision (64 bit) calculations	Single precision (32 bit) calculations	Introduced
SSE2/SSE3/SSE4	128	2	4	~2001-2007
AVX/AVX2	256	4	8	~2011-2015
AVX-512	512	8	16	~2017

Other platforms that support SIMD have different extensions

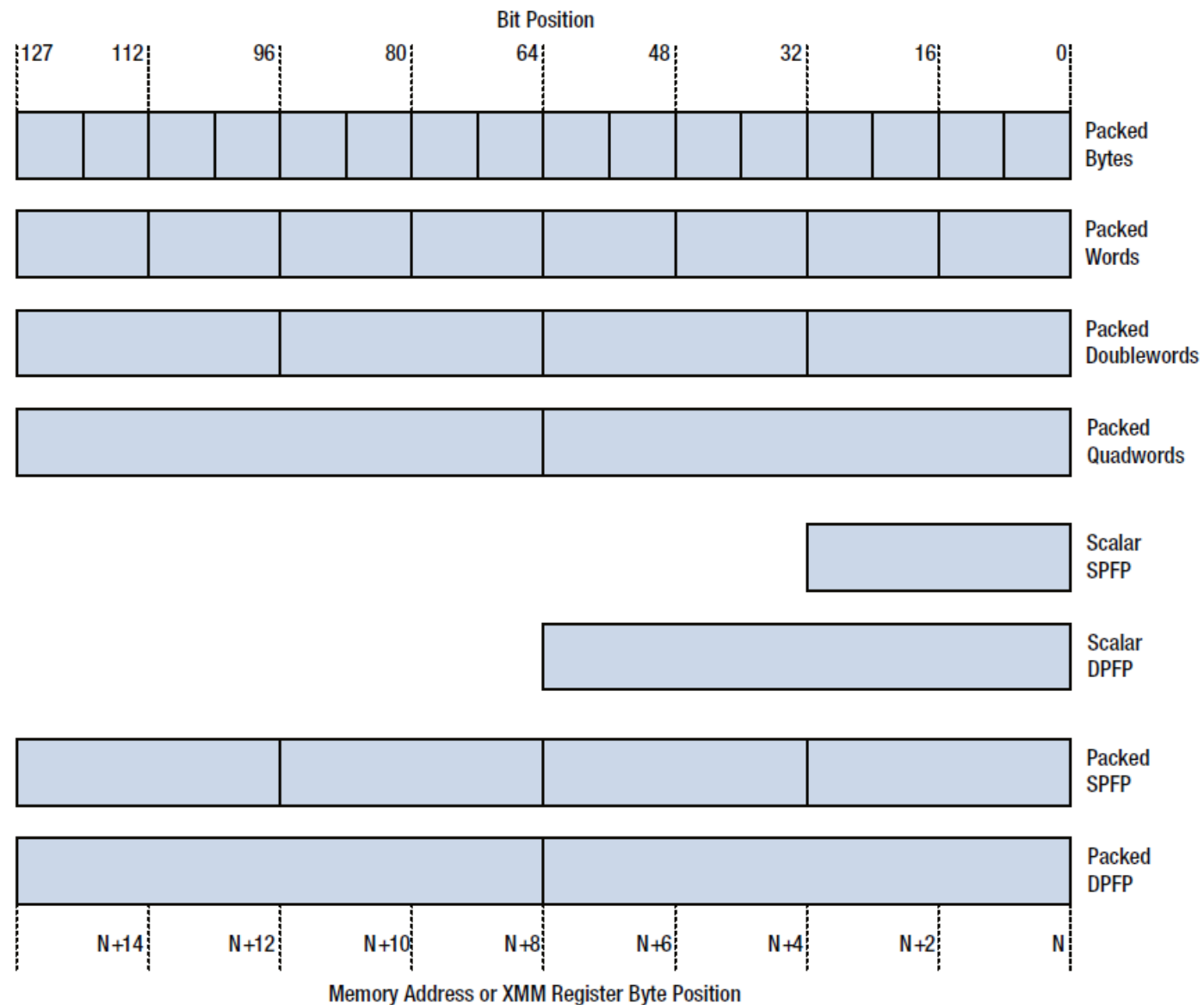
SIMD Vectorization

- Use of SIMD units can speed up the program
- Intel SSE has 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

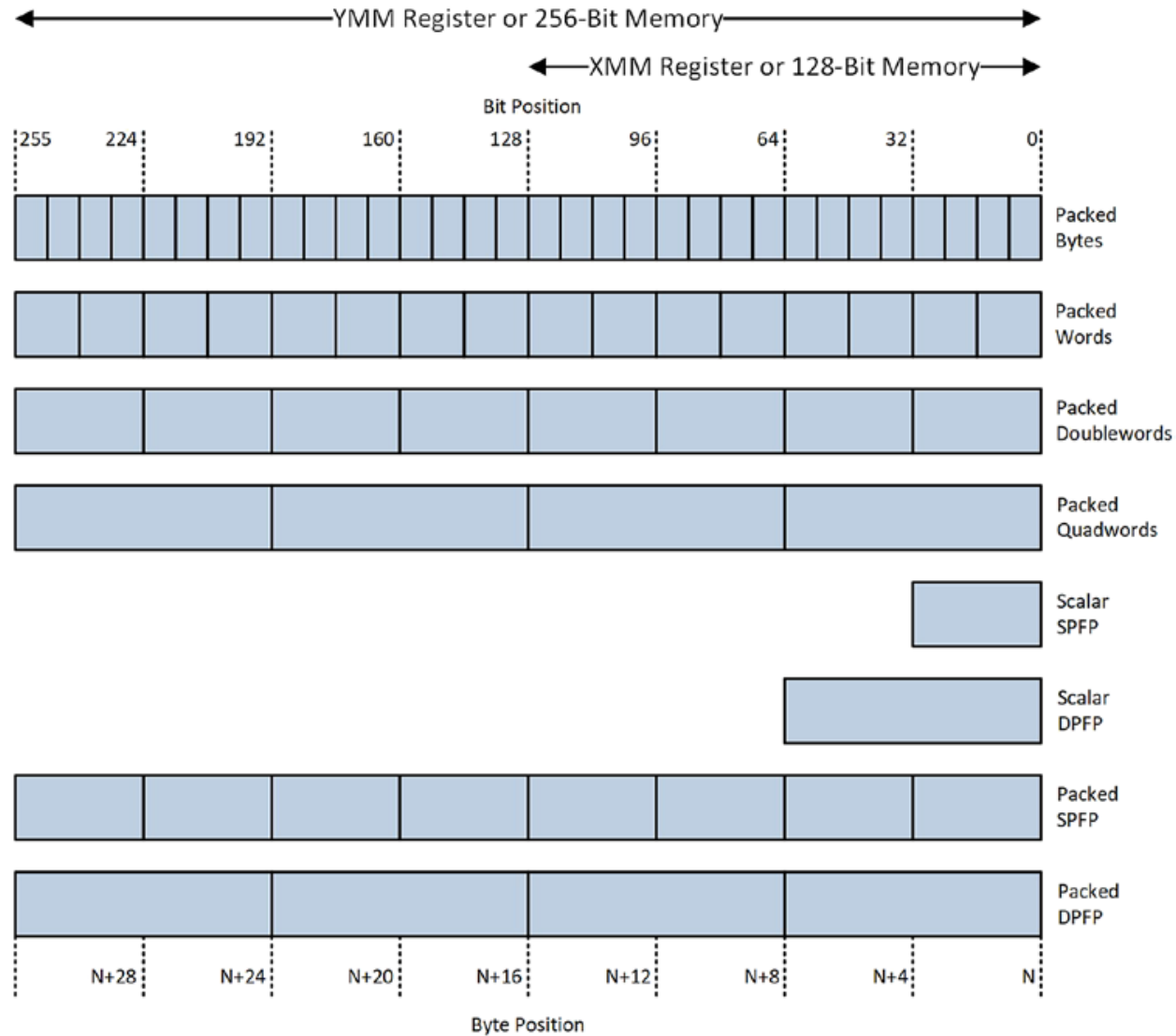
128-bit wide operands using integers



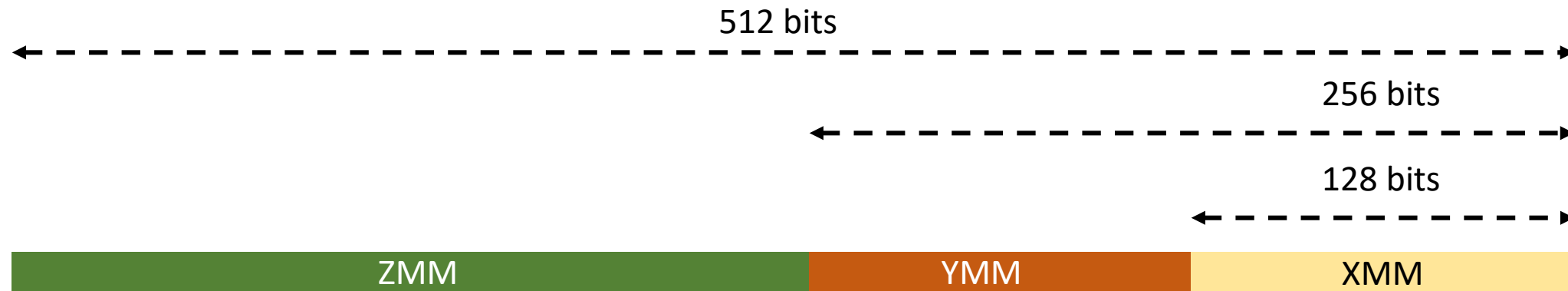
SSE Data Types



AVX2 Data Types



Intel-Supported SIMD Extensions



64-bit architecture		
SSE	XMM0-XMM15	
AVX	YMM0-YMM15	Low- order 128 bits of each YMM register are aliased to a corresponding XMM register
AVX-512	ZMM-ZMM31	low-order 256 and 128 bits are aliased to registers YMM0-YMM31 and XMM0-XMM31 respectively

x86-64 Vector Operations

[] – required
() - optional

- Example instructions
 - Move: (V)MOV[A/U]P[D/S]
 - Comparing: (V)CMP[P/S][D/S]
 - Arithmetic operations: (V)[ADD/SUB/MUL/DIV][P/S][D/S]
- Instruction decoding
 - V – AVX
 - P, S – packaged, scalar
 - A, U – aligned, unaligned
 - D, S – double, single
 - B, W, D, Q – byte, word, doubleword, quadword integers

x86-64 Vector Operations

Instruction

Explanation

`movss xmm2, xmm1`

`xmm2 = xmm2 + xmm1 (SSE/SSE2)`

`vmovapd ymmword ptr [edi], ymm1`

Move aligned packed double-precision floating-point values from ymm2/mem to ymm1.

AVX Scalar Floating-Point Instruction Examples

Instruction

Explanation

`vaddss xmm0, xmm1, xmm2`

$\text{xmm0}[31:0] = \text{xmm1}[31:0] + \text{xmm2}[31:0]$
 $\text{xmm0}[127:32] = \text{xmm1}[127:32]$
 $\text{ymm0}[255:128] = 0$

`vaddsd xmm0, xmm1, xmm2`

$\text{xmm0}[63:0] = \text{xmm1}[63:0] + \text{xmm2}[63:0]$
 $\text{xmm0}[127:64] = \text{xmm1}[127:64]$
 $\text{ymm0}[255:128] = 0$

Cumulative (app.) # of Vector Instructions



Copyright © 2015 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

Optimization Notice



Ways to Vectorize Code

Vectorize Code

- Auto-vectorizing compiler
- Vector intrinsics
- Assembly language

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

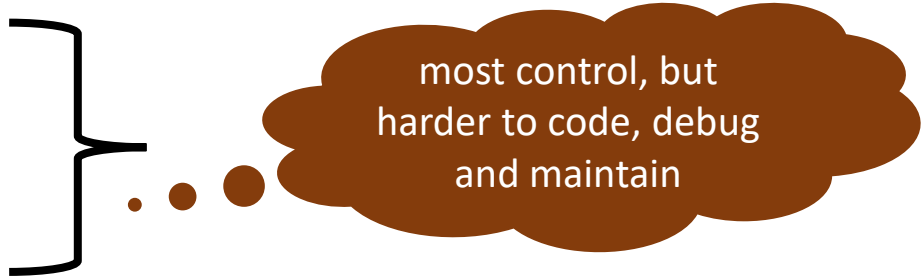
```
void example(){  
    __m128 rA, rB, rC;  
    for (int i = 0; i < LEN; i+=4){  
        rA = _mm_load_ps(&a[i]);  
        rB = _mm_load_ps(&b[i]);  
        rC = _mm_add_ps(rA, rB);  
        _mm_store_ps(&c[i], rC);  
    }  
}
```

```
..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
```

easy, but low
control

hard, but
most control

Vectorize Code

- Auto-vectorization
 - Compiler vectorizes automatically – No code changes
 - Semi auto-vectorization – Use pragmas as hints to guide compiler
 - Explicit vector programming – OpenMP SIMD pragmas
 - SIMD/Vector intrinsics
 - Inline assembly language
- 
- Use SIMD-capable libraries like Intel Math Kernel Library (MKL)

Auto-Vectorization

Transparent to programmers

Compilers can apply other transformations

Portability of code across architectures

- Vectorization instructions may differ but compilers take care of it

Auto-Vectorization

Transparent to programmers

Compilers can apply other transformations

Portability of code across architectures

- Vectorization instructions may differ but compilers take care of it

Compilers may fail to vectorize

- Programmers may give hints to help the compiler
- Programmers may have to manually vectorize their code

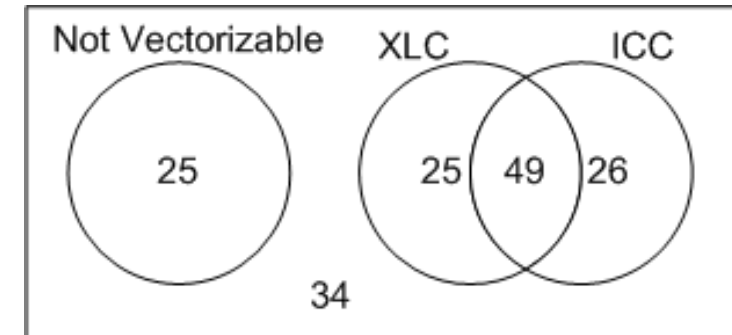
Data Dependences and Vectorization

- Loop dependences guide vectorization
- A statement inside a loop which is not in a cycle of the dependence graph can be vectorized

How well do compilers vectorize?

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

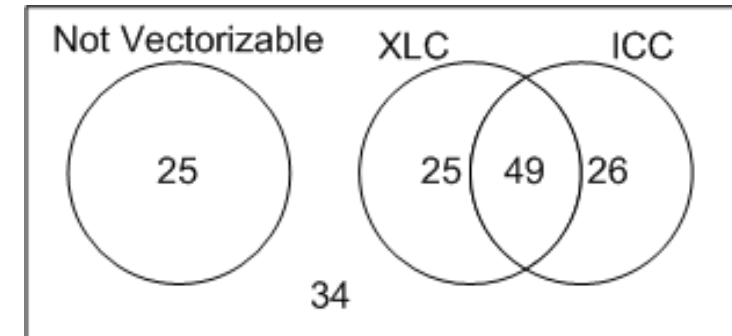
Compiler \ Loops	XLC but not ICC	ICC but not XLC
Vectorized	25	26



How well do compilers vectorize?

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

Compiler \ Loops	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)

Experimental results

- These slides show vectorization results for two different platforms with the following compilers:
 - Report generated by the compiler
 - Execution Time for each platform

Platform 1: Intel Nehalem
Intel Core i7 CPU 920@2.67GHz
Intel ICC compiler, version 11.1
OS Ubuntu Linux 9.04

Platform 2: IBM Power 7
IBM Power 7, 3.55 GHz
IBM xlc compiler, version 11.0
OS Red Hat Linux Enterprise 5.4

The examples use single precision floating point numbers



Compiler directives

```
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];
    }
}
```



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



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

Power 7

Compiler report: Loop was not
vectorized.

Exec. Time scalar code: 2.3

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

Power 7

Compiler report: Loop was
vectorized.

Exec. Time scalar code: 1.6

Exec. Time vector code: 0.6

Speedup: 2.7



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;  
}
```

???

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

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

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

Intel Nehalem

Compiler report: Loop was not vectorized. Vectorization possible but seems inefficient.

Exec. Time scalar code: 3.7

Exec. Time vector code: --

Speedup: --

Intel Nehalem

Compiler report: Permuted loop was vectorized.

scalar code: 1.6

vector code: 0.6

Speedup: 2.6

Intel Nehalem

Compiler report: Permuted loop was vectorized.

scalar code: 1.6

vector code: 0.6

Speedup: 2.6

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

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

IBM Power 7

Compiler report: Loop was not SIMD vectorized

Exec. Time scalar code: 2.0

Exec. Time vector code: --

Speedup: --

IBM Power 7

Compiler report: Loop interchanging applied. Loop was SIMD vectorized

scalar code: 0.4

vector code: 0.2

Speedup: 2.0

IBM Power 7

Compiler report: Loop interchanging applied. Loop was SIMD

scalar code: 0.4

vector code: 0.16

Speedup: 2.7

SSE Intrinsics

```
#define n 1024

__attribute__((aligned(16))) float
a[n], b[n], c[n];

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

```
#include <xmmintrin.h>
#define n 1024

__attribute__((aligned(16))) float
a[n], b[n], c[n];

int main() {
    __m128 rA, rB, rC;
    for (i = 0; i < n; i+=4) {
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC= _mm_mul_ps(rA,rB);
        _mm_store_ps(&c[i], rC);
    }
}
```

Challenges in Vectorization

Data Dependences

- Statements not data dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation
- Loop dependences guide vectorization

A statement inside a loop which is not in a cycle of the dependence graph can be vectorized

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

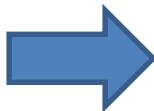


```
a[0:n-1] = b[0:n-1] + 1;
```

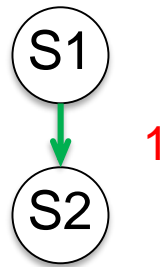
Data dependences and vectorization

- **Main idea:** A statement inside a loop which is not in a cycle of the dependence graph can be vectorized.

```
for (i=1; i<n; i++){  
S1  a[i] = b[i] + 1;  
S2  c[i] = a[i-1] + 2;  
}
```



```
a[1:n] = b[1:n] + 1;  
c[1:n] = a[0:n-1] + 2;
```



Data dependences and transformations

- When cycles are present, vectorization can be achieved by:
 - Separating (distributing) the statements not in a cycle
 - Removing dependences
 - Freezing loops
 - Changing the algorithm

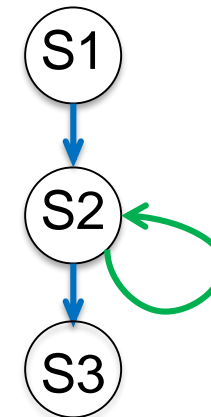


Distributing

```
for (i=1; i<n; i++){  
S1  b[i] = b[i] + c[i];  
S2  a[i] = a[i-1]*a[i-2]+b[i];  
S3  c[i] = a[i] + 1;  
}
```



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



Removing dependencies

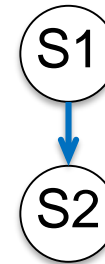
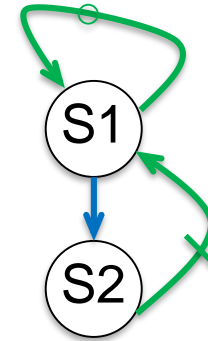
```
for (i=0; i<n; i++){  
S1   a = b[i] + 1;  
S2   c[i] = a + 2;  
}
```



```
for (i=0; i<n; i++){  
S1   a'[i] = b[i] + 1;  
S2   c[i] = a'[i] + 2;  
}  
a=a'[n-1]
```

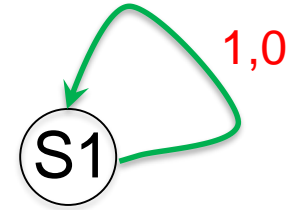


```
S1   a'[0:n-1] = b[0:n-1] + 1;  
S2   c[0:n-1] = a'[0:n-1] + 2;  
a=a'[n-1]
```



Freezing Loops

```
for (i=1; i<n; i++) {  
    for (j=1; j<n; j++) {  
        a[i][j]=a[i][j]+a[i-1][j];  
    }  
}
```



Ignoring (freezing) the outer loop:

```
for (j=1; j<n; j++) {  
    a[i][j]=a[i][j]+a[i-1][j];  
}
```



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



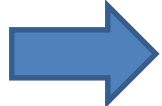
Changing the algorithm

- When there is a recurrence, it is necessary to change the algorithm in order to vectorize.
- Compiler use pattern matching to identify the recurrence and then replace it with a parallel version.
- Examples of recurrences include:
 - Reductions ($S += A[i]$)
 - Linear recurrences ($A[i] = B[i] * A[i-1] + C[i]$)
 - Boolean recurrences ($\text{if } (A[i] > \text{max}) \text{ max} = A[i]$)



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

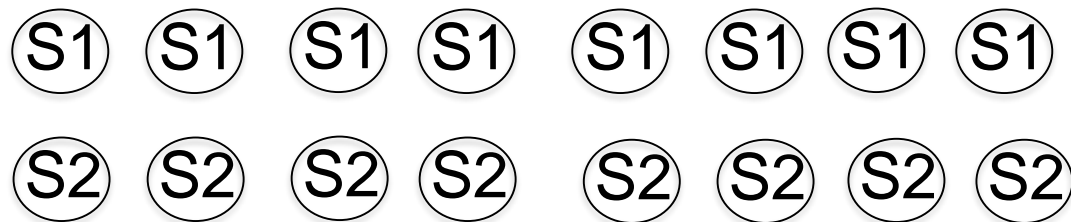


Loop Vectorization

- 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;  
}
```

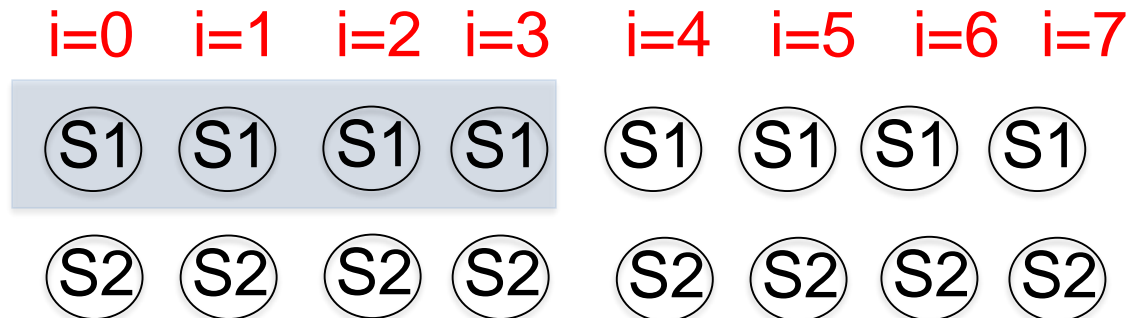
i=0 i=1 i=2 i=3 i=4 i=5 i=6 i=7



Loop Vectorization

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

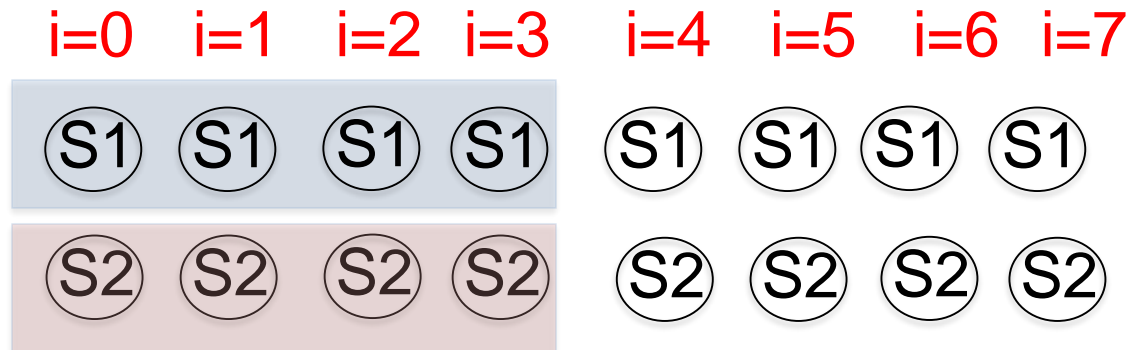
```
for (i=0; i<LEN; i++){  
S1 a[i]=b[i]+(float)1.0;  
S2 c[i]=b[i]+(float)2.0;  
}  
→  
for (i=0; i<LEN; i+=strip_size){  
  for (j=i; j<i+strip_size; j++)  
    a[j]=b[j]+(float)1.0;  
  for (j=i; j<i+strip_size; j++)  
    c[j]=b[j]+(float)2.0;  
}
```



Loop Vectorization

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

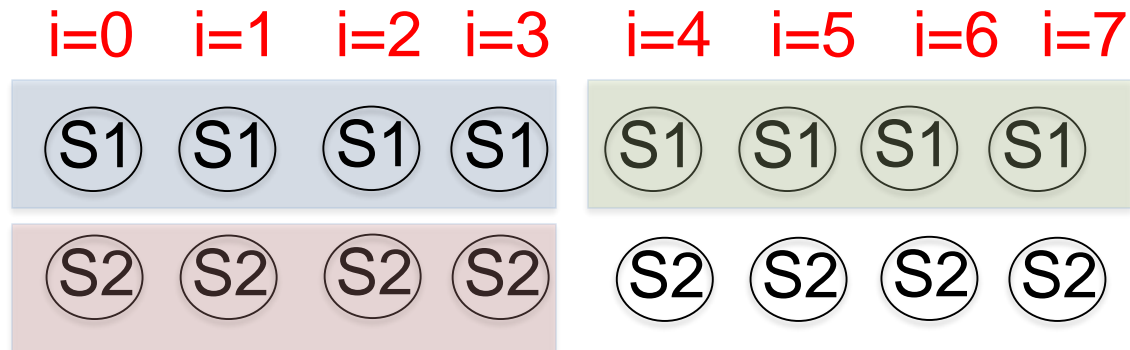
```
for (i=0; i<LEN; i++){  
  S1 a[i]=b[i]+(float)1.0;  
  S2 c[i]=b[i]+(float)2.0;  
}  
→  
for (i=0; i<LEN; i+=strip_size){  
  for (j=i; j<i+strip_size; j++)  
    a[j]=b[j]+(float)1.0;  
  for (j=i; j<i+strip_size; j++)  
    c[j]=b[j]+(float)2.0;  
}
```



Loop Vectorization

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

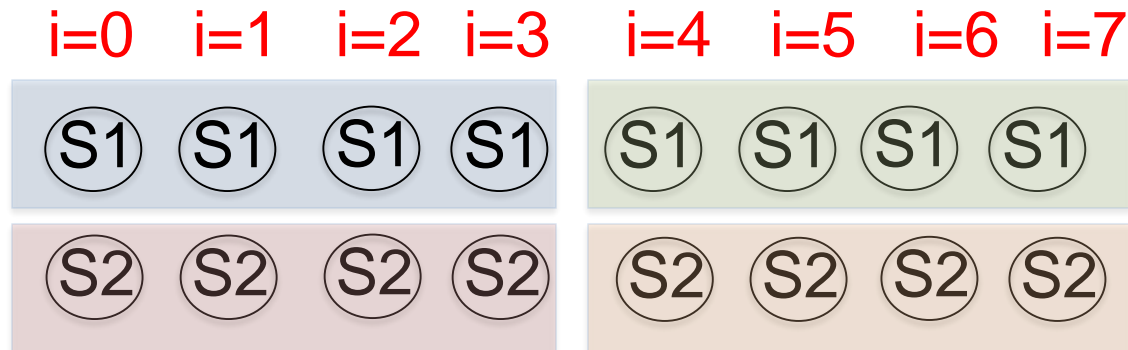
```
for (i=0; i<LEN; i++){  
S1 a[i]=b[i]+(float)1.0;  
S2 c[i]=b[i]+(float)2.0;  
}  
→  
for (i=0; i<LEN; i+=strip_size){  
  for (j=i; j<i+strip_size; j++)  
    a[j]=b[j]+(float)1.0;  
  for (j=i; j<i+strip_size; j++)  
    c[j]=b[j]+(float)2.0;  
}
```



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;  
}
```



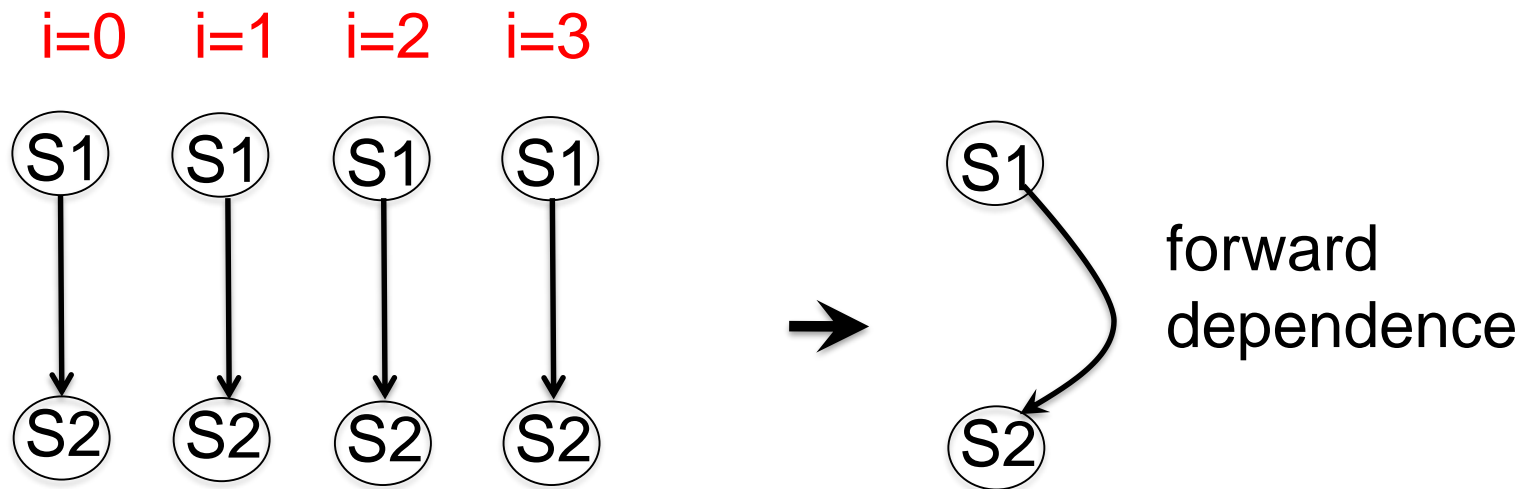
Dependence Graphs and Compiler Vectorization

- No dependences: previous two slides
- Acyclic graphs:
 - All dependences are forward:
 - Vectorized by the compiler
 - Some backward dependences:
 - Sometimes vectorized by the compiler
- Cycles in the dependence graph
 - Self-antidependence:
 - Vectorized by the compiler
 - Recurrence:
 - Usually not vectorized by the the compiler
 - Other examples



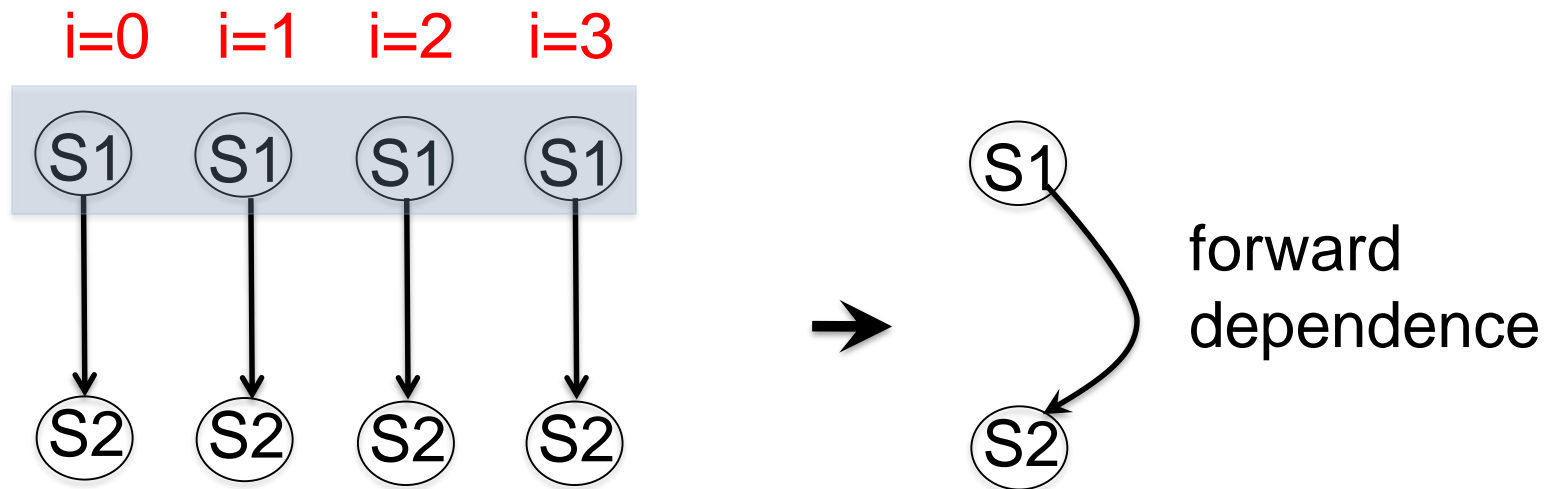
Acyclic Dependence Graphs: Forward Dependences

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



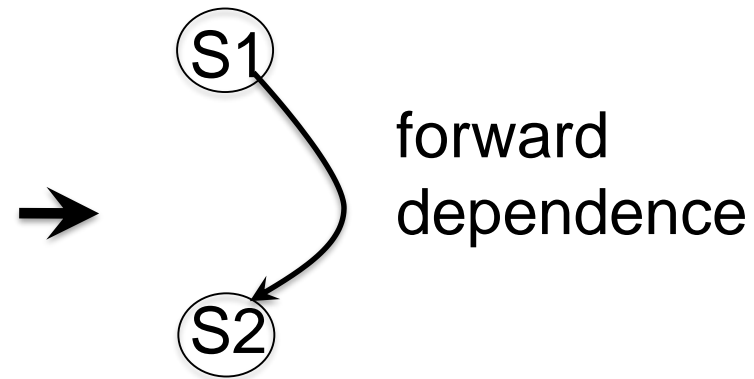
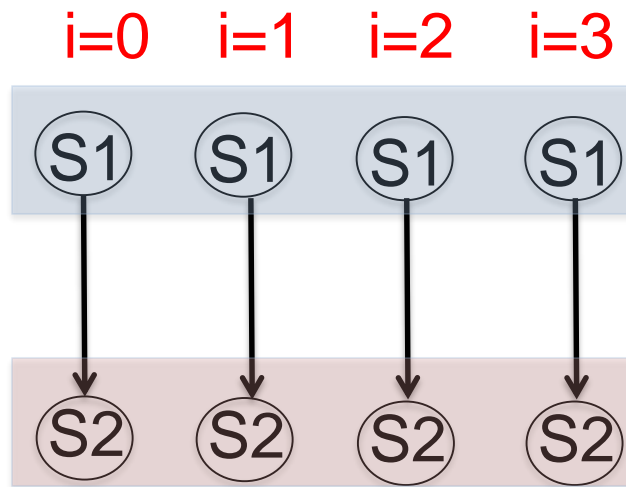
Acyclic Dependence Graphs: Forward Dependences

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



Acyclic Dependence Graphs: Forward Dependences

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



Acyclic Dependence Graphs: Forward Dependences

S113

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

Intel Nehalem

Compiler report: Loop was
vectorized

Exec. Time scalar code: 10.2

Exec. Time vector code: 6.3

Speedup: 1.6

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 3.1

Exec. Time vector code: 1.5

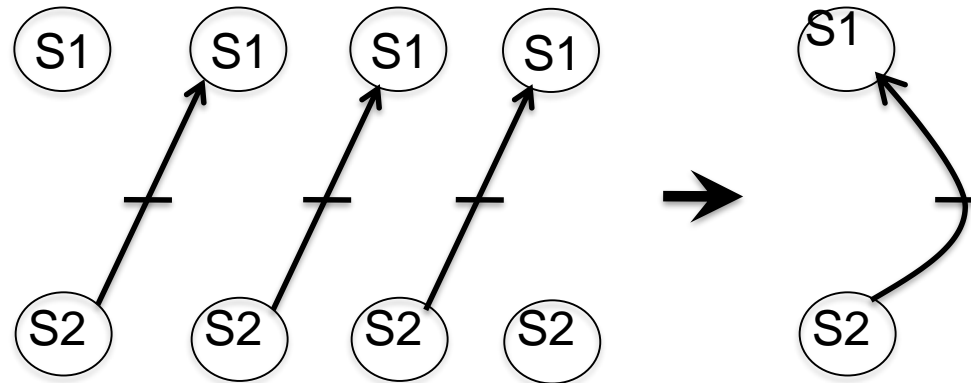
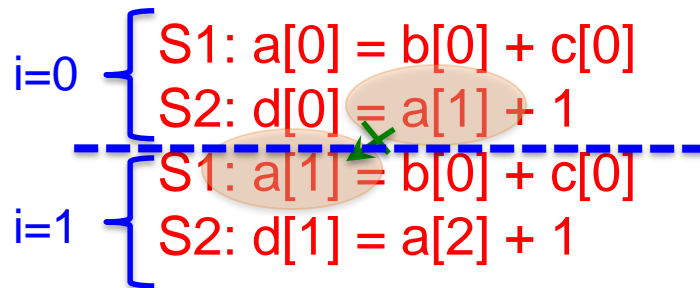
Speedup: 2.0



Acyclic Dependenden Graphs

Backward Dependences (I)

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



This loop cannot be vectorized as it is

Acyclic Dependenden Graphs

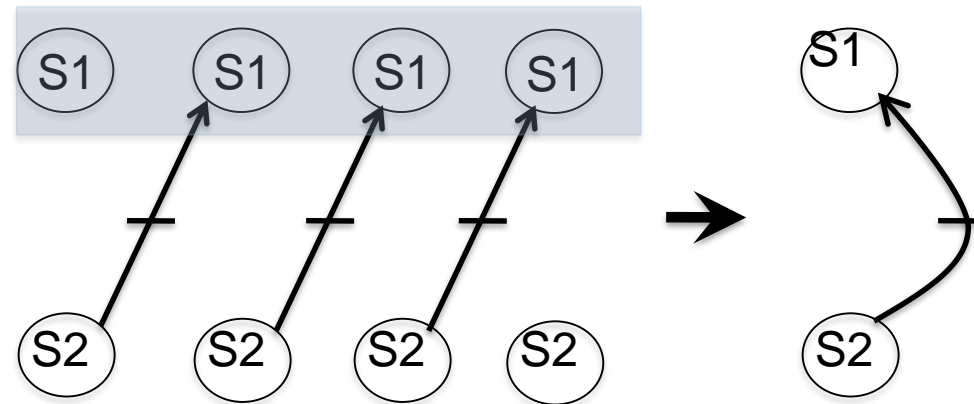
Backward Dependences (I)

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

i=0 { S1: a[0] = b[0] + c[0]
 S2: d[0] = a[1] + 1

i=1 { S1: a[1] = b[0] + c[0]
 S2: d[1] = a[2] + 1

A green 'x' marks the backward dependence from S2 at i=0 to S1 at i=1.



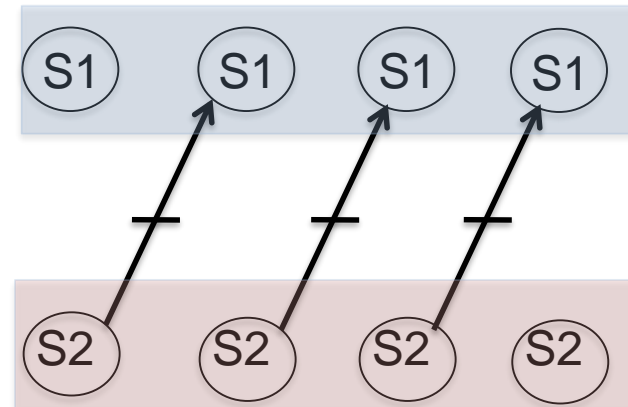
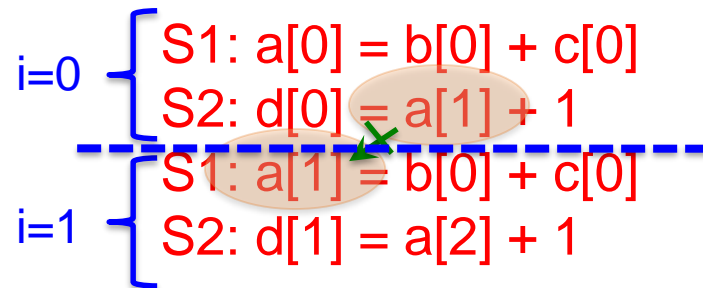
This loop cannot be vectorized as it is



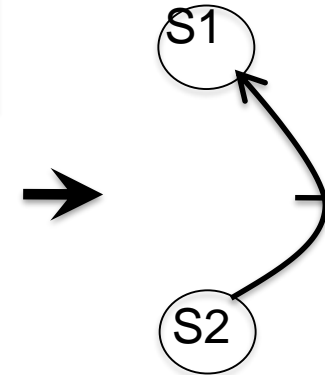
Acyclic Dependenden Graphs

Backward Dependences (I)

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



backward
dependence



This loop cannot be vectorized as it is

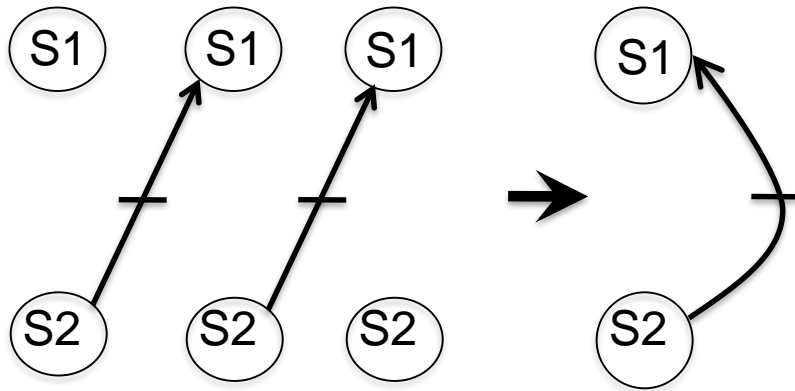


Acyclic Dependenden Graphs

Backward Dependences (I)

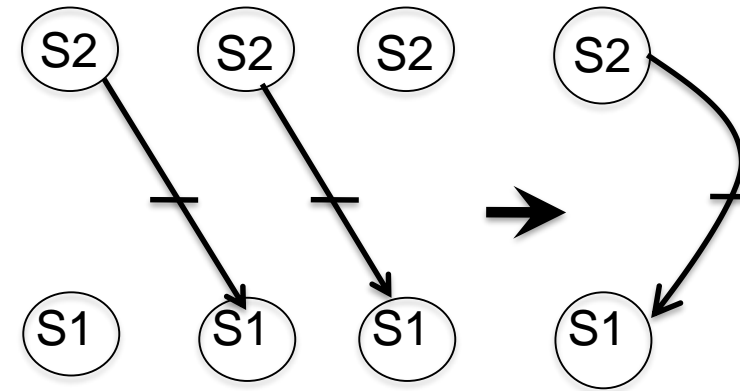
Reorder of statements

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



backward
dependence

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



forward
dependence



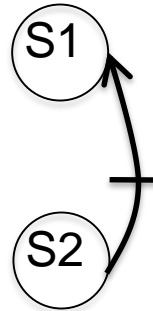
Acyclic Dependenden Graphs

Backward Dependences (I)

S114

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

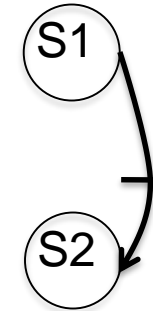
S114



S114_1

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

S114_1



Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 12.6

Exec. Time vector code: --

Speedup: --

Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 10.7

Exec. Time vector code: 6.2

Speedup: 1.72

Speedup vs non-reordered code: 2.03



Acyclic Dependenden Graphs

Backward Dependences (I)

S114

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

S114_1

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

The IBM XLC compiler generated the same code in both cases

S114

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 1.2

Exec. Time vector code: 0.6

Speedup: 2.0

S114_1

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 1.2

Exec. Time vector code: 0.6

Speedup: 2.0

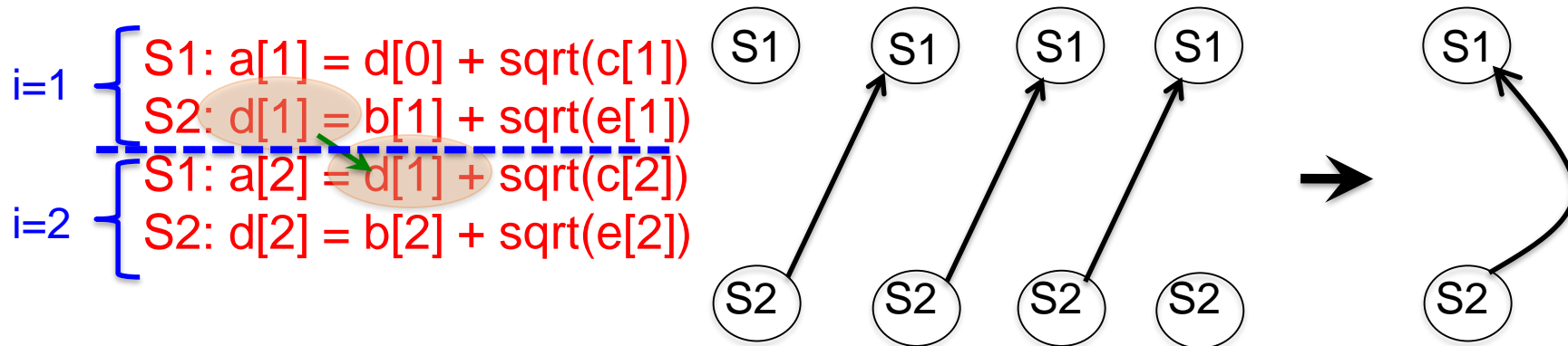


Acyclic Dependenden Graphs

Backward Dependences (II)

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

backward
dependence



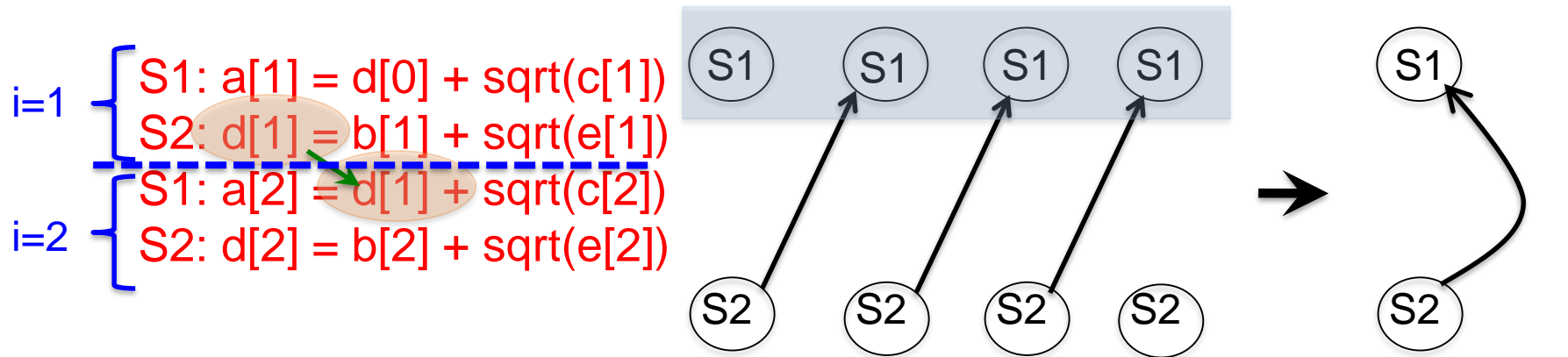
This loop cannot be vectorized as it is



Acyclic Dependenden Graphs

Backward Dependences (II)

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

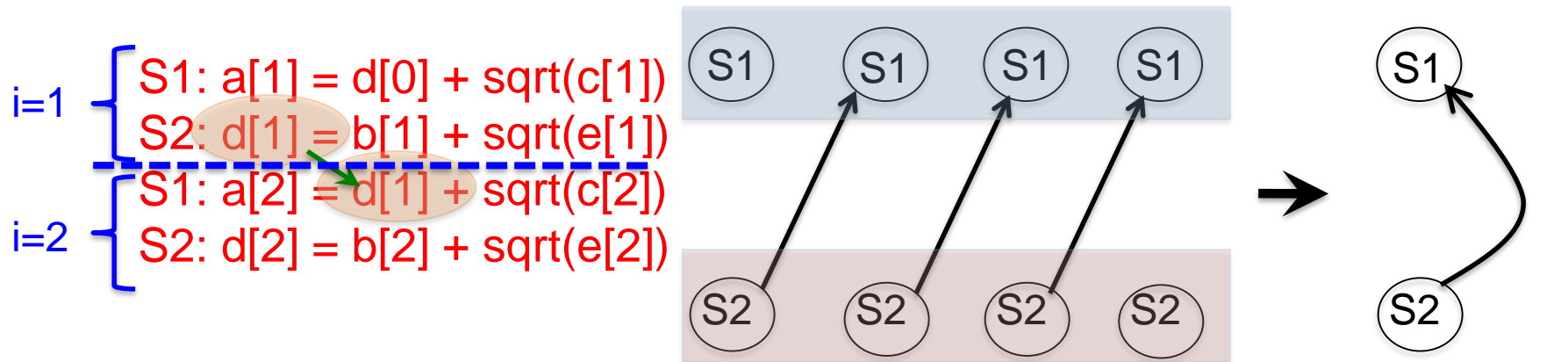


This loop cannot be vectorized as it is

Acyclic Dependenden Graphs

Backward Dependences (II)

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

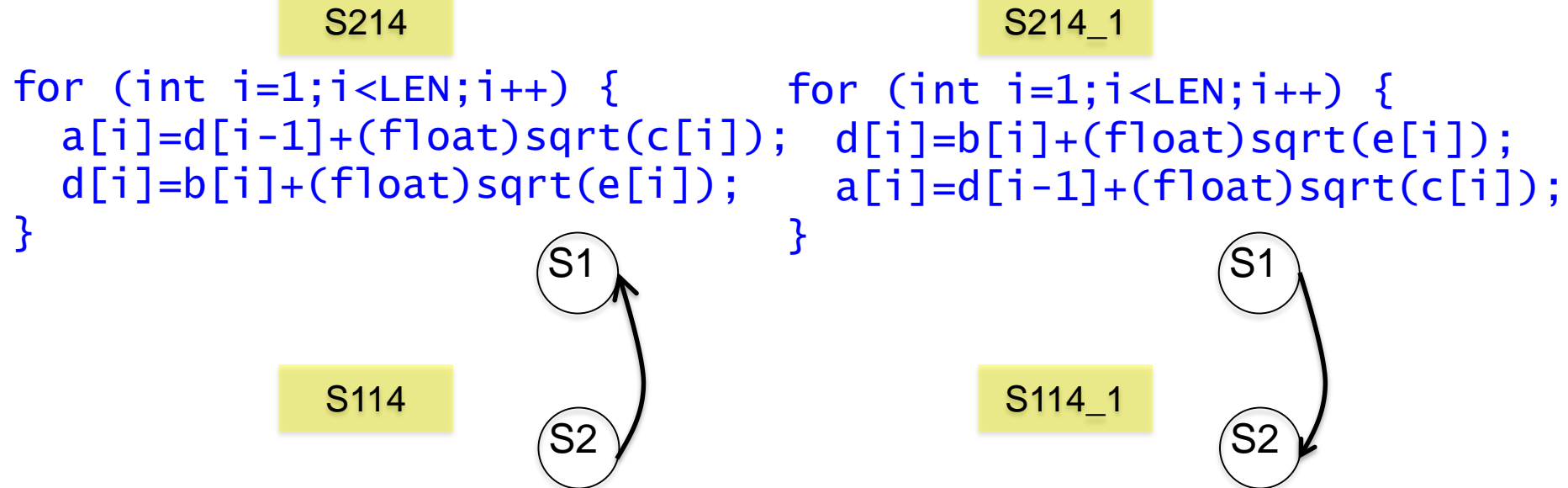


This loop cannot be vectorized as it is



Acyclic Dependenden Graphs

Backward Dependences (II)



Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 7.6

Exec. Time vector code: --

Speedup: --

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 (II)

S114

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

S114_1

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

The IBM XLC compiler generated the same code in both cases

S114

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 3.3

Exec. Time vector code: 1.8

Speedup: 1.8

S114_1

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 3.3

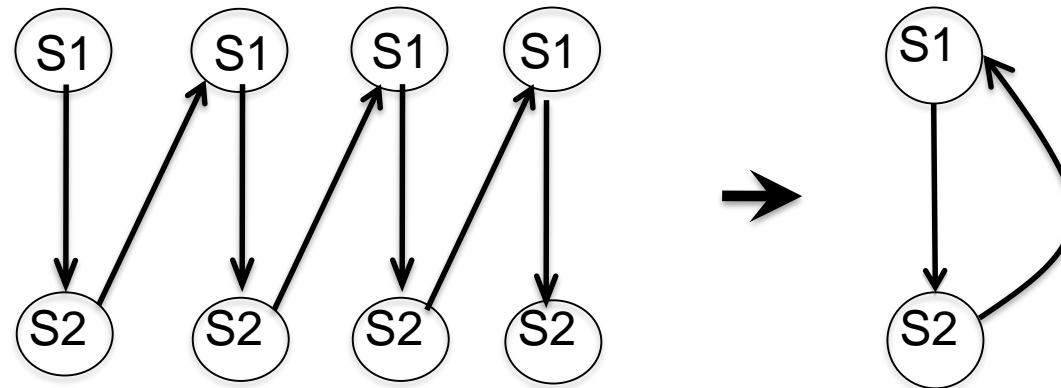
Exec. Time vector code: 1.8

Speedup: 1.8



Cycles in the DG (I)

```
for (int i=0;i<LEN-1;i++){  
S1  b[i]  = a[i] + (float) 1.0;  
S2  a[i+1] = b[i] + (float) 2.0;  
}
```

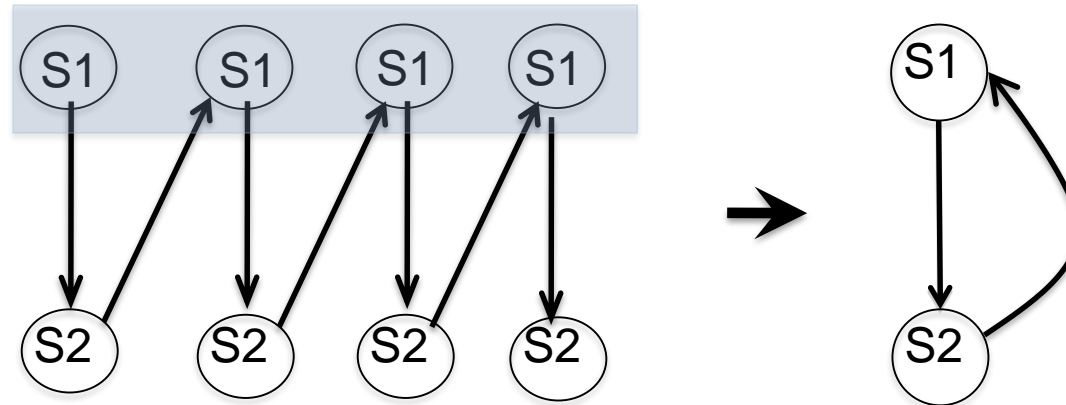


This loop cannot be vectorized (as it is)
Statements cannot be simply reordered



Cycles in the DG (I)

```
for (int i=0;i<LEN-1;i++){  
S1  b[i]  = a[i] + (float) 1.0;  
S2  a[i+1] = b[i] + (float) 2.0;  
}
```

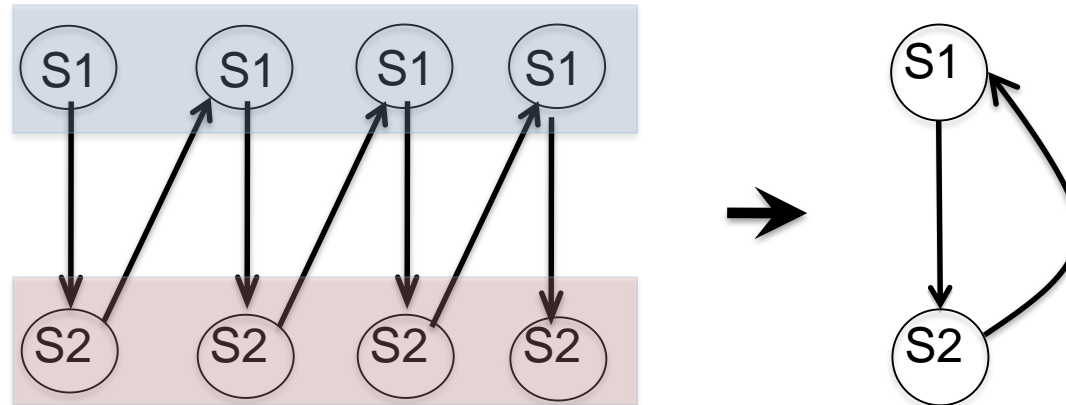


This loop cannot be vectorized (as it is)
Statements cannot be simply reordered



Cycles in the DG (I)

```
for (int i=0;i<LEN-1;i++){  
S1  b[i]  = a[i] + (float) 1.0;  
S2  a[i+1] = b[i] + (float) 2.0;  
}
```



This loop cannot be vectorized (as it is)
Statements cannot be simply reordered



Cycles in the DG (I)

S115

```
for (int i=0;i<LEN-1;i++){  
    b[i]    = a[i] + (float) 1.0;  
    a[i+1] = b[i] + (float) 2.0;  
}
```

S115

Intel Nehalem

Compiler report: Loop was not vectorized.

Existence of vector dependence

Exec. Time scalar code: 12.1

Exec. Time vector code: --

Speedup: --



Cycles in the DG (I)

S115

```
for (int i=0;i<LEN-1;i++){  
    b[i]    = a[i] + (float) 1.0;  
    a[i+1] = b[i] + (float) 2.0;  
}
```

S115

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 3.1

Exec. Time vector code: 2.2

Speedup: 1.4



Cycles in the DG (I)

S115

```
for (int i=0;i<LEN-1;i++){  
    b[i]    = a[i] + (float) 1.0;  
    a[i+1] = b[i] + (float) 2.0;  
}
```

The IBM XLC compiler applies
forward substitution and reordering
to vectorize the code

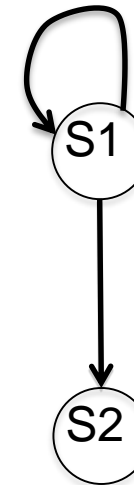
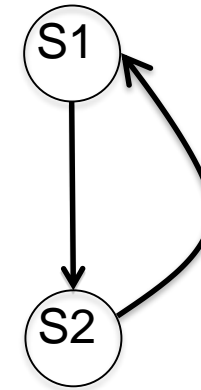
compiler generated code

This loop is
not vectorized

```
for (int i=0;i<LEN-1;i++){  
    a[i+1]=a[i]+(float)1.0+(float)2.0;
```

This loop is
vectorized

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



Cycles in the DG (I)

S115

```
for (int i=0;i<LEN-1;i++){  
    b[i]  =a[i]+(float)1.0;  
    a[i+1]=b[i]+(float)2.0;  
}
```

S215

```
for (int i=0;i<LEN-1;i++){  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i];  
    a[i+1]=b[i]+(float)2.0;  
}
```

Will the IBM XLC compiler
vectorize this code as before?



Cycles in the DG (I)

S115

```
for (int i=0;i<LEN-1;i++){  
    b[i]  =a[i]+(float)1.0;  
    a[i+1]=b[i]+(float)2.0;  
}
```

S215

```
for (int i=0;i<LEN-1;i++){  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i];  
    a[i+1]=b[i]+(float)2.0;  
}
```

Will the IBM XLC compiler
vectorize this code as before?

To vectorize, the compiler needs to do this

```
for (int i=0;i<LEN-1;i++)  
    a[i+1]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i]+(float)2.0;  
  
for (int i=0;i<LEN-1;i++)  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i]+(float) 1.0;
```



Cycles in the DG (I)

S115

```
for (int i=0;i<LEN-1;i++){  
    b[i]  =a[i]+(float)1.0;  
    a[i+1]=b[i]+(float)2.0;  
}
```

S215

```
for (int i=0;i<LEN-1;i++){  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i];  
    a[i+1]=b[i]+(float)2.0;  
}
```

Will the IBM XLC compiler
vectorize this code as before?

No, the compiler does not
vectorize S215 because
it is not cost-effective

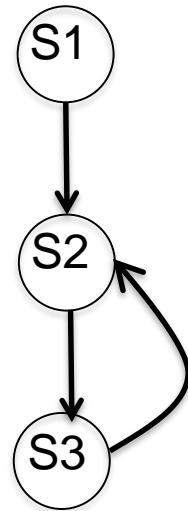
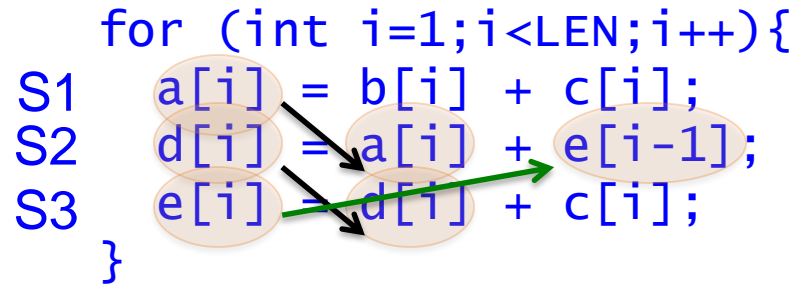
```
for (int i=0;i<LEN-1;i++)  
    a[i+1]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i]+(float)2.0;  
  
for (int i=0;i<LEN-1;i++)  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i]+(float) 1.0;
```



Cycles in the DG (II)

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];  
}
```



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



Cycles in the DG (II)

S116

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

S116

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

S116

Intel Nehalem

Compiler report: Loop was partially vectorized

Exec. Time scalar code: 14.7

Exec. Time vector code: 18.1

Speedup: 0.8

S116

IBM Power 7

Compiler report: Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization

Exec. Time scalar code: 13.5

Exec. Time vector code: --

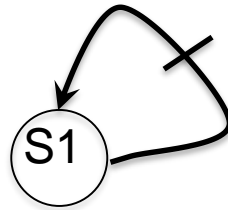
Speedup: --



Cycles in the DG (III)

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

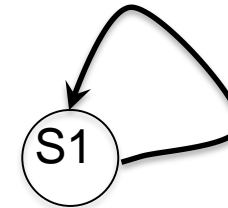
$a[0]=a[1]+b[0]$
 $a[1]=a[2]+b[1]$
 $a[2]=a[3]+b[2]$
 $a[3]=a[4]+b[3]$



Self-antidependence
can be vectorized

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

$a[1]=a[0]+b[1]$
 $a[2]=a[1]+b[2]$
 $a[3]=a[2]+b[3]$
 $a[4]=a[3]+b[4]$

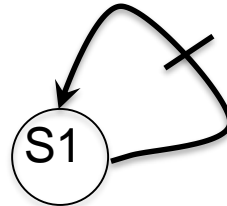


Self true-dependence
can not vectorized
(as it is)

Cycles in the DG (III)

S117

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

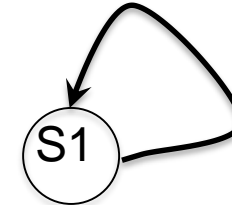


S117

Intel Nehalem
Compiler report: Loop was vectorized
Exec. Time scalar code: 6.0
Exec. Time vector code: 2.7
Speedup: 2.2

S118

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



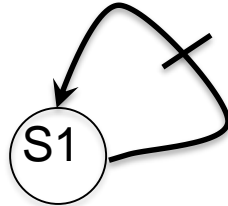
S118

Intel Nehalem
Compiler report: Loop was not vectorized. Existence of vector dependence
Exec. Time scalar code: 7.2
Exec. Time vector code: --
Speedup: --

Cycles in the DG (III)

S117

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



S117

IBM Power 7

Compiler report: Loop was SIMD vectorized

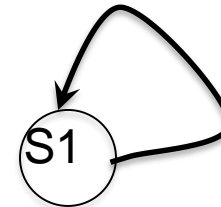
Exec. Time scalar code: 2.0

Exec. Time vector code: 1.0

Speedup: 2.0

S118

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



S118

IBM Power 7

Compiler report: : Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization

Exec. Time scalar code: 7.2

Exec. Time vector code: --

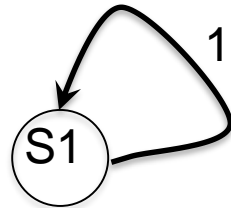
Speedup: --



Cycles in the DG (IV)

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

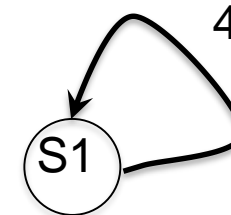
$a[1]=a[0]+b[1]$
 $a[2]=a[1]+b[2]$
 $a[3]=a[2]+b[3]$



Self true-dependence
is **not** vectorized

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

$i=4$ $a[4]=a[0]+b[4]$
 $i=5$ $a[5]=a[1]+b[5]$
 $i=6$ $a[6]=a[2]+b[6]$
 $i=7$ $a[7]=a[3]+b[7]$
 $i=8$ $a[8]=a[4]+b[8]$
 $i=9$ $a[9]=a[5]+b[9]$
 $i=10$ $a[10]=a[6]+b[10]$
 $i=11$ $a[11]=a[7]+b[11]$

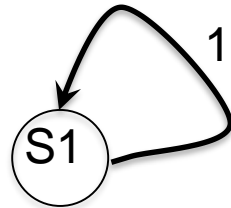


This is also a self-true
dependence. But ...
can it be vectorized?

Cycles in the DG (IV)

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

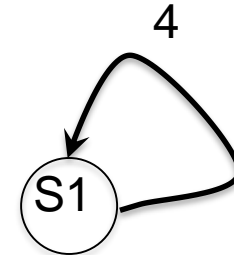
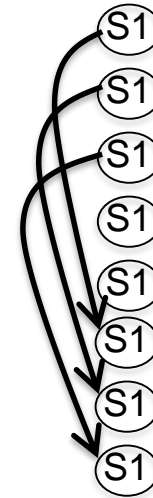
$a[1]=a[0]+b[1]$
 $a[2]=a[1]+b[2]$
 $a[3]=a[2]+b[3]$



Self true-dependence
cannot be vectorized

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

$i=4$ $a[4]=a[0]+b[4]$
 $i=5$ $a[5]=a[1]+b[5]$
 $i=6$ $a[6]=a[2]+b[6]$
 $i=7$ $a[7]=a[3]+b[7]$
 $i=8$ $a[8]=a[4]+b[8]$
 $i=9$ $a[9]=a[5]+b[9]$
 $i=10$ $a[10]=a[6]+b[10]$
 $i=11$ $a[11]=a[7]+b[11]$



Yes, it can be vectorized because the dependence distance is 4, which is the number of iterations that the SIMD unit can execute simultaneously.

Cycles in the DG (IV)

S119

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

Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 8.4

Exec. Time vector code: 3.9

Speedup: 2.1

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 6.6

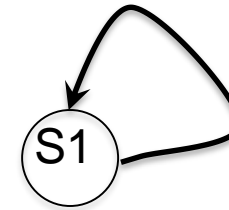
Exec. Time vector code: 1.8

Speedup: 3.7



Cycles in the DG (V)

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

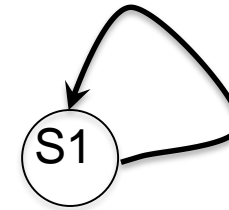


Can this loop be vectorized?

```
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
```

Cycles in the DG (V)

```
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;  
}
```

Cycles in the DG (V)

S121

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

Intel Nehalem

Compiler report: Loop was
vectorized

Exec. Time scalar code: 11.6

Exec. Time vector code: 3.2

Speedup: 3.5

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 3.9

Exec. Time vector code: 1.8

Speedup: 2.1

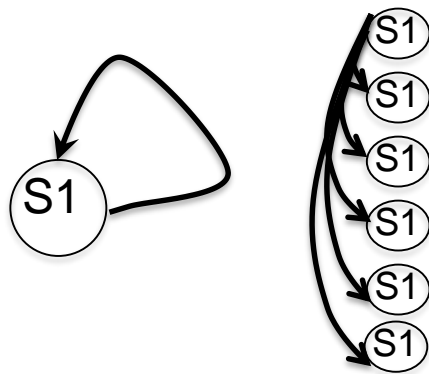


Cycles in the DG (VI)

- Cycles can appear because the compiler does not know if there are dependences

```
for (int i=0;i<LEN;i++){  
S1  a[r[i]] = a[r[i]] * (float) 2.0;  
}
```

Is there a value of i such
that $r[i'] = r[i]$, such that $i' \neq i$?



Compiler cannot resolve the system

To be safe, it considers that a data dependence is possible for every instance of S1

Cycles in the DG (VI)

- The compiler is conservative.
- The compiler only vectorizes when it can prove that it is safe to do it.

```
for (int i=0;i<LEN;i++){  
    r[i] = i;  
    a[r[i]] = a[r[i]]* (float) 2.0;  
}
```

Does the compiler use the info that $r[i] = i$ to compute data dependences?



Cycles in the DG (VI)

S122

```
for (int i=0;i<LEN;i++){  
    a[r[i]]=a[r[i]]*(float)2.0;  
}
```

S123

```
for (int i=0;i<LEN;i++){  
    r[i] = i;  
    a[r[i]]=a[r[i]]*(float)2.0;  
}
```

Does the compiler use the info that $r[i] = i$ to compute data dependences?

S122

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 5.0

Exec. Time vector code: --

Speedup: --

S123

Intel Nehalem

Compiler report: Partial Loop was vectorized

Exec. Time scalar code: 5.8

Exec. Time vector code: 5.7

Speedup: 1.01



Cycles in the DG (VI)

S122

```
for (int i=0;i<LEN;i++){  
    a[r[i]]=a[r[i]]*(float)2.0;  
}
```

S123

```
for (int i=0;i<LEN;i++){  
    r[i] = i;  
    a[r[i]]=a[r[i]]*(float)2.0;  
}
```

Does the compiler use the info that $r[i] = i$ to compute data dependences?

S122

IBM Power 7

Compiler report: Loop was not vectorized because a data dependence prevents SIMD vectorization

Exec. Time scalar code: 2.6

Exec. Time vector code: 2.3

Speedup: 1.1

S123

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 2.1

Exec. Time vector code: 0.9

Speedup: 2.3

Uses forward substitution to vectorize



Dependence Graphs and Compiler Vectorization

- No dependences: Vectorized by the compiler
- Acyclic graphs:
 - All dependences are forward:
 - Vectorized by the compiler
 - Some backward dependences:
 - Sometimes vectorized by the compiler
- Cycles in the dependence graph
 - Self-antidependence:
 - Vectorized by the compiler
 - Recurrence:
 - Usually not vectorized by the the compiler
 - Other examples



Loop Transformations

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



Compiler Directives (I)

- When the compiler does not vectorize automatically due to dependences the programmer can inform the compiler that it is safe to vectorize:

`#pragma ivdep (ICC compiler)`

`#pragma ibm independent_loop (XLC compiler)`

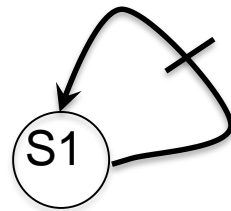


Compiler Directives (I)

- This loop can be vectorized when $k < -3$ and $k \geq 0$.
- Programmer knows that $k \geq 0$

```
for (int i=val; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];
```

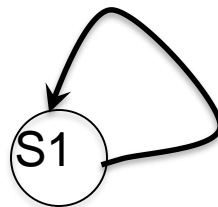
If ($k \geq 0$) \rightarrow no dependence
or self-anti-dependence



$k = 1$

$a[0] = a[1] + b[0]$
 $a[1] = a[2] + b[1]$
 $a[2] = a[3] + b[2]$

Can be vectorized



If ($k < 0$) \rightarrow self-true dependence

$k = -1$

$a[1] = a[0] + b[0]$
 $a[2] = a[1] + b[1]$
 $a[3] = a[2] + b[2]$

Cannot be vectorized



Compiler Directives (I)

- This loop can be vectorized when $k < -3$ and $k \geq 0$.
- Programmer knows that $k \geq 0$

How can the programmer tell the compiler that $k \geq 0$?

```
for (int i=val; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];
```



Compiler Directives (I)

- This loop can be vectorized when $k < -3$ and $k \geq 0$.
- Programmer knows that $k \geq 0$

Intel ICC provides the `#pragma ivdep` to tell the compiler that it is safe to ignore unknown dependences

```
#pragma ivdep
for (int i=val; i<LEN-k; i++)
    a[i]=a[i+k]+b[i];
```

wrong results will be obtained if loop is vectorized when $-3 < k < 0$



Compiler Directives (I)

S124

```
for (int i=0;i<LEN-k;i++)  
    a[i]=a[i+k]+b[i];
```

S124_1

```
if (k>=0)  
    for (int i=0;i<LEN-k;i++)  
        a[i]=a[i+k]+b[i];  
if (k<0)  
    for (int i=0;i<LEN-k;i++)  
        a[i]=a[i+k]+b[i];
```

S124_2

```
if (k>=0)  
    #pragma ivdep  
    for (int i=0;i<LEN-k;i++)  
        a[i]=a[i+k]+b[i];  
if (k<0)  
    for (int i=0;i<LEN-k;i++)  
        a[i]=a[i+k]+b[i];
```

S124 and S124_1

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 6.0

Exec. Time vector code: --

Speedup: --

S124_2

Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 6.0

Exec. Time vector code: 2.4

Speedup: 2.5



Compiler Directives (I)

S124

```
for (int i=0;i<LEN-k;i++) if (k>=0)
    a[i]=a[i+k]+b[i];
```

S124_1

```
for (int i=0;i<LEN-k;i++)
    a[i]=a[i+k]+b[i];
if (k<0)
    for (int i=0;i<LEN-k;i++)
        a[i]=a[i+k]+b[i];
```

S124_2

```
if (k>=0)
    #pragma ibm independent_loop
    for (int i=0;i<LEN-k;i++)
        a[i]=a[i+k]+b[i];
if (k<0)
    for (int i=0;i<LEN-k;i++)
        a[i]=a[i+k]+b[i];
```

S124 and S124_1

IBM Power 7

Compiler report: Loop was not vectorized because a data dependence prevents SIMD vectorization

Exec. Time scalar code: 2.2

Exec. Time vector code: --

Speedup: --

S124_2

#pragma ibm independent_loop
needs AIX OS (we ran the experiments on Linux)



Compiler Directives (II)

- Programmer can disable vectorization of a loop when the when the vector code runs slower than the scalar code

`#pragma novector (ICC compiler)`

`#pragma nosimd (XLC compiler)`

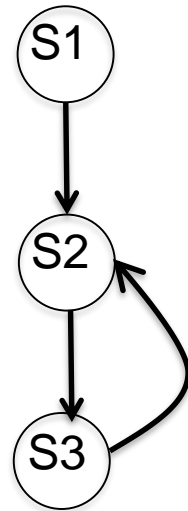


Compiler Directives (II)

Vector code can run slower than scalar code

```
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];  
}
```

Less locality when
executing in vector mode



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



Compiler Directives (II)

S116

```
#pragma novector
```

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

S116

Intel Nehalem

Compiler report: Loop was
partially vectorized

Exec. Time scalar code: 14.7

Exec. Time vector code: 18.1

Speedup: 0.8



Loop Distribution

- It is also called loop fission.
- Divides loop control over different statements in the loop body.

```
for (i=1; i<LEN; i++) {  
    a[i]= (float)sqrt(b[i])+  
          (float)sqrt(c[i]);  
    dummy(a,b,c);  
}
```



Loop Distribution

- It is also called loop fission.
- Divides loop control over different statements in the loop body.

```
for (i=1; i<LEN; i++) {  
    a[i]= (float)sqrt(b[i])+  
          (float)sqrt(c[i]);  
    dummy(a,b,c);  
}  
→  
for (i=1; i<LEN; i++)  
    a[i]= (float)sqrt(b[i])+  
          (float)sqrt(c[i]);  
  
for (i=1; i<LEN; i++)  
    dummy(a,b,c);
```

- Compiler cannot analyze the dummy function.
As a result, the compiler cannot apply loop distribution,
because it does not know if it is a legal transformation
- Programmer can apply loop distribution if legal.



Loop Distribution

S126

```
for (i=1; i<LEN; i++) {  
    a[i]= (float)sqrt(b[i])+  
          (float)sqrt(c[i]);  
    dummy(a,b,c);  
}
```

S126_1

```
for (i=1; i<LEN; i++)  
    a[i]= (float)sqrt(b[i])+  
          (float)sqrt(c[i]);  
for (i=1; i<LEN; i++)  
    dummy(a,b,c);
```

S126

Intel Nehalem

Compiler report: Loop was not vectorized

Exec. Time scalar code: 4.3

Exec. Time vector code: --

Speedup: --

S126_1

Intel Nehalem

Compiler report:

- Loop 1 was vectorized.

- Loop 2 was not vectorized

Exec. Time scalar code: 5.1

Exec. Time vector code: 1.1

Speedup: 4.6



Loop Distribution

S126

```
for (i=1; i<LEN; i++) {  
    a[i]= (float)sqrt(b[i])+  
          (float)sqrt(c[i]);  
    dummy(a,b,c);  
}
```

S126

IBM Power 7

Compiler report: Loop was not
SIMD vectorized

Exec. Time scalar code: 1.3

Exec. Time vector code: --

Speedup: --

S126_1

```
for (i=1; i<LEN; i++)  
    a[i]= (float)sqrt(b[i])+  
          (float)sqrt(c[i]);  
for (i=1; i<LEN; i++)  
    dummy(a,b,c);
```

S126_1

IBM Power 7

Compiler report:

- Loop 1 was SIMD vectorized.
- Loop 2 was not SIMD vectorized

Exec. Time scalar code: 1.14

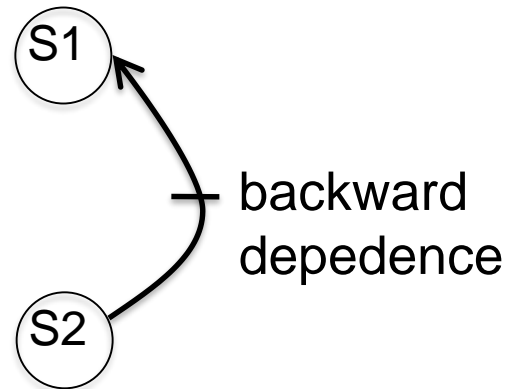
Exec. Time vector code: 1.0

Speedup: 1.14

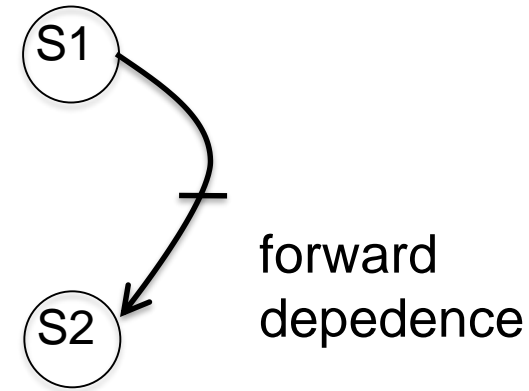


Reordering Statements

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



Reordering Statements

S114

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

S114_1

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

S114

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 12.6

Exec. Time vector code: --

Speedup: --

S114_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 10.7

Exec. Time vector code: 6.2

Speedup: 1.7



Reordering Statements

S114

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

S114_1

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

The IBM XLC compiler generated the same code in both cases

S114

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 3.3

Exec. Time vector code: 1.8

Speedup: 1.8

S114_1

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 3.3

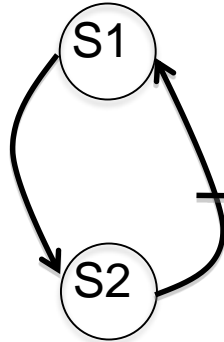
Exec. Time vector code: 1.8

Speedup: 1.8

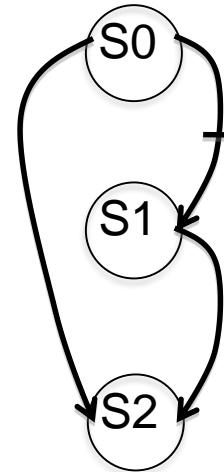


Node Splitting

```
for (int i=0;i<LEN-1;i++){  
  S1 a[i]=b[i]+c[i];  
  S2 d[i]=(a[i]+a[i+1])*(float)0.5;  
}
```



```
for (int i=0;i<LEN-1;i++){  
  S0 temp[i]=a[i+1];  
  S1 a[i]=b[i]+c[i];  
  S2 d[i]=(a[i]+temp[i])*(float) 0.5;  
}
```



Node Splitting

S126

```
for (int i=0;i<LEN-1;i++){  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+a[i+1])*(float)0.5;  
}
```

S126_1

```
for (int i=0;i<LEN-1;i++){  
    temp[i]=a[i+1];  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+temp[i])*(float)0.5;  
}
```

S126

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 12.6

Exec. Time vector code: --

Speedup: --

S126_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 13.2

Exec. Time vector code: 9.7

Speedup: 1.3



Node Splitting

S126

```
for (int i=0;i<LEN-1;i++){  
S1  a[i]=b[i]+c[i];  
S2  d[i]=(a[i]+a[i+1])*(float)0.5;  
}
```

S126_1

```
for (int i=0;i<LEN-1;i++){  
S0 temp[i]=a[i+1];  
S1 a[i]=b[i]+c[i];  
S2 d[i]=(a[i]+temp[i])*(float) 0.5;  
}
```

S126

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 3.8

Exec. Time vector code: 1.7

Speedup: 2.2

S126_1

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 5.1

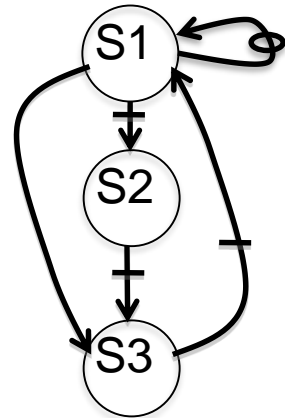
Exec. Time vector code: 2.4

Speedup: 2.0

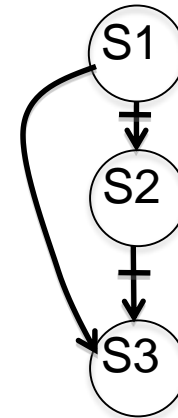


Scalar Expansion

```
for (int i=0;i<n;i++){  
  S1  t = a[i];  
  S2  a[i] = b[i];  
  S3  b[i] = t;  
}
```



```
for (int i=0;i<n;i++){  
  S1  t[i] = a[i];  
  S2  a[i] = b[i];  
  S3  b[i] = t[i];  
}
```



Scalar Expansion

S139

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

S139_1

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

S139

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 0.7

Exec. Time vector code: 0.4

Speedup: 1.5

S139_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 0.7

Exec. Time vector code: 0.4

Speedup: 1.5



Scalar Expansion

S139

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

S139

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 0.28

Exec. Time vector code: 0.14

Speedup: 2

S139_1

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

S139_1

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 0.28

Exec. Time vector code: 0.14

Speedup: 2.0



Loop Peeling

- Remove the first/s or the last/s iteration of the loop into separate code outside the loop
- It is always legal, provided that no additional iterations are introduced.
- When the trip count of the loop is not constant the peeled loop has to be protected with additional runtime tests.
- This transformation is useful to enforce a particular initial memory alignment on array references prior to loop vectorization.

```
for (i=0; i<LEN; i++)  
    A[i] = B[i] + C[i];
```

→

```
A[0] = B[0] + C[0];  
for (i=1; i<LEN; i++)  
    A[i] = B[i] + C[i];
```



Loop Peeling

- Remove the first/s or the last/s iteration of the loop into separate code outside the loop
- It is always legal, provided that no additional iterations are introduced.
- When the trip count of the loop is not constant the peeled loop has to be protected with additional runtime tests.
- This transformation is useful to enforce a particular initial memory alignment on array references prior to loop vectorization.

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

→

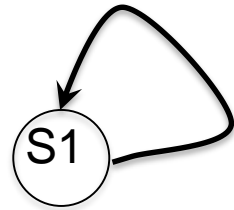
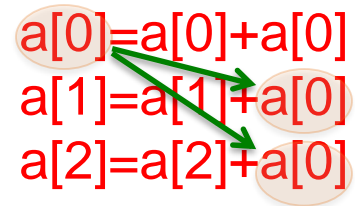
```
if (N>=1)  
    A[0] = B[0] + C[0];  
for (i=1; i<N; i++)  
    A[i] = B[i] + C[i];
```



Loop Peeling

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

$a[0] = a[0] + a[0]$
 $a[1] = a[1] + a[0]$
 $a[2] = a[2] + a[0]$



Self true-dependence
is not vectorized

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

After loop peeling, there are no
dependences, and the loop can be
vectorized



Loop Peeling

S127

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

S127_1

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

S127

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 6.7

Exec. Time vector code: --

Speedup: --

S127_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 6.6

Exec. Time vector code: 1.2

Speedup: 5.2



Loop Interchanging

- This transformation switches the positions of one loop that is tightly nested within another loop.

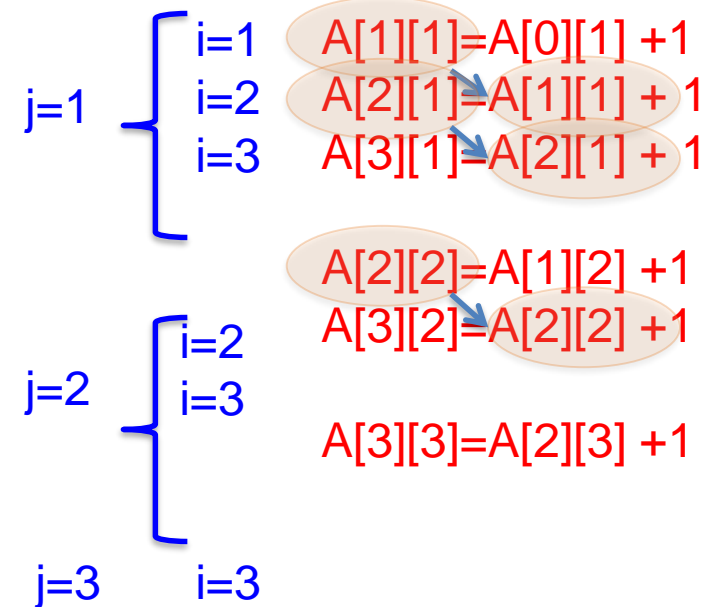
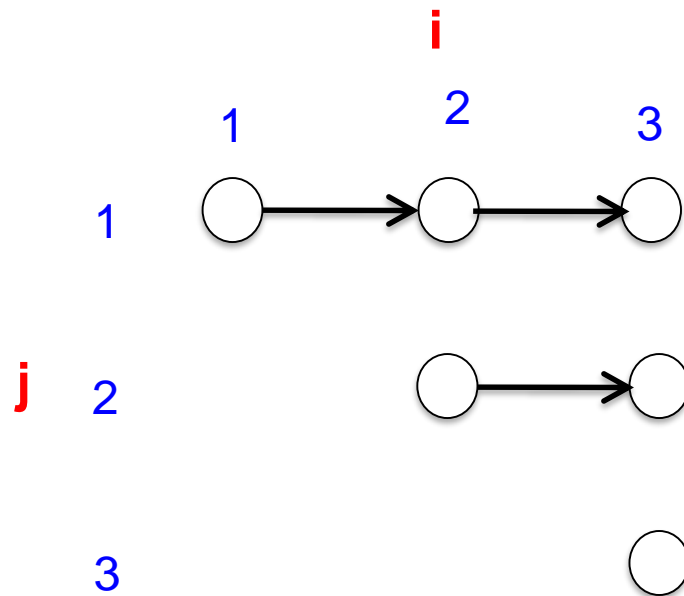
```
for (i=0; i<LEN; i++)  
    for (j=0; j<LEN; j++)  
        A[i][j]=0.0;
```

```
for (j=0; j<LEN; j++)  
    for (i=0; i<LEN; i++)  
        A[i][j]=0.0;
```



Loop Interchanging

```
for (j=1; j<LEN; j++){  
  for (i=j; i<LEN; i++){  
    A[i][j]=A[i-1][j]+(float) 1.0;  
  }  
}
```



$A[1][1] = A[0][1] + 1$
 $A[2][1] = A[1][1] + 1$
 $A[3][1] = A[2][1] + 1$

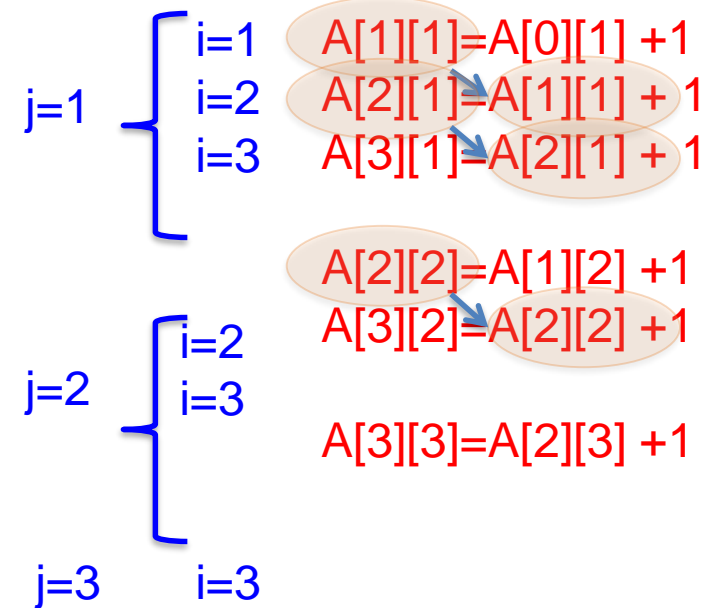
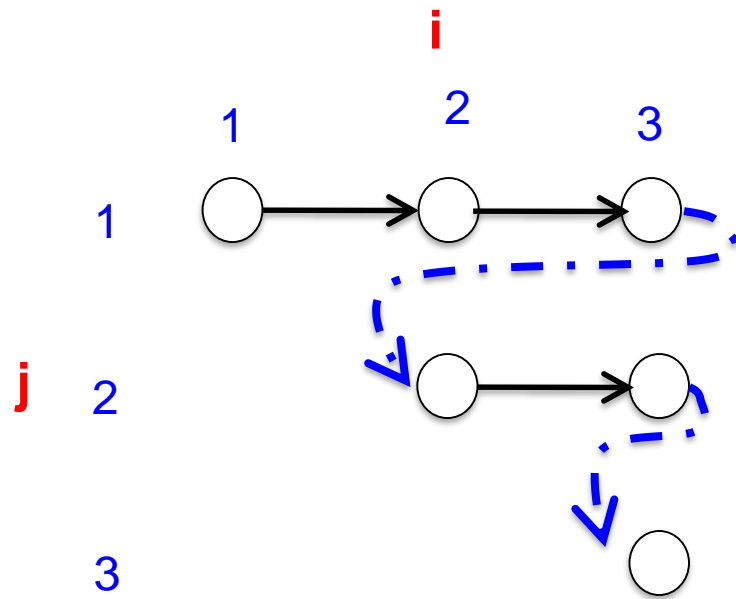
$A[2][2] = A[1][2] + 1$
 $A[3][2] = A[2][2] + 1$

$A[3][3] = A[2][3] + 1$



Loop Interchanging

```
for (j=1; j<LEN; j++){  
  for (i=j; i<LEN; i++){  
    A[i][j]=A[i-1][j]+(float) 1.0;  
  }  
}
```

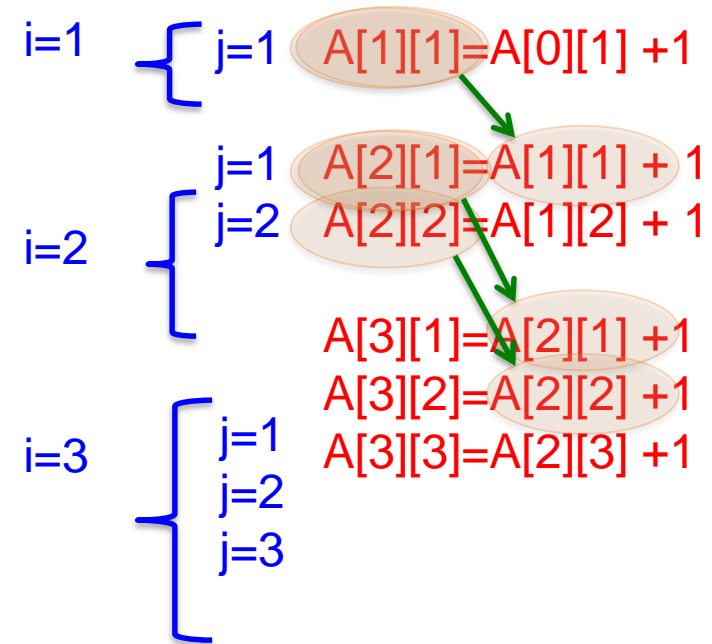
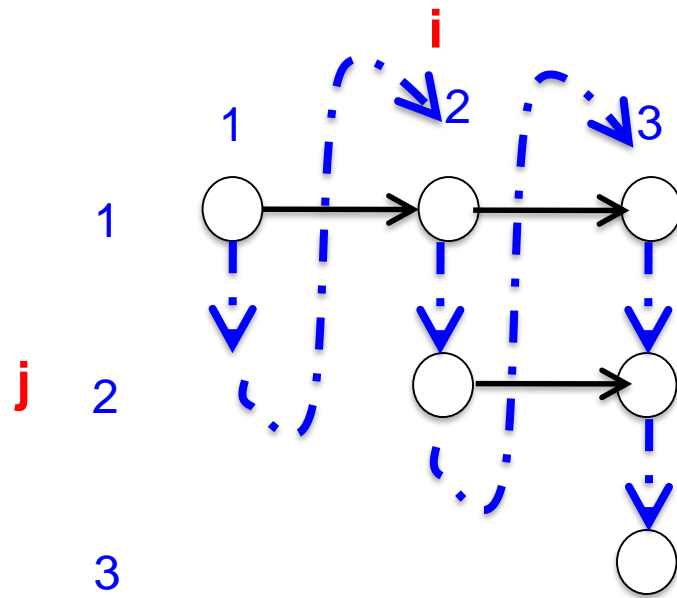


Inner loop cannot be vectorized
because of self-dependence



Loop Interchanging

```
for (i=1; i<LEN; i++){  
  for (j=1; j<i+1; j++){  
    A[i][j]=A[i-1][j]+(float) 1.0;  
  }  
}
```



Loop interchange is legal
No dependences in inner loop



Loop Interchanging

S228

```
for (j=1; j<LEN; j++){  
  for (i=j; i<LEN; i++){  
    A[i][j]=A[i-1][j]+(float)1.0;  
  }}  
}}
```

S228_1

```
for (i=1; i<LEN; i++){  
  for (j=1; j<i+1; j++){  
    A[i][j]=A[i-1][j]+(float)1.0;  
  }}  
}}
```

S228

Intel Nehalem

Compiler report: Loop was not vectorized.

Exec. Time scalar code: 2.3

Exec. Time vector code: --

Speedup: --

S228_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 0.6

Exec. Time vector code: 0.2

Speedup: 3



Loop Interchanging

S228

```
for (j=1; j<LEN; j++){  
  for (i=j; i<LEN; i++){  
    A[i][j]=A[i-1][j]+(float)1.0;  
  }}  
}}
```

S228_1

```
for (i=1; i<LEN; i++){  
  for (j=1; j<i+1; j++){  
    A[i][j]=A[i-1][j]+(float)1.0;  
  }}  
}}
```

S228

IBM Power 7

Compiler report: Loop was not
SIMD vectorized

Exec. Time scalar code: 0.5

Exec. Time vector code: --

Speedup: --

S228_1

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 0.2

Exec. Time vector code: 0.14

Speedup: 1.42



Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Reductions
 - Data Alignment
 - Aliasing
 - Non-unit strides
 - Conditional Statements
4. Vectorization using intrinsics

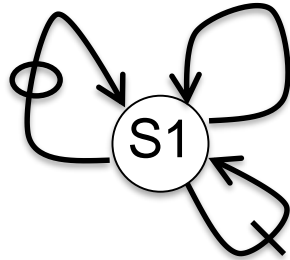


Reductions

- Reduction is an operation, such as addition, which is applied to the elements of an array to produce a result of a lesser rank.

Sum Reduction

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



Max Loc Reduction

```
x = a[0];  
index = 0;  
for (int i=0;i<LEN;++i){  
    if (a[i] > x) {  
        x = a[i];  
        index = i;  
    }  
}}
```



Reductions

S131

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

S132

```
x = a[0];
index = 0;
for (int i=0;i<LEN;++i){
    if (a[i] > x) {
        x = a[i];
        index = i;
    }
}
```

S131

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 5.2

Exec. Time vector code: 1.2

Speedup: 4.1

S132

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 9.6

Exec. Time vector code: 2.4

Speedup: 3.9



Reductions

S131

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

S131

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 1.1

Exec. Time vector code: 0.4

Speedup: 2.4

S132

```
x = a[0];
index = 0;
for (int i=0;i<LEN;++i){
    if (a[i] > x) {
        x = a[i];
        index = i;
    }
}
```

S132

IBM Power 7

Compiler report: Loop was not
SIMD vectorized

Exec. Time scalar code: 4.4

Exec. Time vector code: --

Speedup: --



Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Induction variables
 - Data Alignment
 - Aliasing
 - Non-unit strides
 - Conditional Statements
4. Vectorization with intrinsics



Induction variables

- Induction variable is a variable that can be expressed as a function of the loop iteration variable

```
float s = (float)0.0;  
for (int i=0;i<LEN;i++){  
    s += (float)2.;  
    a[i] = s * b[i];  
}
```

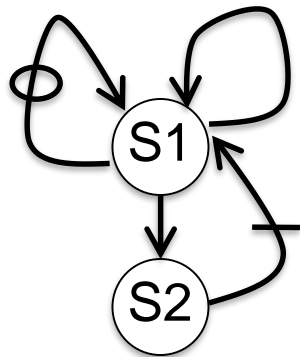


Induction variables

- Induction variable is a variable that can be expressed as a function of the loop iteration variable

```
float s = (float)0.0;  
for (int i=0;i<LEN;i++){  
    s += (float)2.;  
    a[i] = s * b[i];  
}
```

```
for (int i=0;i<LEN;i++){  
    a[i] = (float)2.*(i+1)*b[i];  
}
```



Induction variables

S133

```
float s = (float)0.0;
for (int i=0;i<LEN;i++){
    s += (float)2.;
    a[i] = s * b[i];
}
```

S133_1

```
for (int i=0;i<LEN;i++){
    a[i] = (float)2.*(i+1)*b[i];
}
```

The Intel ICC compiler generated the same vector code in both cases

S133

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 6.1

Exec. Time vector code: 1.9

Speedup: 3.1

S133_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 8.4

Exec. Time vector code: 1.9

Speedup: 4.2



Induction variables

S133

```
float s = (float)0.0;
for (int i=0;i<LEN;i++){
    s += (float)2.;
    a[i] = s * b[i];
}
```

S133

IBM Power 7

Compiler report: Loop was not SIMD vectorized

Exec. Time scalar code: 2.7

Exec. Time vector code: --

Speedup: --

S133_1

```
for (int i=0;i<LEN;i++){
    a[i] = (float)2.*(i+1)*b[i];
}
```

S133_1

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 3.7

Exec. Time vector code: 1.4

Speedup: 2.6



Induction Variables

- Coding style matters:

```
for (int i=0;i<LEN;i++) {  
    *a = *b + *c;  
    a++; b++; c++;  
}
```

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

These codes are equivalent, but ...



Induction Variables

S134

```
for (int i=0;i<LEN;i++) {  
    *a = *b + *c;  
    a++; b++; c++;  
}
```

S134_1

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

S134

Intel Nehalem

Compiler report: Loop was not vectorized.

Exec. Time scalar code: 5.5

Exec. Time vector code: --

Speedup: --

S134_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 6.1

Exec. Time vector code: 3.2

Speedup: 1.8



Induction Variables

S134

```
for (int i=0;i<LEN;i++) {  
    *a = *b + *c;  
    a++; b++; c++;  
}
```

S134_1

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

The IBM XLC compiler generated the same code in both cases

S134

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 2.2

Exec. Time vector code: 1.0

Speedup: 2.2

S134_1

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 2.2

Exec. Time vector code: 1.0

Speedup: 2.2



Outline

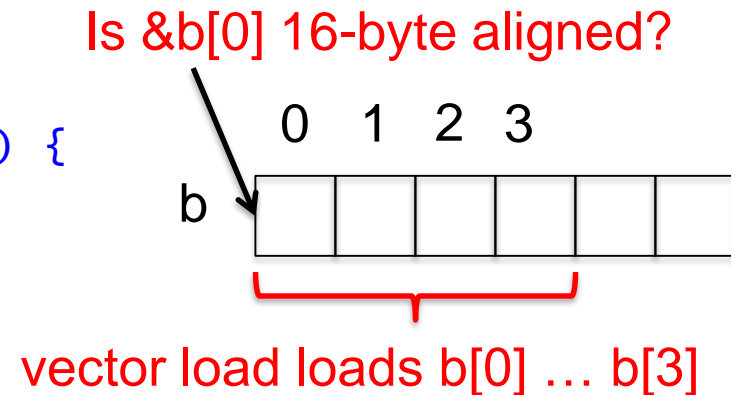
1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - **Data Alignment**
 - Aliasing
 - Non-unit strides
 - Conditional Statements
4. Vectorization with intrinsics



Data Alignment

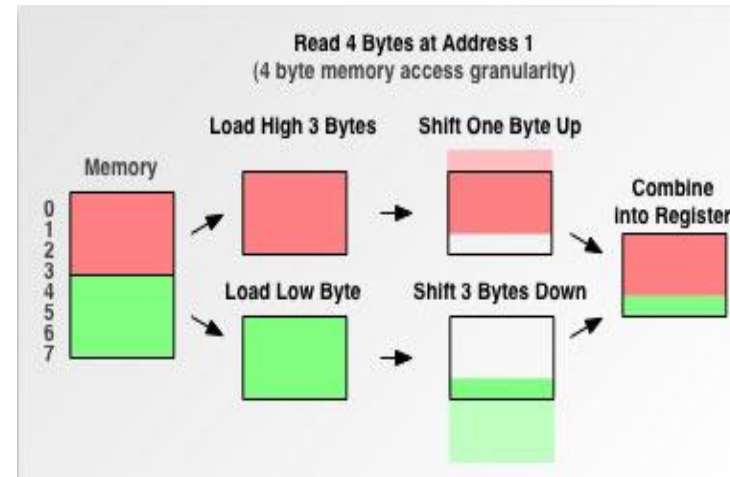
- Vector loads/stores load/store 128 consecutive bits to a vector register.
- Data addresses need to be 16-byte (128 bits) aligned to be loaded/stored
 - Intel platforms support aligned and unaligned load/stores
 - IBM platforms do not support unaligned load/stores

```
void test1(float *a, float *b, float *c) {  
    for (int i=0; i<LEN; i++){  
        a[i] = b[i] + c[i];  
    }  
}
```



Why data alignment may improve efficiency

- Vector load/store from aligned data requires one memory access
- Vector load/store from unaligned data requires multiple memory accesses and some shift operations



Reading 4 bytes from address 1
requires two loads



Data Alignment

- To know if a pointer is 16-byte aligned, the last digit of the pointer address in hex must be 0.
- Note that if &b[0] is 16-byte aligned, and is a single precision array, then &b[4] is also 16-byte aligned

```
__attribute__((aligned(16))) float B[1024];
```

```
int main(){  
    printf("%p, %p\n", &B[0], &B[4]);  
}
```

Output:

0x7fff1e9d8580, 0x7fff1e9d8590



Data Alignment

- In many cases, the compiler cannot statically know the alignment of the address in a pointer
- The compiler assumes that the base address of the pointer is 16-byte aligned and adds a run-time checks for it
 - if the runtime check is false, then it uses another code (which may be scalar)



Data Alignment

- Manual 16-byte alignment can be achieved by forcing the base address to be a multiple of 16.

```
__attribute__((aligned(16))) float b[N];  
float* a = (float*) memalign(16, N*sizeof(float));
```

- When the pointer is passed to a function, the compiler should be aware of where the 16-byte aligned address of the array starts.

```
void func1(float *a, float *b,  
float *c) {  
    __assume_aligned(a, 16);  
    __assume_aligned(b, 16);  
    __assume_aligned(c, 16);  
    for (int i=0; i<LEN; i++) {  
        a[i] = b[i] + c[i];  
    }  
}
```



Data Alignment - Example

```
float A[N] __attribute__((aligned(16)));  
float B[N] __attribute__((aligned(16)));  
float C[N] __attribute__((aligned(16)));  
  
void test(){  
    for (int i = 0; i < N; i++){  
        C[i] = A[i] + B[i];  
    }  
}
```



Data Alignment - Example

```
float A[N] __attribute__((aligned(16)));  
float B[N] __attribute__((aligned(16)));  
float C[N] __attribute__((aligned(16)));
```

```
void test1(){  
    __m128 rA, rB, rC;  
    for (int i = 0; i < N; 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);  
    }  
}
```

```
void test3(){  
    __m128 rA, rB, rC;  
    for (int i = 1; i < N-3; i+=4){  
        rA = _mm_loadu_ps(&A[i]);  
        rB = _mm_loadu_ps(&B[i]);  
        rC = _mm_add_ps(rA,rB);  
        _mm_storeu_ps(&C[i], rC);  
    }  
}
```

```
void test2(){  
    __m128 rA, rB, rC;  
    for (int i = 0; i < N; i+=4){  
        rA = _mm_loadu_ps(&A[i]);  
        rB = _mm_loadu_ps(&B[i]);  
        rC = _mm_add_ps(rA,rB);  
        _mm_storeu_ps(&C[i], rC);  
    }  
}
```

Nanosecond per iteration			
	Core 2 Duo	Intel i7	Power 7
Aligned	0.577	0.580	0.156
Aligned (unaligned ld)	0.689	0.581	0.241
Unaligned	2.176	0.629	0.243



Alignment in a struct

```
struct st{
    char A;
    int B[64];
    float C;
    int D[64];
};

int main(){
    st s1;
    printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);}
```

Output:

0x7ffe6765f00, 0x7ffe6765f04, 0x7ffe6766004, 0x7ffe6766008

- Arrays B and D are not 16-bytes aligned (see the address)



Alignment in a struct

```
struct st{
    char A;
    int B[64] __attribute__((aligned(16)));
    float C;
    int D[64] __attribute__((aligned(16)));
};

int main(){
    st s1;
    printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);}
```

Output:

0x7fff1e9d8580, 0x7fff1e9d8590, 0x7fff1e9d8690, 0x7fff1e9d86a0

- Arrays A and B are aligned to 16-bytes (notice the 0 in the 4 least significant bits of the address)
- Compiler automatically does padding



Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Data Alignment
 - **Aliasing**
 - Non-unit strides
 - Conditional Statements
4. Vectorization with intrinsics



Aliasing

- Can the compiler vectorize this loop?

```
void func1(float *a, float *b, float *c){  
    for (int i = 0; i < LEN; i++) {  
        a[i] = b[i] + c[i];  
    }  
}
```



Aliasing

- Can the compiler vectorize this loop?

```
float* a = &b[1];
```

```
...
```

```
void func1(float *a, float *b, float *c)
{
    for (int i = 0; i < LEN; i++)
        a[i] = b[i] + c[i];
}
```

$b[1] = b[0] + c[0]$

$b[2] = b[1] + c[1]$



Aliasing

- Can the compiler vectorize this loop?

```
float* a = &b[1];
```

```
...
```

```
void func1(float *a, float *b, float *c)
{
    for (int i = 0; i < LEN; i++)
        a[i] = b[i] + c[i];
}
```

a and b are aliasing

There is a self-true dependence

Vectorizing this loop would
be illegal



Aliasing

- To vectorize, the compiler needs to guarantee that the pointers are not aliased.
- When the compiler does not know if two pointer are alias, it still vectorizes, but needs to add up-to $O(n^2)$ run-time checks, where n is the number of pointers

When the number of pointers is large, the compiler may decide to not vectorize

```
void func1(float *a, float *b, float *c){  
    for (int i=0; i<LEN; i++)  
        a[i] = b[i] + c[i];  
}
```



Aliasing

- Two solutions can be used to avoid the run-time checks
 1. static and global arrays
 2. `__restrict__` attribute



Aliasing

1. Static and Global arrays

```
__attribute__((aligned(16))) float a[LEN];  
__attribute__((aligned(16))) float b[LEN];  
__attribute__((aligned(16))) float c[LEN];
```

```
void func1(){  
    for (int i=0; i<LEN; i++)  
        a[i] = b[i] + c[i];  
}
```

```
int main() {  
    ...  
    func1();  
}
```



Aliasing

1. __restrict__ keyword

```
void func1(float* __restrict__ a, float* __restrict__ b,  
float* __restrict__ c) {  
    __assume_aligned(a, 16);  
    __assume_aligned(b, 16);  
    __assume_aligned(c, 16);  
    for int (i=0; i<LEN; i++)  
        a[i] = b[i] + c[i];  
}  
int main() {  
    float* a=(float*) memalign(16,LEN*sizeof(float));  
    float* b=(float*) memalign(16,LEN*sizeof(float));  
    float* c=(float*) memalign(16,LEN*sizeof(float));  
    ...  
    func1(a,b,c);  
}
```



Aliasing – Multidimensional arrays

- Example with 2D arrays: pointer-to-pointer declaration.

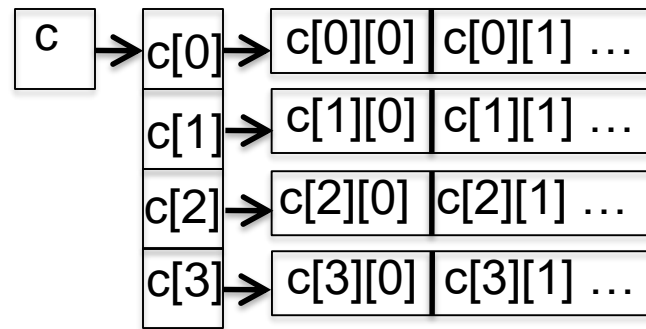
```
void func1(float** __restrict__ a, float**  
__restrict__ b, float** __restrict__ c) {  
    for (int i=0; i<LEN; i++)  
        for (int j=1; j<LEN; j++)  
            a[i][j] = b[i][j-1] * c[i][j];  
}
```



Aliasing – Multidimensional arrays

- Example with 2D arrays: pointer-to-pointer declaration.

```
void func1(float** __restrict__ a, float** __restrict__  
b, float** __restrict__ c) {  
    for (int i=0; i<LEN; i++)  
        for (int j=1; j<LEN; j++)  
            a[i][j] = b[i][j-1] * c[i][j];  
}
```



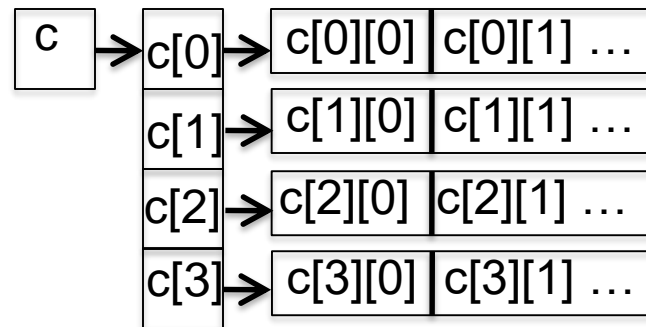
`__restrict__` only qualifies
the first dereferencing of `c`;

Nothing is said about the
arrays that can be accessed
through `c[i]`

Aliasing – Multidimensional arrays

- Example with 2D arrays: pointer-to-pointer declaration.

```
void func1(float** __restrict__ a, float** __restrict__  
b, float** __restrict__ c) {  
    for (int i=0; i<LEN; i++)  
        for (int j=1; j<LEN; j++)  
            a[i][j] = b[i][j-1] * c[i][j];  
}
```



`__restrict__` only qualifies
the first dereferencing of `c`;

Nothing is said about the
arrays that can be accessed
through `c[i]`

Intel ICC compiler, version 11.1 will vectorize this code.

Previous versions of the Intel compiler or compilers from
other vendors, such as IBM XLC, will not vectorize it.



Aliasing – Multidimensional Arrays

- Three solutions when `__restrict__` does not enable vectorization
 1. Static and global arrays
 2. Linearize the arrays and use `__restrict__` keyword
 3. Use compiler directives



Aliasing – Multidimensional arrays

1. Static and Global declaration

```
__attribute__((aligned(16))) float a[N][N];  
void t(){  
    a[i][j]...  
}  
  
int main() {  
    ...  
    t();  
}
```



Aliasing – Multidimensional arrays

2. Linearize the arrays

```
void t(float* __restrict__ A){  
    //Access to Element A[i][j] is now A[i*128+j]  
    ...  
}  
  
int main() {  
    float* A = (float*) memalign(16,128*128*sizeof(float));  
    ...  
    t(A);  
}
```



Aliasing – Multidimensional arrays

3. Use compiler directives:

```
#pragma ivdep (Intel ICC)  
#pragma disjoint(IBM XLC)
```

```
void func1(float **a, float **b, float **c) {  
    for (int i=0; i<m; i++) {  
        #pragma ivdep  
        for (int j=0; j<LEN; j++)  
            c[i][j] = b[i][j] * a[i][j];  
    }  
}
```



Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Data Alignment
 - Aliasing
 - Non-unit strides
 - Conditional Statements
4. Vectorization with intrinsics

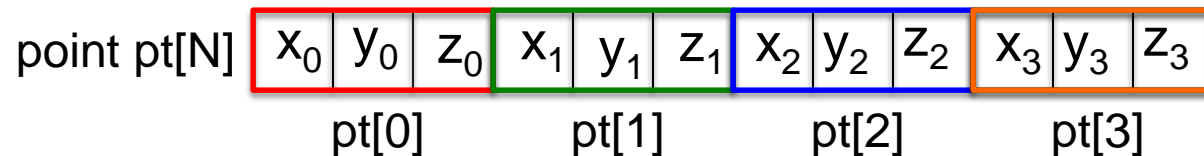


Non-unit Stride – Example I

- Array of a struct

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];
```

```
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

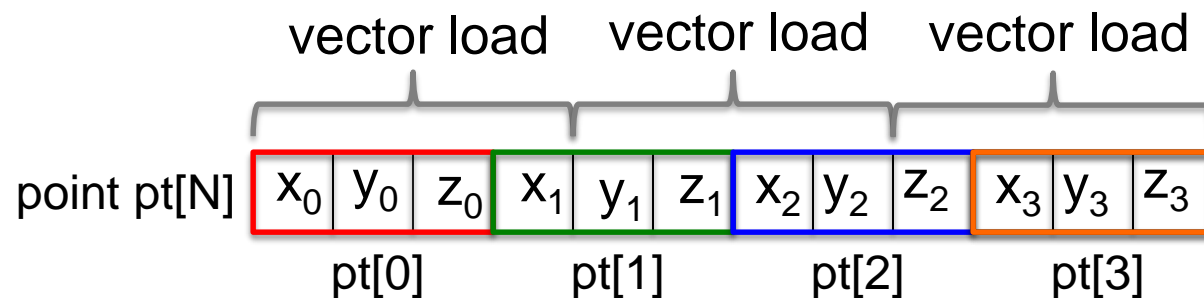


Non-unit Stride – Example I

- Array of a struct

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];
```

```
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

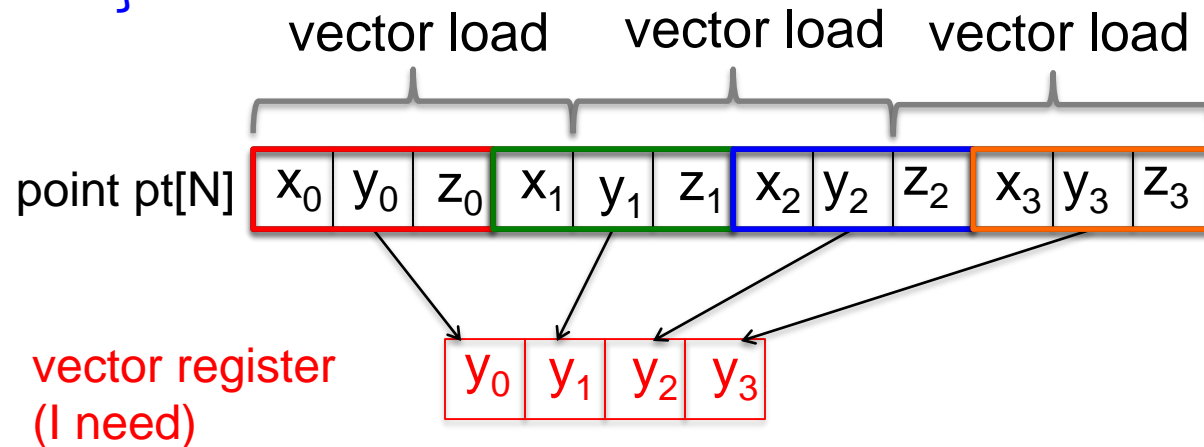


Non-unit Stride – Example I

- Array of a struct

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];
```

```
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

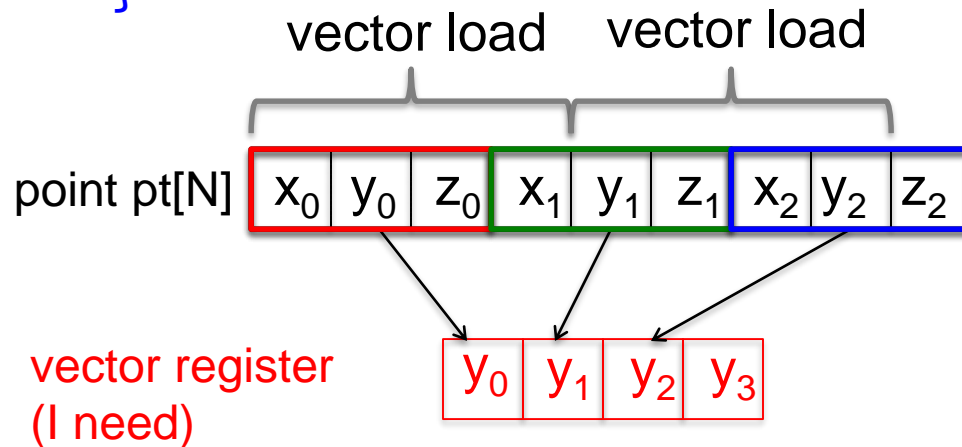


Non-unit Stride – Example I

- Array of a struct

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];
```

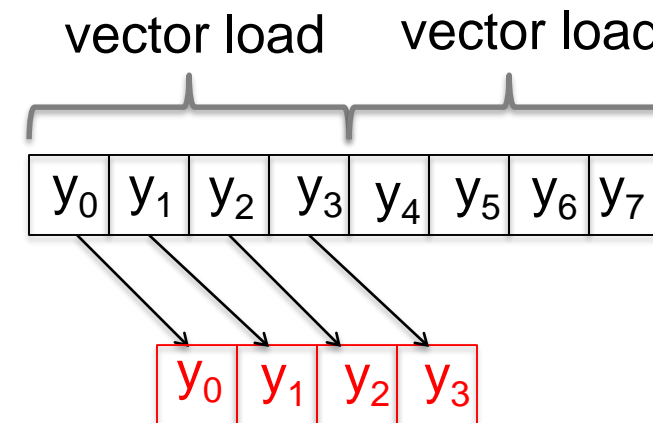
```
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```



- Arrays

```
int ptx[LEN], int pty[LEN],  
int ptz[LEN];
```

```
for (int i=0; i<LEN; i++) {  
    pty[i] *= scale;  
}
```



Non-unit Stride – Example I

S135

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

S135

Intel Nehalem

Compiler report: Loop was not vectorized. Vectorization possible but seems inefficient

Exec. Time scalar code: 6.8

Exec. Time vector code: --

Speedup: --

S135_1

```
int ptx[LEN], int pty[LEN],  
int ptz[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pty[i] *= scale;  
}
```

S135_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 4.8

Exec. Time vector code: 1.3

Speedup: 3.7



Non-unit Stride – Example I

S135

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

S135

IBM Power 7

Compiler report: Loop was not SIMD vectorized because it is not profitable to vectorize

Exec. Time scalar code: 2.0

Exec. Time vector code: --

Speedup: --

S135_1

```
int ptx[LEN], int pty[LEN],  
int ptz[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pty[i] *= scale;  
}
```

S135_1

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 1.8

Exec. Time vector code: 1.5

Speedup: 1.2



Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Data Alignment
 - Aliasing
 - Non-unit strides
 - Conditional Statements
4. Vectorization with intrinsics



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
<code>#pragma ivdep</code>	Ignore assume data dependences
<code>#pragma vector always</code>	override efficiency heuristics
<code>#pragma novector</code>	disable vectorization
<code>__restrict__</code>	assert exclusive access through pointer
<code>__attribute__((aligned(int-val)))</code>	request memory alignment
<code>memalign(int-val,size);</code>	malloc aligned memory
<code>__assume_aligned(exp, int-val)</code>	assert alignment property



Compiler Directives

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

Compiler Hints for IBM XLC	Semantics
<code>#pragma ibm independent_loop</code>	Ignore assumed data dependences
<code>#pragma nosimd</code>	disable vectorization
<code>__restrict__</code>	assert exclusive access through pointer
<code>__attribute__((aligned(int-val)))</code>	request memory alignment
<code>memalign(int-val,size);</code>	malloc aligned memory
<code>__alignx (int-val, exp)</code>	assert alignment property



Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Data Alignment
 - Aliasing
 - Non-unit strides
 - Conditional Statements
4. Vectorization with intrinsics



Access the SIMD through intrinsics

- Intrinsics are vendor/architecture specific
- We will focus on the Intel vector intrinsics
- Intrinsics are useful when
 - the compiler fails to vectorize
 - when the programmer thinks it is possible to generate better code than the one produced by the compiler



The Intel SSE intrinsics Header file

- SSE can be accessed using intrinsics.
- You must use one of the following header files:
 - `#include <xmmintrin.h>` (for SSE)
 - `#include <emmintrin.h>` (for SSE2)
 - `#include <pmmmintrin.h>` (for SSE3)
 - `#include <smmintrin.h>` (for SSE4)
- These include the prototypes of the intrinsics.



Intel SSE intrinsics Data types

- We will use the following data types:
 - `__m128` packed single precision (vector XMM register)
 - `__m128d` packed double precision (vector XMM register)
 - `__m128i` packed integer (vector XMM register)
- Example

```
#include <xmmintrin.h>
int main ( ) {
    ...
    __m128 A, B, C; /* three packed s.p. variables */
    ...
}
```



Intel SSE intrinsic Instructions

- Intrinsics operate on these types and have the format:

`_mm_instruction_suffix(...)`

- Suffix can take many forms. Among them:

`ss` scalar single precision

`ps` packed (vector) single precision

`sd` scalar double precision

`pd` packed double precision

`si#` scalar integer (8, 16, 32, 64, 128 bits)

`su#` scalar unsigned integer (8, 16, 32, 64, 128 bits)



Intel SSE intrinsics

Instructions – Examples

- Load four 16-byte aligned single precision values in a vector:

```
float a[4]={1.0,2.0,3.0,4.0}; //a must be 16-byte aligned  
__m128 x = _mm_load_ps(a);
```

- Add two vectors containing four single precision values:

```
__m128 a, b;  
__m128 c = _mm_add_ps(a, b);
```



Intrinsics (SSE)

```
#define n 1024
__attribute__((aligned(16)))
float a[n], b[n], c[n];
```

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

```
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float
a[n], b[n], c[n];
```

```
int main() {
    __m128 rA, rB, rC;
    for (i = 0; i < n; i+=4) {
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC= _mm_mul_ps(rA,rB);
        _mm_store_ps(&c[i], rC);
    }
}
```



Intel SSE intrinsics

A complete example

```
#define n 1024
```

```
int main() {  
    float a[n], b[n], c[n];  
    for (i = 0; i < n; i+=4) {  
        c[i:i+3]=a[i:i+3]+b[i:i+3];  
    }  
}
```

Header file

```
#include <xmmintrin.h>
```

```
#define n 1024  
__attribute__((aligned(16))) float  
a[n], b[n], c[n];
```

```
int main() {  
    __m128 rA, rB, rC;  
    for (i = 0; i < n; i+=4) {  
        rA = _mm_load_ps(&a[i]);  
        rB = _mm_load_ps(&b[i]);  
        rC = _mm_mul_ps(rA, rB);  
        _mm_store_ps(&c[i], rC);  
    }  
}
```



Intel SSE intrinsics

A complete example

```
#define n 1024
```

```
int main() {  
    float a[n], b[n], c[n];  
    for (i = 0; i < n; i+=4) {  
        c[i:i+3]=a[i:i+3]  
    }  
}
```

Declare 3 vector registers

```
#include <xmmintrin.h>
```

```
#define n 1024
```

```
__attribute__((aligned(16))) float  
a[n], b[n], c[n];
```

```
int main() {  
    __m128 rA, rB, rC;  
    for (i = 0; i < n; i+=4) {  
        rA = _mm_load_ps(&a[i]);  
        rB = _mm_load_ps(&b[i]);  
        rC= _mm_mul_ps(rA,rB);  
        _mm_store_ps(&c[i], rC);  
    }  
}
```



Intel SSE intrinsics

A complete example

```
#define n 1000
```

```
int main() {  
    float a[n], b[n], c[n];  
    for (i = 0; i < n; i+=4) {  
        c[i:i+3]=a[i:i+3]+b[i:i+3];  
    }  
}
```



Execute vector
statements

```
#include <xmmintrin.h>
```

```
#define n 1024
```

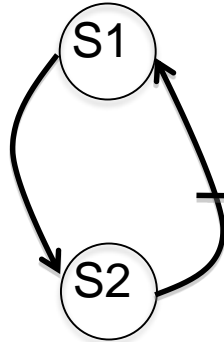
```
__attribute__((aligned(16))) float  
a[n], b[n], c[n];
```

```
int main() {  
    __m128 rA, rB, rC;  
    for (i = 0; i < n; i+=4) {  
        rA = _mm_load_ps(&a[i]);  
        rB = _mm_load_ps(&b[i]);  
        rC= _mm_mul_ps(rA,rB);  
        _mm_store_ps(&c[i], rC);  
    }  
}
```

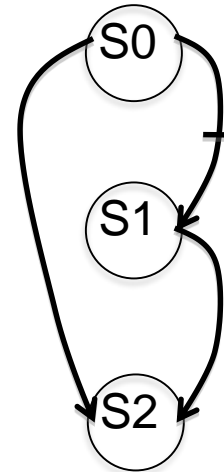


Node Splitting

```
for (int i=0;i<LEN-1;i++){  
  S1 a[i]=b[i]+c[i];  
  S2 d[i]=(a[i]+a[i+1])*(float)0.5;  
}
```



```
for (int i=0;i<LEN-1;i++){  
  S0 temp[i]=a[i+1];  
  S1 a[i]=b[i]+c[i];  
  S2 d[i]=(a[i]+temp[i])*(float) 0.5;  
}
```



Node Splitting with intrinsics

```
for (int i=0;i<LEN-1;i++){  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+a[i+1])*(float)0.5;  
}
```

```
for (int i=0;i<LEN-1;i++){  
    temp[i]=a[i+1];  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+temp[i])*(float)0.5;  
}
```

Which code runs faster ?

Why?

```
#include <xmmintrin.h>  
#define n 1000  
  
int main() {  
    __m128 rA1, rA2, rB, rC, rD;  
    __m128 r5=_mm_set1_ps((float)0.5)  
    for (i = 0; i < LEN-4; i+=4) {  
        rA2= _mm_loadu_ps(&a[i+1]);  
        rB= _mm_load_ps(&b[i]);  
        rC= _mm_load_ps(&c[i]);  
        rA1= _mm_add_ps(rB, rC);  
        rD= _mm_mul_ps(_mm_add_ps(rA1, rA2), r5);  
        _mm_store_ps(&a[i], rA1);  
        _mm_store_ps(&d[i], rD);  
    }  
}
```



Node Splitting with intrinsics

S126

```
for (int i=0;i<LEN-1;i++){  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+a[i+1])*(float)0.5;  
}
```

S126_1

```
for (int i=0;i<LEN-1;i++){  
    temp[i]=a[i+1];  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+temp[i])*(float)0.5;  
}
```

S126_2

```
#include <xmmintrin.h>  
#define n 1000  
  
int main() {  
    __m128 rA1, rA2, rB, rC, rD;  
    __m128 r5=_mm_set1_ps((float)0.5)  
    for (i = 0; i < LEN-4; i+=4) {  
        rA2= _mm_loadu_ps(&a[i+1]);  
        rB= _mm_load_ps(&b[i]);  
        rC= _mm_load_ps(&c[i]);  
        rA1= _mm_add_ps(rB, rC);  
        rD= _mm_mul_ps(_mm_add_ps(rA1, rA2), r5);  
        _mm_store_ps(&a[i], rA1);  
        _mm_store_ps(&d[i], rD);  
    }  
}
```



Node Splitting with intrinsics

S126

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 12.6

Exec. Time vector code: --

Speedup: --

S126_1

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 13.2

Exec. Time vector code: 9.7

Speedup: 1.3

S126_2

Intel Nehalem

Exec. Time intrinsics: 6.1

Speedup (versus vector code): 1.6



Node Splitting with intrinsics

S126

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 3.8

Exec. Time vector code: 1.7

Speedup: 2.2

S126_1

IBM Power 7

Compiler report: Loop was SIMD vectorized

Exec. Time scalar code: 5.1

Exec. Time vector code: 2.4

Speedup: 2.0

S126_2

IBM Power 7

Exec. Time intrinsics: 1.6

Speedup (versus vector code): 1.5



Summary

- Microprocessor vector extensions can contribute to improve program performance and the amount of this contribution is likely to increase in the future as vector lengths grow.
- Compilers are only partially successful at vectorizing
- When the compiler fails, programmers can
 - add compiler directives
 - apply loop transformations
- If after transforming the code, the compiler still fails to vectorize (or the performance of the generated code is poor), the only option is to program the vector extensions directly using intrinsics or assembly language.



Vectorization with GCC

GCC Autovectorization Compiler Flags

- Active autovectorization: `-O -ftree-vectorize`
- `-O3` optimizations include autovectorization
 - `-O3` turns on all optimizations specified by `-O2`
 - Also turns on `-ftree-loop-vectorize`, `-ftree-loop-distribute-patterns`, `-ftree-slp-vectorize`, and `-fvect-cost-model`
- List vectorization reports
 - `-fopt-info-vec-optimized` – list vectorized loops
 - `-fopt-info-vec-missed` – list loops which were not vectorized
 - `-fopt-info-vec-all` – list all information

GCC Autovectorization Compiler Flags

- Use instructions supported by the local CPU: `-march=native`
- Specify the ISA extension: `-mavx`, `-msse4`
- Programmer asserts no loop-carried dependence
 - `#pragma GCC ivdep`
- Generate the GNU assembly and make sure the compiler is doing what you expect

References

- Michael Voss, Software and Services Group, Intel. Topics in Loop Vectorization.
- María Garzarán, Saeed Maleki, William Gropp and David Padua. Program Optimization Through Loop Vectorization. UIUC.
- Daniel Kusswurm – Modern X86 Assembly Language Programming.