Program Optimization Through Loop Vectorization

María Garzarán, Saeed Maleki
William Gropp and David Padua

Department of Computer Science
University of Illinois at Urbana-Champaign

Program Optimization Through Loop Vectorization

Materials for this tutorial can be found:
http://polaris.cs.uiuc.edu/~garzaran/pldi-polv.zip

Questions?
Send an email to garzaran@uiuc.edu

2

Topics covered in this tutorial

- What are the microprocessor vector extensions or SIMD (Single Instruction Multiple Data Units)
- How to use them
  - Through the compiler via automatic vectorization
    - Manual transformations that enable vectorization
    - Directives to guide the compiler
  - Through intrinsics
- Main focus on vectorizing through the compiler.
  - Code more readable
  - Code portable

3

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

4

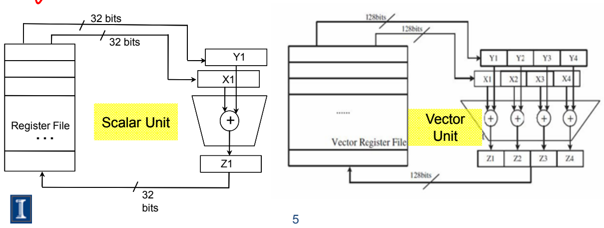
Simple Example

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

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

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

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



SIMD Vectorization

- The use of SIMD units can speed up the program.
- Intel SSE and IBM Altivec have 128-bit vector registers and functional units
  - 4 32-bit single precision floating point numbers
  - 2 64-bit double precision floating point numbers
  - 4 32-bit integer numbers
  - 2 64 bit integer
  - 8 16-bit integer or shorts
  - 16 8-bit bytes or chars
- Assuming a single ALU, these SIMD units can execute 4 single precision floating point number or 2 double precision operations in the time it takes to do only one of these operations by a scalar unit.

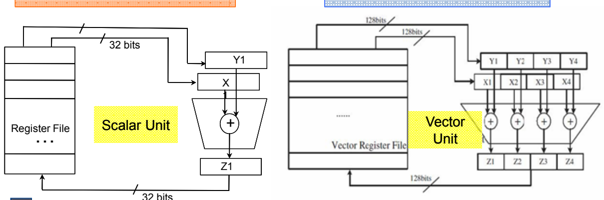
Executing Our Simple Example

S000

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

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

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



How do we access the SIMD units?

- Three choices

1. C code and a vectorizing compiler

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

1. Macros or Vector Intrinsics

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

1. Assembly Language

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

Why should the compiler vectorize?

1. Easier
2. Portable across vendors and machines
   – Although compiler directives differ across compilers
3. Better performance of the compiler generated code
   – Compiler applies other transformations

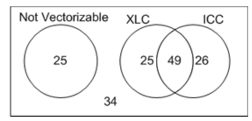
Compilers make your codes (almost) machine independent

But, compilers fail:
- Programmers need to provide the necessary information
- Programmers need to transform the code

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

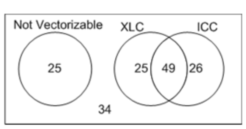
Compiler / Loops	XLC but not ICC	ICC but not XLC
Vectorized	25	26



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)

How much programmer intervention?

- Next, three examples to illustrate what the programmer may need to do:
  – Add compiler directives
  – Transform the code
  – Program using vector intrinsics

Experimental results

- The tutorial shows results for two different platforms with their 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];
  }
}
```

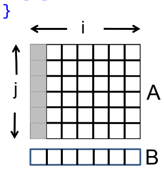
S1111

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;
}
```

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



Loop Transformations

S136

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

S136_1

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

S136_2

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

S136

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

S136_1

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

S136_2

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

Loop Transformations

S136

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

S136_1

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

S136_2

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

S136

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. Time scalar code: 2.0
Exec. Time vector code: --
Speedup: --

S136_1

IBM Power 7
report: Loop interchanging applied. Loop was SIMD vectorized
scalar code: 0.4
vector code: 0.2
Speedup: 2.0

S136_2

IBM Power 7
report: Loop interchanging applied. Loop was SIMD vectorized
scalar code: 0.4
vector code: 0.16
Speedup: 2.7

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);
}}
```

Intrinsics (Altivec)

```
#define n 1024
__attribute__ ((aligned(16))) float a[n],b[n],c[n];
...
for (int i=0; i<LEN; i++)
  c[i]=a[i]*b[i];
```

```
vector float rA,rB,rC,r0;    // Declares vector registers
r0 = vec_xor(r0,r0);         // Sets r0 to {0,0,0,0}
for (int i=0; i<LEN; i+=4){    // Loop stride is 4
  rA = vec_ld(0, &a[i]);     // Load values to rA
  rB = vec_ld(0, &b[i]);     // Load values to rB
  rC = vec_madd(rA,rB,r0);   // rA and rB are multiplied
  vec_st(rC, 0, &c[i]);   // rC is stored to the c[i:i+3]
}
```


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 dependences

- The notion of dependence is the foundation of the process of vectorization.
- It is used to build a calculus of program transformations that can be applied manually by the programmer or automatically by a compiler.

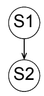
Definition of Dependence

- A statement S is said to be data dependent on statement T if
  – T executes before S in the original sequential/scalar program
  – S and T access the same data item
  – At least one of the accesses is a write.

Data Dependence

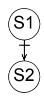
Flow dependence (True dependence)

S1: X = A+B
S2: C= X+A



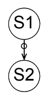
Anti dependence

S1: A = X + B
S2: X= C + D



Output dependence

S1: X = A+B
S2: X= C + D



Data Dependence

- Dependences indicate an execution order that must be honored.
- Executing statements in the order of the dependences guarantee correct results.
- Statements not dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation.

Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement “executions”.

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


Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

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

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

Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

i=0 i=1 i=2

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

S1: a[1] = b[1] + 1
S2: c[1] = a[1] + 2

S1: a[2] = b[2] + 1
S2: c[2] = a[2] + 2

29

Dependences in Loops (I)

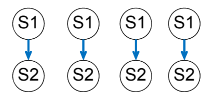
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

iteration: 0 1 2 3 …

instances of S1:

instances of S2:



30

Dependences in Loops (I)

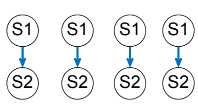
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

iteration: 0 1 2 3 …

instances of S1:

instances of S2:



→ Loop independent dependence

31

Dependences in Loops (I)

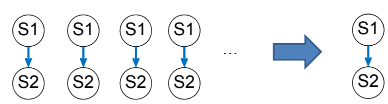
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

iteration: 0 1 2 3 …

instances of S1:

instances of S2:



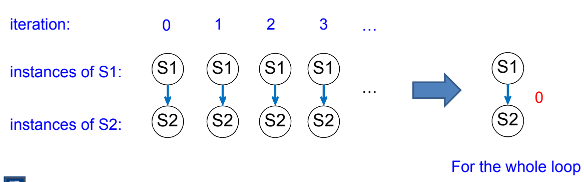
For the whole loop

32

Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

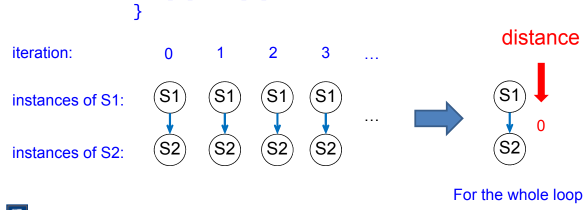
```
     for (i=0; i<n; i++){
S1   a[i] = b[i] + 1;
S2   c[i] = a[i] + 2;
     }
```



Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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



Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

For the dependences shown here, we assume that arrays do not overlap in memory (no aliasing). Compilers must know that there is no aliasing in order to vectorize.

Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

i=1	i=2	i=3
S1: a[1] = b[1] + 1	S1: a[2] = b[2] + 1	S1: a[3] = b[3] + 1
S2: c[1] = a[0] + 2	S2: c[2] = a[1] + 2	S2: c[3] = a[2] + 2

Dependences in Loops (II)

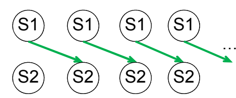
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

iteration: 1 2 3 4 …

instances of S1:

instances of S2:



Dependences in Loops (II)

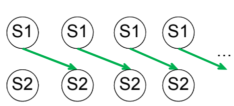
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

iteration: 1 2 3 4 …

instances of S1:

instances of S2:



Loop carried dependence

Dependences in Loops (II)

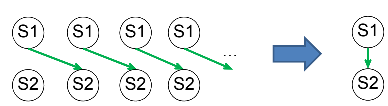
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

iteration: 1 2 3 4 …

instances of S1:

instances of S2:

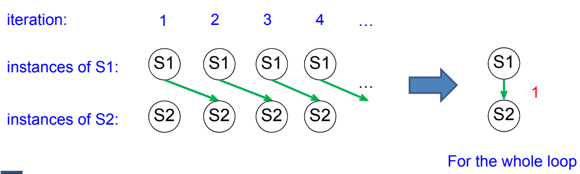


For the whole loop

Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

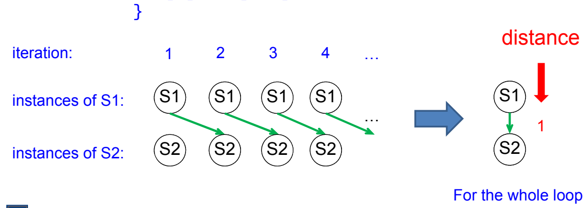
```
     for (i=1; i<n; i++){
S1     a[i] = b[i] + 1;
       c[i] = a[i-1] + 2;
     }
```



Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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



Dependences in Loops (III)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

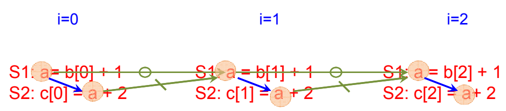

Dependences in Loops (III)

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

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

Dependences in Loops (III)

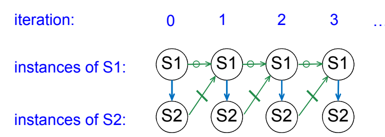
```
     for (i=0; i<n; i++){
S1     a = b[i] + 1;
S2     c[i] = a + 2;
     }
```



Loop independent dependence
Loop carried dependence

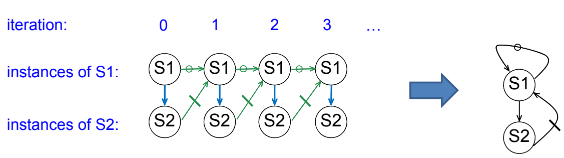
Dependences in Loops (III)

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



Dependences in Loops (III)

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



Dependences in Loops (IV)

- Doubly nested loops

```
 for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
 }}
```

Dependences in Loops (IV)

```
 for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
 }}
```

	i=1	i=2
j=1	a[1][1] = a[1][0] + a[0][1]	a[2][1] = a[2][0] + a[1][1]
j=2	a[1][2] = a[1][1] + a[0][2]	a[2][2] = a[2][1] + a[1][2]
j=3	a[1][3] = a[1][2] + a[0][3]	a[2][3] = a[2][2] + a[1][3]
j=4	a[1][4] = a[1][3] + a[0][4]	a[2][4] = a[2][3] + a[1][4]

Loop carried dependences

49

Dependences in Loops (IV)

```
 for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
 }}
```

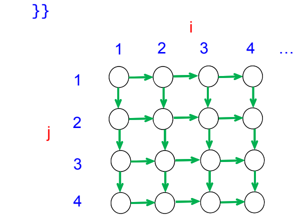
	i=1	i=2
j=1	a[1][1] = a[1][0] + a[0][1]	a[2][1] = a[2][0] + a[1][1]
j=2	a[1][2] = a[1][1] + a[0][2]	a[2][2] = a[2][1] + a[1][2]
j=3	a[1][3] = a[1][2] + a[0][3]	a[2][3] = a[2][2] + a[1][3]
j=4	a[1][4] = a[1][3] + a[0][4]	a[2][4] = a[2][3] + a[1][4]

Loop carried dependences

50

Dependences in Loops (IV)

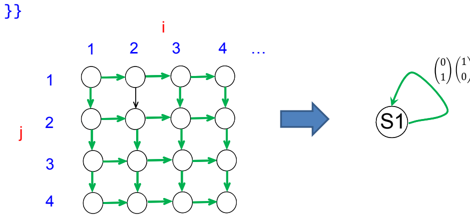
```
 for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
 }}
```



51

Dependences in Loops (IV)

```
 for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
 }}
```



52

Data dependences and vectorization

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

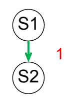
```
   for (i=0; i<n; i++){
S1   a[i] = b[i] + 1;        a[0:n-1] = b[0:n-1] + 1;
   }
```

53

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



54

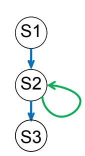
Data dependences and transformations

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

55

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;
  }
```

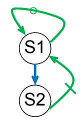



```
b[1:n-1] = b[1:n-1] + c[1:n-1];
for (i=1; i<n; i++){
      a[i] = a[i-1]*a[i-2]+b[i];
}
c[1:n-1] = a[1:n-1] + 1;
```

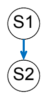
56

Removing dependences

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



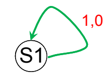

```
    for (i=0; i<n; i++){
S1    a'[i] = b[i] + 1;
S2    c[i] = a'[i] + 2;
    }
    a=a'[n-1]
```




```
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];
}
```

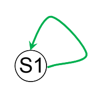

Changing the algorithm

- 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 or recurrences include:
  - Reductions (`S+=A[i]`)
  - Linear recurrences (`A[i]=B[i]*A[i-1]+C[i]`)
  - Boolean recurrences (`if (A[i]>max) max = A[i]`)

Changing the algorithm (cont.)

.

```
S1  a[0]=b[0];
    for (i=1; i<n; i++)
S2      a[i]=a[i-1]+b[i];
```




```
a[0:n-1]=b[0:n-1];
for (i=0;i<k;i++)  /* n = 2^k */
    a[2**i:n-1]=a[2**i:n-1]+b[0:n-2**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;
}
```


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

Loop Vectorization

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

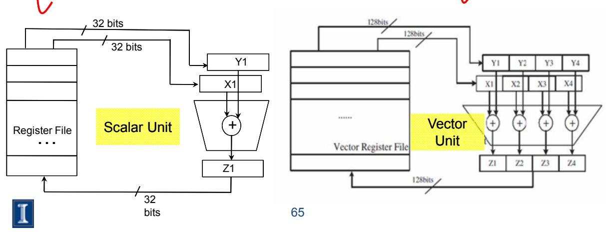
Simple Example

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

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

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

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



Loop Vectorization

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

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

→

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

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

S1 S1 S1 S1 S1 S1 S1 S1

S2 S2 S2 S2 S2 S2 S2 S2

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

S1 S1 S1 S1 S1 S1 S1 S1

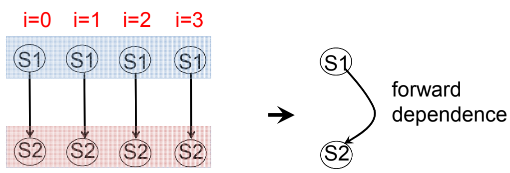
S2 S2 S2 S2 S2 S2 S2 S2

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

S113

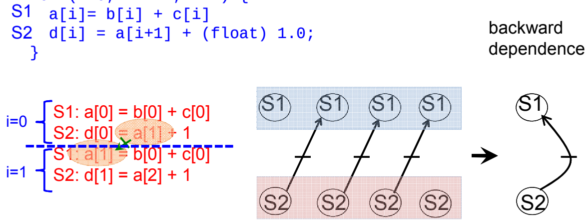
```
for (i=0; i<LEN; i++) {
  a[i]= b[i] + c[i]
  d[i] = a[i] + (float) 1.0;
}
```

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;
 }
```



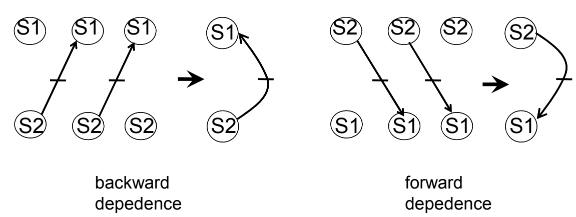
This loop cannot be vectorized as it is

Acyclic Dependenden Graphs Backward Dependences (I)

Reorder of statements

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

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

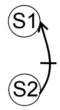


Acyclic Dependenden Graphs Backward Dependences (I)

S114

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

S114

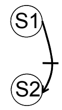


Intel Nehalem
Compiler report: Loop was not vectorized. Existence of vector dependence
Exec. Time scalar code: 12.6
Exec. Time vector code: --
Speedup: --

S114_1

```
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 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

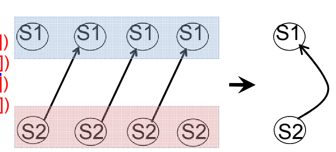
IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 1.2
Exec. Time vector code: 0.6
Speedup: 2.0

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

i=1 S1: a[1] = d[0] + sqrt(c[1])
S2: d[1] = b[1] + sqrt(e[1])
i=2 S1: a[2] = d[1] + sqrt(c[2])
S2: d[2] = b[2] + sqrt(e[2])



This loop cannot be vectorized as it is

Acyclic Dependenden Graphs Backward Dependences (II)

S214

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

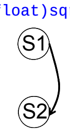
S114

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

S214_1

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

S114_1



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

Acyclic Dependenden Graphs Backward Dependences (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

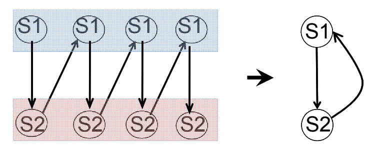
IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 3.3
Exec. Time vector code: 1.8
Speedup: 1.8

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)

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;
}
```

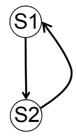
S115

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

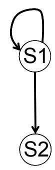
This loop is not vectorized

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

b[i]

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?

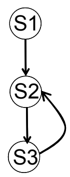
No, the compiler does not vectorize S215 because it is not cost-effective

```
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)

85

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

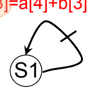
IBM Power 7
Compiler report: Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization
Exec. Time scalar code: 13.5
Exec. Time vector code: --
Speedup: --

86

Cycles in the DG (III)

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

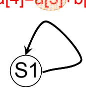
a[0]=a[1]+b[0]
a[1]=a[2]+b[1]
a[2]=a[3]+b[2]
a[3]=a[4]+b[3]



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]



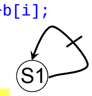
Self true-dependence
can not vectorized
(as it is)

87

Cycles in the DG (III)

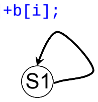
S117

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



S118

```
   for (int i=1;i<LEN;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

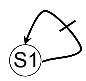
Intel Nehalem
Compiler report: Loop was not vectorized. Existence of vector dependence
Exec. Time scalar code: 7.2
Exec. Time vector code: --
Speedup: --

88

Cycles in the DG (III)

S117

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

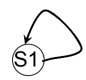


S117

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

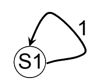
IBM Power 7
Compiler report: : Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization
Exec. Time scalar code: 7.2
Exec. Time vector code: --
Speedup: --

89

Cycles in the DG (IV)

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

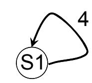
a[1]=a[0]+b[1]
a[2]=a[1]+b[2]
a[3]=a[2]+b[3]



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]



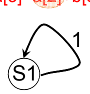
This is also a self-true dependence. But … can it be vectorized?

90

Cycles in the DG (IV)

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

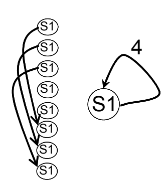
a[1]=a[0]+b[1]
a[2]=a[1]+b[2]
a[3]=a[2]+b[3]



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.

91

Cycles in the DG (IV)

S119

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

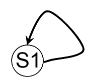
Intel Nehalem
Compiler report: Loop was vectorized
Exec. Time scalar code: 8.4
Exec. Time vector code: 3.9
Speedup: 2.1

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 6.6
Exec. Time vector code: 1.8
Speedup: 3.7

92

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;
   }
```

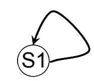


Can this loop be vectorized?

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?

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

```
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

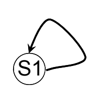
IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 3.9
Exec. Time vector code: 1.8
Speedup: 2.1

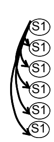
Cycles in the DG (VI)

- 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' ≠ 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 uses 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 uses 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

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

101

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)
```

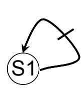
102

Compiler Directives (I)

- This loop can be vectorized when k < -3 and k >= 0.
- Programmer knows that k>=0

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

If (k >= 0) → no dependence or self-anti-dependence

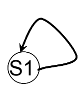


k =1

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

Can be vectorized

If (k <0) → 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

103

Compiler Directives (I)

- This loop can be vectorized when k < -3 and k >= 0.
- Programmer knows that k>=0

How can the programmer tell the compiler that k >= 0

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

104

Compiler Directives (I)

- This loop can be vectorized when k < -3 and k >= 0.
- Programmer knows that k>=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

105

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

106

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 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 vectoriced 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)

107

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)
```

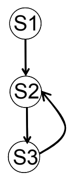
108

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);
}
```

➔

```
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_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

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. Time scalar code: 1.3
Exec. Time vector code: --
Speedup: --

S126_1

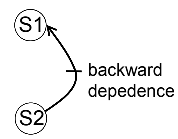
IBM Power 7
Compiler report:
- Loop 1 was SIMD vectorized.
- Loop 2 was not SIMD vectorized
Exec. Time scalar code: 1.14
Exec. Time vector code: 1.0
Speedup: 1.14

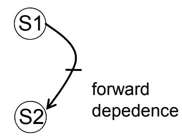
113

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];
   }
```





114

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

115

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

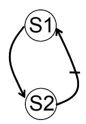
IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 3.3
Exec. Time vector code: 1.8
Speedup: 1.8

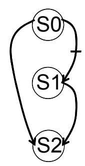
116

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

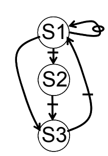
S126_1

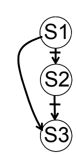
IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 5.1
Exec. Time vector code: 2.4
Speedup: 2.0

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_1

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

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

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];
```

→

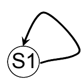
```
if (N>=1)
  A[0] = B[0] + C[0];
for (i=1; i<N; i++)
  A[i] = B[i] + C[i];
```

Loop Peeling

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

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

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

Loop Peeling

S127

S1
```
for (int i=0;i<LEN;i++){
  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 Peeling

S127

```
for (int i=0;i<LEN;i++)
{
  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_2

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

S127

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Time scalar code: 2.4
Time vector code: --
Speedup: --

S127_1

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. scalar code: 2.4
Exec. vector code: --
Speedup: --

S127_2

IBM Power 7
Compiler report: Loop was vectorized
Exec. scalar code: 1.58
Exec. vector code: 0.62
Speedup: 2.54

Loop Interchanging

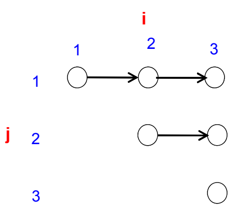
- This transformation switches the positions of one loop that is tightly nested within another loop.

```
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;
 }}
```



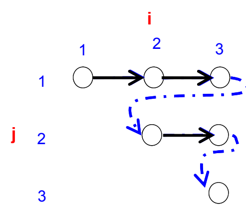
j=1
- i=1 A[1][1]=A[0][1] +1
- i=2 A[2][1]=A[1][1] + 1
- i=3 A[3][1]=A[2][1] + 1

j=2
- i=2 A[2][2]=A[1][2] +1
- i=3 A[3][2]=A[2][2] +1

j=3
- i=3 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;
 }}
```



j=1
- i=1 A[1][1]=A[0][1] +1
- i=2 A[2][1]=A[1][1] + 1
- i=3 A[3][1]=A[2][1] + 1

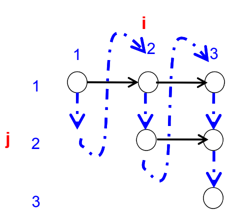
j=2
- i=2 A[2][2]=A[1][2] +1
- i=3 A[3][2]=A[2][2] +1

j=3
- i=3 A[3][3]=A[2][3] +1

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;
 }}
```



i=1
- j=1 A[1][1]=A[0][1] +1

i=2
- j=1 A[2][1]=A[1][1] + 1
- j=2 A[2][2]=A[1][2] + 1

i=3
- j=1 A[3][1]=A[2][1] +1
- j=2 A[3][2]=A[2][2] +1
- j=3 A[3][3]=A[2][3] +1

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

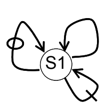
1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
   – Data Dependences
     • Reductions
   – Data Alignment
   – Aliasing
   – Non-unit strides
   – Conditional Statements
4. Vectorization using intrinsics

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];
}
```

S132

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

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. Time scalar code: 4.4
Exec. Time vector code: --
Speedup: --

Reductions

S141_1

```
for (int i = 0; i < 64; i++){
  max[i] = a[i];
  loc[i] = i; }
for (int i = 0; i < LEN; i+=64){
 for (int j=0, k=i; k<i+64;
k++,j++){
     int cmp = max[j] < a[k];
     max[j] = cmp ? a[k] : max[j];
     loc[j] = cmp ? k : loc[j];
} }
MAX = max[0];
LOC = 0;
for (int i = 0; i < 64; i++){
  if (MAX < max[i]){
     MAX = max[i];
     LOC = loc[i];
 } }
```

S141_1

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 10.2
Exec. Time vector code: 2.7
Speedup: 3.7

S141_2

IBM Power 7
A version written with intrinsics runs in 1.6 **secs.**

Outline

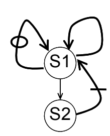
1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
   - Data Dependences
     - Induction variables
   - Data Alignment
   - Aliasing
   - Non-unit strides
   - Conditional Statements
4. Vectorization with intrinsics

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];
}
```

```
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_1

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

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

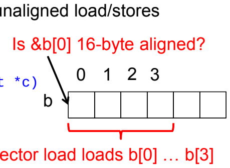
Outline

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

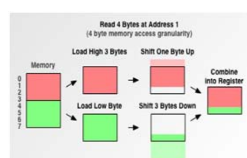
- Vector loads/stores load/store 128 consecutive bits to a vector register.
- Data addresses need to be 16-byte (128 bits) aligned to be loaded/stored
  - Intel platforms support aligned and unaligned load/stores
  - IBM platforms do not support unaligned load/stores

```
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 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);
}}

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);
}}
```

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:
0x7fffe6765f00, 0x7fffe6765f04, 0x7fffe6766004, 0x7fffe6766008

- 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-byes (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 withintrinsics

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];
}
```

157

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]

158

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

159

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];
}
```

160

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

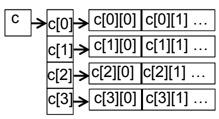
- Example with 2D arrays: pointer-to-pointer declaration.

```
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];
}
```



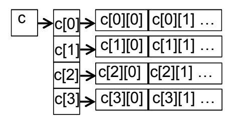
__restrict__ only qualifies the first dereferencing of c;

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 – Multidemensional 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);
}
```

169

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];
}}
```

170

Outline

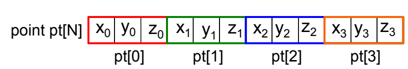
1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
   - Data Dependences
   - Data Alignment
   - Aliasing
   - Non-unit strides
   - Conditional Statements
4. Vectorization with intrinsics

171

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;
}
```

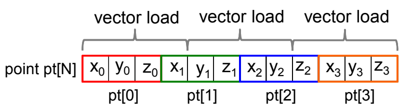


172

Non-unit Stride – Example I

- Array of a struct

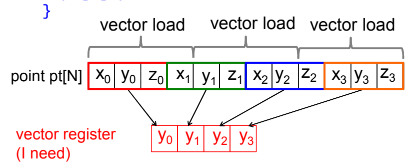
```
typedef struct{int x, y, z}
point;
point pt[LEN];

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



Non-unit Stride – Example I

- Array of a struct

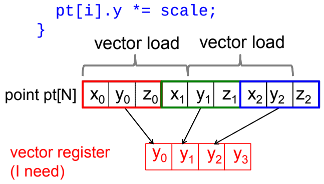
```
typedef struct{int x, y, z}
point;
point pt[LEN];

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



Non-unit Stride – Example I

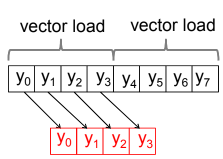
- Array of a struct

```
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_1

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

for (int i=0; i<LEN; i++) {
  pty[i] *= 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: --

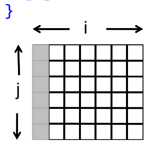
S135_1

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 1.8
Exec. Time vector code: 1.5
Speedup: 1.2

177

Non-unit Stride – Example II

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

178

Non-unit Stride – Example II

S136

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

S136_1

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

S136_2

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

S136

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

S136_1

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

S136_2

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

179

Non-unit Stride – Example II

S136

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

S136_1

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

S136_2

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

S136

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. Time scalar code: 2.0
Exec. Time vector code: --
Speedup: --

S136_1

IBM Power 7
report: Loop interchanging applied. Loop was SIMD vectorized
scalar code: 0.4
vector code: 0.2
Speedup: 2.0

S136_2

IBM Power 7
report: Loop interchanging applied. Loop was SIMD
scalar code: 0.4
vector code: 0.16
Speedup: 2.7

180

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

181

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];
}
```

182

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

183

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

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

compiled with flag -qdebug=alwaysspec

S137

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 4.0
Exec. Time vector code: 1.5
Speedup: 2.5

S137_1

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 4.0
Exec. Time vector code: 1.5
Speedup: 2.5

184

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];
}
```


Conditional Statements

```
for (int i=0;i<1024;i++){
  if (c[i] < (float) 0.0)
    a[i]=a[i]*b[i]+d[i];
}
```

rC	2	-1	1	-2
rCmp	False	True	False	True
rThen	0	3.2	0	3.2
rElse	1.	0	1.	0
rS	1.	3.2	1.	3.2

```
vector bool char = rCmp
vector float r0={0.,0.,0.,0.};
vector float rA,rB,rC,rD,rS, rT,
rThen,rElse;
for (int i=0;i<1024;i+=4){
 // load rA, rB, and rD;
  rCmp = vec_cmplt(rC, r0);
  rT= rA*rB+rD;
  rThen = vec_and(rT.rCmp);
  rElse = vec_andc(rA.rCmp);
  rS = vec_or(rthen, relse);
  //store rS
}
```


Conditional Statements

```
for (int i=0;i<1024;i++){
  if (c[i] < (float) 0.0)
    a[i]=a[i]*b[i]+d[i];
}
```

Speedups will depend on the values on c[i]

Compiler tends to be conservative, as the condition may prevent from segmentation faults

```
vector bool char = rCmp
vector float r0={0.,0.,0.,0.};
vector float rA,rB,rC,rD,rS, rT,
rThen,rElse;
for (int i=0;i<1024;i+=4){
 // load rA, rB, and rD;
  rCmp = vec_cmplt(rC, r0);
  rT= rA*rB+rD;
  rThen = vec_and(rT.rCmp);
  rElse = vec_andc(rA.rCmp);
  rS = vec_or(rthen, relse);
  //store rS
}
```


Compiler Directives

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

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

Compiler Directives

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

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

193

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) singe 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)

194

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);
```

195

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);
}}
```

196

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);
}}
```

197

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];
 }
}
```

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);
}}
```

198

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

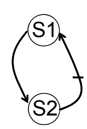
```
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float
a[n], b[n], c[n];

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);
}}
```

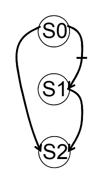
199

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
 }
```





200

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);
}}
```

201

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);
}}
```

202

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

203

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

204

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.

205

Data Dependences

- The correctness of many many loop transformations including vectorization can be decided using dependences.
- A good introduction to the notion of dependence and its applications can be found in D. Kuck, R. Kuhn, D. Padua, B. Leasure, M. Wolfe: Dependence Graphs and Compiler Optimizations. POPL 1981.

206

Compiler Optimizations

- For a longer discussion see:
  - Kennedy, K. and Allen, J. R. 2002 Optimizing Compilers for Modern Architectures: a Dependence-Based Approach. Morgan Kaufmann Publishers Inc.
  - U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell, Mass., 1988.
  - Advanced Compiler Optimizations for Supercomputers, by David Padua and Michael Wolfe in Communications of the ACM, December 1986, Volume 29, Number 12.
  - Compiler Transformations for High-Performance Computing, by David Bacon, Susan Graham and Oliver Sharp, in ACM Computing Surveys, Vol. 26, No. 4, December 1994.

207

Algorithms

- W. Daniel Hillis and Guy L. Steele, Jr.. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (December 1986), 1170-1183.
- Shyh-Ching Chen, D.J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems," IEEE Transactions on Computers, pp. 701-717, July, 1975

208

Thank you

Questions?

María Garzarán, Saeed Maleki
William Gropp and David Padua
{garzaran,maleki,wgropp,padua}@illinois.edu

Program Optimization Through Loop Vectorization

María Garzarán, Saeed Maleki
William Gropp and David Padua
{garzaran,maleki,wgropp,padua}@illinois.edu
Department of Computer Science
University of Illinois at Urbana-Champaign

Back-up Slides

Measuring execution time

```
time1 = time();

 for (i=0; i<32000; i++)
  c[i] = a[i] + b[i];

time2 = time();
```

Measuring execution time

- Added an outer loop that runs (serially)
  - to increase the running time of the loop

```
time1 = time();
for (j=0; j<200000; j++){
 for (i=0; i<32000; i++)
  c[i] = a[i] + b[i];

}
time2 = time();
```

213

Measuring execution times

- Added an outer loop that runs (serially)
  - to increase the running time of the loop
- Call a dummy () function that is compiled separately
  - to avoid loop interchange or dead code elimination

```
time1 = time();
for (j=0; j<200000; j++){
 for (i=0; i<32000; i++)
  c[i] = a[i] + b[i];
 dummy();
}
time2 = time();
```

214

Measuring execution times

- Added an outer loop that runs (serially)
  - to increase the running time of the loop
- Call a dummy () function that is compiled separately
  - to avoid loop interchange or dead code elimination
- Access the elements of one output array and print the result
  - to avoid dead code elimination

```
time1 = time();
for (j=0; j<200000; j++){
 for (i=0; i<32000; i++)
  c[i] = a[i] + b[i];
 dummy();
}
time2 = time();
for (j=0; j<32000; j++)
    ret+= a[i];
printf ("Time %f, result %f", (time2 –time1), ret);
```

215

Compiling

- Intel icc scalar code

  icc -O3 –no-vec dummy.o tsc.o –o runnovec
- Intel icc vector code

  icc -O3 –vec-report[n] –xSSE4.2 dummy.o tsc.o –o runvec

[n] can be 0,1,2,3,4,5

– vec-report0, no report is generated

– vec-report1, indicates the line number of the loops that were vectorized

– vec-report2 .. 5, gives a more detailed report that includes the loops that were not vectorized and the reason for that.

216

Compiling

flags = -O3 –qaltivec -qhot -qarch=pwr7 -qtune=pwr7
-qipa=malloc16 -qdebug=NSIMDCOST
-qdebug=alwaysspec –qreport

- IBM xlc scalar code
  xlc -qnoenablevmx dummy.o tsc.o –o runnovec
- IBM vector code
  xlc –qenablevmx dummy.o tsc.o –o runvec

217

Strip Mining

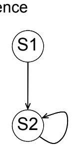
This transformation improves locality and is usually combined with vectorization

218

Strip Mining

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

- first statement can be vectorized
- second statement cannot be vectorized because of self-true dependence



→

```
for (i=1; i<LEN; i++)
  a[i]= b[i];

for (i=1; i<LEN; i++)
  c[i] = c[i-1] + a[i];
```

By applying loop distribution the compiler will vectorize the first statement

But, … loop distribution will increase the cache miss ratio if array a[] is large

219

Strip Mining

Loop Distribution

```
for (i=1; i<LEN; i++)
  a[i]= b[i];
for (i=1; i<LEN; i++)
  c[i] = c[i-1] + a[i];
```

→

Strip Mining

```
for (i=1; i<LEN;
i+=strip_size){
  for (j=i; j<strip_size; j++)
    a[j]= b[j];
  for (j=i; j<strip_size; j++)
    c[j] = c[j-1] + a[j];
}
```

strip_size is usually a small value (4, 8, 16 or 32).

220

Strip Mining

- Another example

```
int v[N];
…
for (int i=0;i<N;i++){
  Transform (v[i]);
}
for (int i=0;i<N;i++){
  Light (v[i]);
}
```

```
int v[N];
…
for (int i=0;i<N;i+=strip_size){
  for (int j=i;j<strip_size;j++){
    Transform (v[j]);
  }
  for (int j=i;i<strip_size;j++){
    Light (v[j]);
  }
}
```


What are Microprocessor vector extensions?

- Instructions that execute on SIMD (Single Instruction Multiple Data) functional units

  – They operate on short vectors of integers and floats (from 16-way on 1-byte (chars) to 2-way (doubles))
- Why are they here?
  – **Useful:** Many applications (e.g., multimedia) feature the required fine grain parallelism – code potentially faster
  – **Doable:** Chip designers have enough transistors available, easy to implement

 Based on slide provided by Markus Püschel

Overview Floating-Point Vector ISAs

Vendor	Name	n-ways	Precision	Introduced with
Intel	SSE	4-way	single	Pentium III
	SSE2	+ 2-way	double	Pentium 4
	SSE3			Pentium 4(Prescott
	SSSE3			Core Duo
	SSE4			Core2 Extreme (Penryn)
	AVX	+ 4-way	double	Sandy Bridge, 2011
Intel	IPF	2-way	single	Itanium
AMD	3DNOW!	2-way	single	K6
	Enhanced 3DNow!			K7
	3DNow! Professional	+ 4-way	single	AthlonXP
	AMD64	+ 2-way	double	Opteron
Motorola	AltiVec	4-way	single	MPC 7400 G4
IBM	VMX	4-way	single	PowerPC 970 G5
	SPU	+ 2-way	double	Cell BE
	Double FPU	2-way	double	PowerPC 440 FP2

A similar architecture is found in game consoles (PS2, PS3) and GPUs (NVIDIA's GeForce)

 Based on slide provided by Markus Püschel

What are Microprocessor vector extensions?

- Instructions that execute on SIMD (Single Instruction Multiple Data) functional units

  – They operate on short vectors of integers and floats (from 16-way on 1-byte (chars) to 2-way (doubles))
- Why are they here?
  – **Useful:** Many applications (e.g., multimedia) feature the required fine grain parallelism – code potentially faster
  – **Doable:** Chip designers have enough transistors available, easy to implement

 Based on slide provided by Markus Püschel

Overview Floating-Point Vector ISAs

Vendor	Name	n-ways	Precision	Introduced with
Intel	SSE	4-way	single	Pentium III
	SSE2	+ 2-way	double	Pentium 4
	SSE3			Pentium 4(Prescott
	SSSE3			Core Duo
	SSE4			Core2 Extreme (Penryn)
	AVX	+ 4-way	double	Sandy Bridge, 2011
Intel	IPF	2-way	single	Itanium
AMD	3DNOW!	2-way	single	K6
	Enhanced 3DNow!			K7
	3DNow! Professional	+ 4-way	single	AthlonXP
	AMD64	+ 2-way	double	Opteron
Motorola	AltiVec	4-way	single	MPC 7400 G4
IBM	VMX	4-way	single	PowerPC 970 G5
	SPU	+ 2-way	double	Cell BE
	Double FPU	2-way	double	PowerPC 440 FP2

A similar architecture is found in game consoles (PS2, PS3) and GPUs (NVIDIA's GeForce)

Based on slide provided by Markus Püschel

Evolution of Intel Vector Instructions

- MMX (1996, Pentium)
  - *CPU-based MPEG decoding*
  - Integers only, 64-bit divided into 2 x 32 to 8 x 8
  - Phased out with SSE4
- SSE (1999, Pentium III)
  - *CPU-based 3D graphics*
  - 4-way float operations, single precision
  - 8 new 128 bit Register, 100+ instructions
- SSE2 (2001, Pentium 4)
  - *High-performance computing*
  - Adds 2-way float ops, double-precision; same registers as 4-way single-precision
  - Integer SSE instructions make MMX obsolete
- SSE3 (2004, Pentium 4E Prescott)
  - *Scientific computing*
  - New 2-way and 4-way vector instructions for complex arithmetic
- SSSE3 (2006, Core Duo)
  - Minor advancement over SSE3
- SSE4 (2007, Core2 Duo Penryn)
  - *Modern codecs, cryptography*
  - New integer instructions
  - Better support for unaligned data, super shuffle engine

More details at http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

Run-Time Symbolic Resoltuion

```
    for (int i=3;i<n;i++){
S1    a[i+t] = a[i]+b[i];
    }
```

If (t > 0) → self true dependence

t =1

a[4]=a[3]+b[3]
a[5]=a[4]+b[4]
a[6]=a[5]+b[5]

Cannot be vectorized

If (t <=0) → no dependence or self anti-depence

t =-1

a[2]=a[3]+b[3]
a[3]=a[4]+b[4]
a[4]=a[5]+b[5]

Can be vectorized

Loop Vectorization – Example I

S113

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

S113_1

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

The Intel ICC compiler generated the same code in both cases

S113

Intel Nehalem
Compiler report: Loop was vectorized in both cases
Exec. Time scalar code: 10.2
Exec. Time vector code: 6.3
Speedup: 1.6

S113_1

Intel Nehalem
Compiler report: Fused loop was vectorized
Exec. Time scalar code: 10.2
Exec. Time vector code: 6.3
Speedup: 1.6

Loop Vectorization – Example I

S113

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

S113_1

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

S113

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 3.1
Exec. Time vector code: 1.5
Speedup: 2.0

S113_1

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 3.7
Exec. Time vector code: 2.3
Speedup: 1.6

How do we access the SIMD units?

- Three choices
  1. C code and a vectorizing compiler
  1. Macros or Vector Intrinsics
  1. Assembly Language

How do we access the SIMD units?

- Three choices
  1. C code and a vectorizing compiler
  1. Macros or Vector Intrinsics
  1. Assembly Language

```
for (i=0; i<32000; i++)
  c[i] = a[i] + b[i];
```


How do we access the SIMD units?

- Three choices
  1. C code and a vectorizing compiler
  1. Macros or Vector Intrinsics
  1. Assembly Language

```
void example(){
__m128 rA, rB, rC;
 for (int i = 0; i < 32000; 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);
}}
```

How do we access the SIMD units?

- Three choices

1. C code and a vectorizing compiler

1. Macros or Vector Intrinsics

1. Assembly Language

```
..B8.5
   movaps    a(,%rdx,4), %xmm0
   movaps    16+a(,%rdx,4), %xmm1
   addps     b(,%rdx,4), %xmm0
   addps     16+b(,%rdx,4), %xmm1
   movaps    %xmm0, c(,%rdx,4)
   movaps    %xmm1, 16+c(,%rdx,4)
   addq      $8, %rdx
   cmpq      $32000, %rdx
   jl        ..B8.5
```

233

Why should the compiler vectorize?

1. Easier
2. Portable across vendors and across generations of the same class of machines
   – Although compiler directives maybe different across compilers
3. Better performance than programmer generated vector code
   – Compiler applies other transformations such as loop unrolling, instruction scheduling …

Compilers make your codes (almost) machine independent

But, compilers fail:
- Programmers need to provide the necessary information
- Programmers need to transform the code

234

How well do compilers vectorize?

- Results for
  – Test Suite for Vectorizing compilers by David Callahan, Jack Dongarra and David Levine.
  – IBM XLC compiler, version 11

Total	159				
Auto Vectorized	74	Not Vectorized	85		
		Vectorizable by	60	Impossible to Vectorize	25
		Classic Transformation	35	Non-unit Stride Access	16
		New Transformation	6	Data Dependence	5
		Manual Vectorization	19	Other	4

235
235

How well do compilers vectorize?

Loops	Percentage
Vectorizable	84.3%
- Vectorized	46.5%
- Classic transformation applied	22.0%
- New transformation applied	3.8%
- Manual vector code	11.9%

236

Terminology

Transformations	Explanation
Classic transformation (source level)	• Loop Interchange • Scalar Expansion • Scalar and Array Renaming • Node Splitting • Reduction • Loop Peeling • Loop Distribution • Run-Time Symbolic Resolution • Speculating Conditional Statements
New Transformation (Intrinsics)	• Manually vectorized Matrix Transposition • Manually vectorized Prefix Sum
Manual Transformation (Intrinsics)	• Auto vectorization is inefficient • Vectorization of the transformed code is inefficient • No transformation found to enable auto vectorization

237

What are the speedups?

- Speedups obtained by XLC compiler

Test Suite Collection	Average Speed Up
Automatically by XLC	1.73
By adding classic transformations	3.48
By adding new transformations	3.64
By adding manual vectorization	3.78

Why did the compilers fail to vectorize?

Issues	ICC	XLC
Vectorizable but not automatic	59	60
Cyclic data dependence	9	8
Non-Unit stride access	6	7
Conditional statement	6	9
Aliasing	5	0
Acyclic data dependence	4	0
Reduction	4	5
Unsupported loop structure	4	9
Loop Interchange	4	2
Wrap around	1	4
Scalar expansion	3	4
Other	13	12

XLC and ICC Comparison

Compiler / Reasons	XLC but not ICC	ICC but not XLC
Vectorized	25	26
Aliasing	5	0
Acyclic data dependence	4	0
Unrolled loop	3	0
Loop interchange	2	0
Conditional statements	4	7
Unsupported loop structure	0	5
Wrap around variable	0	3
Reduction	1	2
Non-unit stride access	1	2
Other	5	6

Another example: matrix-matrix multiplication

S111

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

Intel Nehalem
Compiler report: Loop was not vectorized: existence of vector dependence
Exec. Time scalar code: 2.5 sec
Exec. Time vector code: --
Speedup: --

IBM Power 7
Compiler report: Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization.
Exec. Time scalar code: 0.74 sec
Exec. Time vector code: --
Speedup: --

241

Another example: matrix-matrix multiplication

S111_1

Added compiler directives

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

Intel Nehalem
Compiler report: Loop was not vectorized: existence of vector dependence
Exec. Time scalar code: 2.5 sec
Exec. Time vector code: --
Speedup: --

IBM Power 7
Compiler report: Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization.
Exec. Time scalar code: 0.74 sec
Exec. Time vector code: --
Speedup: --

242

Another example: matrix-matrix multiplication

S111_3

Loop interchagne

```
void MMM(float** __restrict__ a,float** __restrict__ b,
float** __restrict__ c) {
for (int i = 0; i < LEN2; i++)
   for (int j = 0; j < LEN2; j++)
      C[i][j] = (float)0.;
for (int i=0; i<LEN; i++)
   for (int k=0; k<LEN; j++){
      for (int j=0; j<LEN; k++)
         c[i][j] += a[i][k] * b[k][j];
}}
```

Intel Nehalem
Compiler report: Loop was vectorized
Exec. Time scalar code: 0.8 sec
Exec. Time vector code: 0.3 sec
Speedup: 2.7

IBM Power 7
Compiler report: Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization.
Exec. Time scalar code: 0.74 sec
Exec. Time vector code: --
Speedup: --

243

Definition of Dependence

- A statement S is said to be data dependent on another statement T if
  - S accesses the same data being accessed by an earlier execution of T
  - S, T or both write the data.

- Dependence analysis can be used to discover data dependences between statements

244

Data Dependence

Flow dependence (True dependence)

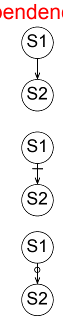
S1: X = A+B
S2: C= X+A

Anti dependence

S1: A = X + B
S2: X= C + D

Output dependence

S1: X = A+B
S2: X= C + D



Dependences in Loops

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "instances"

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

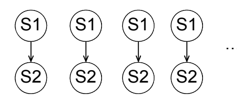
Unrolled loop

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

iteration: 0 1 2 3 …

instances of S1:

instances of S2:



Dependences in Loops

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "instances"

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

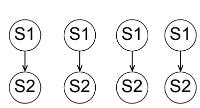
Unrolled loop

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

iteration: 0 1 2 3 …

instances of S1:

instances of S2:



Loop independent: dependence is within the loop iteration boundaries

Dependences in Loops

- A slightly more complex example

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

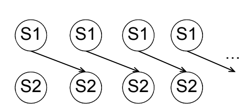
Unrolled loop

i=1 S1: a[1] = b[1] + 1
S2: c[1] = a[0] + 2
i=2 S1: a[2] = b[2] + 1
S2: c[2] = a[1] + 2
i=3 S1: a[3] = b[3] + 1
S2: c[3] = a[2] + 2
…

iteration: 1 2 3 4 …

instances of S1:

instances of S2:



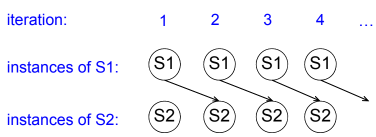
Dependences in Loops

- A slightly more complex example

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

Unrolled loop

i=1 S1: a[1] = b[1] + 1
S2: c[1] = a[0] + 2
i=2 S1: a[2] = b[2] + 1
S2: c[2] = a[1] + 2
i=3 S1: a[3] = b[2] + 1
S2: c[3] = a[2] + 2



Loop carried dependence: across iterations

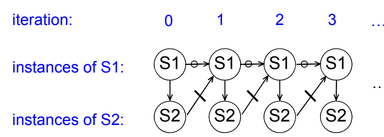
Dependences in Loops

- Even more complex

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

Unrolled loop

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



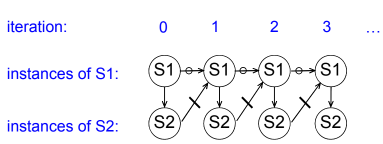
Dependences in Loops

- Even more complex

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

Unrolled loop

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

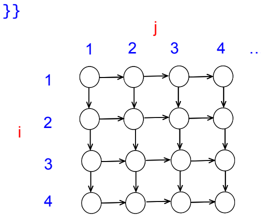


→ Loop Independent
→ Loop carried dependence

Dependences in Loops

- Two dimensional

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



Unrolled loop

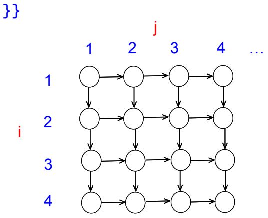
i=1
j=1 a[1][1] = a[1][0] + a[0][1]
j=2 a[1][2] = a[1][1] + a[0][2]
j=3 a[1][3] = a[1][2] + a[0][3]
j=4 a[1][4] = a[1][3] + a[0][4]

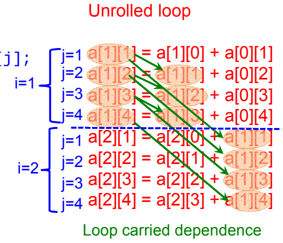
i=2
j=1 a[2][1] = a[2][0] + a[1][1]
j=2 a[2][2] = a[2][1] + a[1][2]
j=3 a[2][3] = a[2][2] + a[1][3]
j=4 a[2][4] = a[2][3] + a[1][4]

Dependences in Loops

- Two dimensional

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

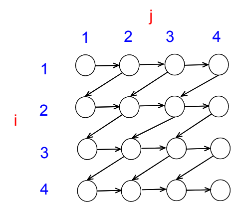


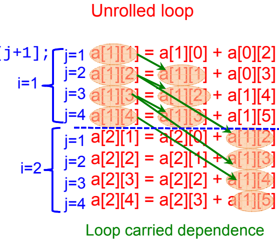


Dependences in Loops

- Another two dimensional loop

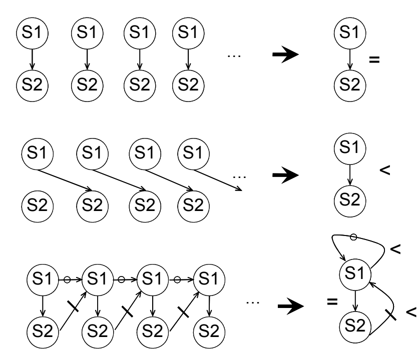
```
for (i=1; i<LEN; i++) {
  for (j=1; j<LEN; j++) {
    a[i][j]=a[i][j-1]+a[i-1][j+1];
}}
```





Dependences in Loops

- The representation of these dependence graphs inside the compiler is a "collapsed" version of the graph where the arcs are annotated with direction (or distances) to reduce ambiguities.
- In the collapsed version each statement is represented by a node and each ordered pair of variable accesses is represented by an arc



Loop Vectorization

- When the loop has several statements, it is better to first strip-mine the loop and then distribute

strip_size is usually a small value (4, 8, 16, 32)

Scalar code

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

Scalar code, loops are distributed

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

loop distribution will increase the cache miss ratio if array b[] is large

Scalar code, loops are strip-mined

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

and then distributed

```
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;
}
```


257

Loop Vectorization

Scalar code

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

Scalar code, loops are distributed

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

Scalar code, loops are strip-mined and distributed

```
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;
}
```

Vector code

```
a[0:LEN-1]=b[0:LEN-1]+(float)1.0;
c[0:LEN-1]=d[0:LEN-1]+(float)2.0;
```

Vector code

```
for (i=0; i<LEN; i+=4{
  a[i:i+3]=b[i:i+3]+(float)1.0;
  c[i:i+3]=d[i:i+3]+(float)2.0;
}
```


258

Loop Vectorization

S112

Scalar code

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

S112_1

Scalar code loops are distributed

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

S112_2

Scalar code loops are strip-mined

```
for (i=0; i<LEN; i+=4){
 for (j=i; j<i+4; j++){
   a[j]=b[j]+(float)1.0;
   c[j]=b[j]+(float)2.0;
}}
```

The Intel ICC generated the same vector code (the equivalent to the strip-mined version) in all the cases

S112 and S112_2

Intel Nehalem
Compiler report: Loop was vectorized
Exec. Time scalar code: 9.6
Exec. Time vector code: 5.5
Speedup: 1.7

S112_1

Intel Nehalem
Compiler report: Fused loop was vectorized
Exec. Time scalar code: 9.6
Exec. Time vector code: 5.5
Speedup: 1.7

Loop Vectorization

S112

Scalar code

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

S112_1

Scalar code loops are distributed

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

S112_2

Scalar code loops are strip-mined

```
for (i=0; i<LEN; i+=64){
 for (j=i; j<i+64; j++){
   a[j]=b[j]+(float)1.0;
   c[j]=b[j]+(float)2.0;
}}
```

S112

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 2.7
Exec. Time vector code: 1.2
Speedup: 2.1

S112_1

IBM Power 7
report: Loop was SIMD vectorized
scalar code: 2.7
vector code: 1.2
Speedup: 2.1

S112_2

IBM Power 7
report: Loop was SIMD vectorized
scalar code: 2.9
vector code: 1.4
Speedup: 2.07

Loop Vectoriation

- Our observations
  - Compiler generates vector code when it can apply loop distribution.
    - compiler may have to transform the code so that loop distribution is legal

Loop Vectorization – Example II

S114

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

S114	S114
Intel Nehalem **Compiler report:** Loop was not vectorized. Existence of vector dependence **Exec. Time scalar code**: 12.6 **Exec. Time vector code**: -- **Speedup:** --	**IBM Power 7** **Compiler report:** Loop was SIMD vectorized **Exec. Time scalar code:** 3.3 **Exec. Time vector code:** 1.8 **Speedup:** 1.8

Loop Vectorization – Example II

We have observed that ICC usually vectorizes only if all the dependences are forward (except for reduction and induction variables)

S114

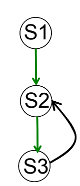
```
for (i=0; i<LEN; i++) {
  a[i]= b[i] + c[i]
  d[i] = a[i+1] + (float) 1.0;
}
```

S114	S114
Intel Nehalem **Compiler report:** Loop was not vectorized. Existence of vector dependence **Exec. Time scalar code**: 12.6 **Exec. Time vector code**: -- **Speedup:** --	**IBM Power 7** **Compiler report:** Loop was SIMD vectorized **Exec. Time scalar code:** 3.3 **Exec. Time vector code:** 1.8 **Speedup:** 1.8

Loop vectorization – Example IV

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];
   }
```




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

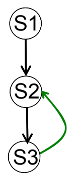
S1 can be vectorized
S2 and S3 cannot be vectorized
(A loop with a cycle in the dependence graph cannot be vectorized)

Loop vectorization – Example IV

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];
   }
```

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



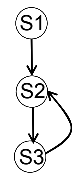
S1 can be vectorized
S2 and S3 cannot be vectorized
(A loop with a cycle in the dependence graph cannot be vectorized)

Loop vectorization – Example III

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];
   }
```

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

Loop vectorization – Example IV

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_1

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

The INTEL ICC compiler generates the same code in both cases

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_1

Intel Nehalem
Compiler report: Loop was vectorized
Exec. Time scalar code: 14.7
Exec. Time vector code: 18.1
Speedup: 0.8

Loop vectorization – Example IV

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_1

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

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: --

S116_1

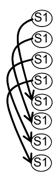
IBM Power 7
Compiler report: Loops were fused. Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization.
Exec. Time scalar code: 13.5
Exec. Time vector code: --
Speedup: --

Cycles in the DG – Example V

- Compiler needs to resolve a system of equations to determine if there are dependences.

```
   for (int i=4;i<LEN;i++){
S1   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]

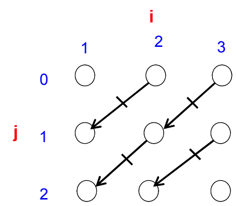


Is there a value of i such that i' = i – 4, such that i' ≠ i?

Yes, i = i' + 4

Loop Interchanging

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

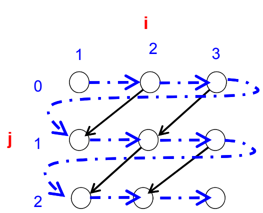


j=0: i=1 A[1][0]=A[0][1]
i=2 A[2][0]=A[1][1]
i=3 A[3][0]=A[2][1]

j=1: i=1 A[1][1]=A[0][2]
i=2 A[2][1]=A[1][2]
i=3 A[3][1]=A[2][2]

Loop Interchanging

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

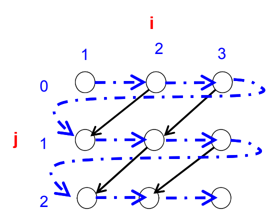


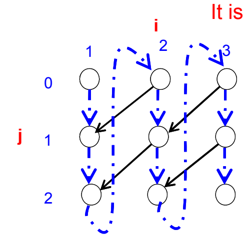
Loop Interchanging

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

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

It is illegal





272

Loop Interchanging

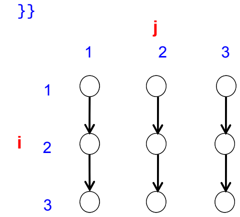
S128

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

Intel Nehalem	IBM Power 7
Compiler report: Loop was not vectorized. Vectorization possible, but inefficient	**Compiler report:** Loop was not vectorized because it is not profitable to vectorize
Exec. Time scalar code: 2.2	**Exec. Time scalar code:** 1.2
Exec. Time vector code: --	**Exec. Time vector code:** --
Speedup: --	**Speedup:** --

Loop Interchanging

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



j=1
- i=1 A[2][1]=A[1][1] + B[2]
- i=2 A[3][1]=A[2][1] + B[2]
- i=3 A[4][1]=A[3][1] + B[2]

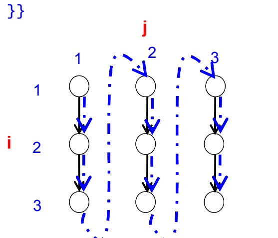
j=2
- i=1 A[2][2]=A[1][2] + B[2]
- i=2 A[3][2]=A[2][2] + B[2]
- i=3 A[4][2]=A[3][2] + B[2]

The innermost loop carries a true dependence with itself

274

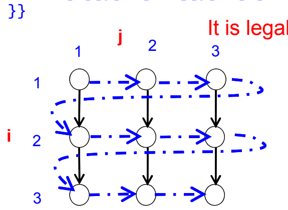
Loop Interchanging

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




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

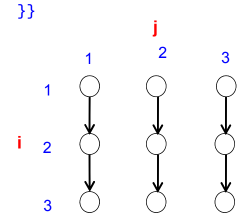
It is legal



275

Loop Interchanging

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




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

Dependence is now carried by the outer loop and the innermost can be vectorized

276

Loop Interchanging

S129

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

S129_1

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

S129

Intel Nehalem
Compiler report: Permuted loop was vectorized.
Exec. Time scalar code: 0.4
Exec. Time vector code: 0.1
Speedup: 4

S129_1

Intel Nehalem
Compiler report: Loop was vectorized.
Exec. Time scalar code: 0.4
Exec. Time vector code: 0.1
Speedup: 4

277
277

Loop Interchanging

S129

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

S129_1

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

S129

IBM Power 7
Compiler report: Loop interchanging applied. Loop was SIMD vectorized
Exec. Time scalar code: 0.5
Exec. Time vector code: 0.1
Speedup: 5

S129_1

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 0.5
Exec. Time vector code: 0.1
Speedup: 5

278
278

Intel SSE intrinsics Instructions – Examples II

- Add two vectors containing four single precision values:

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

- Multiply two vectors containing four floats:

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

- Add two vectors containing two doubles:

```
__m128d x, y;
__m128d z = _mm_add_pd(x, y);
```

279

Intel SSE intrinsics Instructions – Examples III

- Add two vectors of 8 16-bit signed integers using saturation arithmetic (*)

```
__m128i r, s;
__m128i t = _mm_adds_epi16(r, s);
```

- Compare two vectors of 16 8-bit signed integers

```
__m128i a, b;
__m128i c = _mm_cmpgt_epi8(a, b);
```

(*) [In saturation arithmetic] all operations such as addition and multiplication are limited to a fixed range between a minimum and maximum value. If the result of an operation is greater than the maximum it is set ("clamped") to the maximum, while if it is below the minimum it is clamped to the minimum. [From the wikipedia]

280

Blue Water

- Illinois has a long tradition in vectorization.
- Most recent work: vectorization for Blue Waters

Source: Thom Dunning: Blue Waters Project

Compiler work

ADVANCED COMPILER OPTIMIZATIONS FOR SUPERCOMPUTERS

Compilers for vector or multiprocessor computers must have certain optimization features to successfully generate parallel code.

DAVID A. PADUA and MICHAEL J. WOLFE

Communications of the ACM, December 1986 Volume 29, Number 12

Compiler work

ADVANCED COMPILER OPTIMIZATIONS FOR SUPERCOMPUTERS

Compilers for vector or multiprocessor computers must have certain optimization features to successfully generate parallel code.

DAVID A. PADUA and MICHAEL J. WOLFE

Communications of the ACM, December 1986 Volume 29, Number 12

Conditional Statements-II

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

- The programmer puts the condition to reduce the amount of computation, because he/she knows that the condition is true many times.

Options:

1) Remove the condition.

2) Leave it as it is.

Conditional Statements-II

```
for (int i = 0; i < LEN; ++i) {
    a[i] = CHEAP_FUNC(d[i]);
#pragma novector
for (int i = 0; i < LEN; ++i) {
  if (c[i] != 0)
    a[i] += EXPENSIVE_FUNC(b[i])*c[i];
}
```

Performance is input dependent

285

Conditional Statements-II

S138

```
for (int i = 0; i < LEN; ++i) {
  if (c[i] == (float)0.0)
    a[i] = d[i]*(float)2.0;
  else
    a[i]= c[i]/e[i]/b[i]/d[i]/

a[i]+d[i]*(float)2.0;
}
```

S138_1

```
for (int i = 0; i < 1024; ++i) {
    a[i] = d[i]*(float)2.0;
#pragma novector
for (int i = 0; i < 1024; ++i) {
  if (c[i] != (float)0.0)
    a[i] += c[i]/e[i]/b[i]/d[i]/
            a[i];
}
```

S138

Intel Nehalem
Compiler report: Loop was vectorized
Exec. Time scalar code: 1.01
Exec. Time vector code: 1.09
Speedup: 0.9

S138_1

Intel Nehalem
Compiler report:
-Loop 1 was vectorized.
-Loop 2 was not vectorized: #pragma novector used
Exec. Time scalar code: 0.99
Exec. Time vector code: 0.97
Speedup: 1.0

286

Conditional Statements-II

S138

```
for (int i = 0; i < LEN; ++i) {
  if (c[i] == (float)0.0)
    a[i] = d[i]*(float)2.0;
  else
    a[i]= c[i]/e[i]/b[i]/d[i]/

a[i]+d[i]*(float)2.0;
}
```

S138_1

```
for (int i = 0; i < 1024; ++i) {
    a[i] = d[i]*(float)2.0;
#pragma nosimd
for (int i = 0; i < 1024; ++i) {
  if (c[i] != (float)0.0)
    a[i] += c[i]/e[i]/b[i]/d[i]/
            a[i];
}
```

S138

IBM Power 7
Compiler report: Loop was not SIMD vectorized because it is not profitable
Exec. Time scalar code: 2.1
Exec. Time vector code: --
Speedup: --

S138_1

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 2.06
Exec. Time vector code: 2.6
Speedup: 0.8

287