Program Optimization Through Loop Vectorization
Topics covered

• What are the microprocessor vector extensions or SIMD (Single Instruction Multiple Data Units)

• Overcoming limitations to SIMD-Vectorization
  – Data Dependences
  – Data Alignment
  – Aliasing
  – Non-unit strides
  – Conditional Statements

• Vectorization with intrinsics
Simple Example

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

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

```
lw $t0, 0($a0)
lw $t1, 0($a1)
add $t3, $t0, $t1
sw $t3, 0($a3)
```

```
lwv $vt0, 0($a0)
lwv $vt1, 0($a1)
addv $vt3, $vt0, $vt1
swv %vt3, $0($a3)
```
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.

• Newer processors, such as Sandy or Ivy Bridge have AVX that support 256-bit vector registers.
Experimental results

• Results are shown for 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 xl compiler, version 11.0
OS Red Hat Linux Enterprise 5.4

The examples use single precision floating point numbers
Executing Our Simple Example

\[ \text{for } (i=0; i<n; i++) \]
\[ c[i] = a[i] + b[i]; \]

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

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

  2. Macros or Vector Intrinsics

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

  3. C code and a vectorizing compiler
Why use compiler vectorization?

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
How well do compilers vectorize?

<table>
<thead>
<tr>
<th>Loops</th>
<th>Compiler</th>
<th>XLC</th>
<th>ICC</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
<td>159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectorized</td>
<td></td>
<td>74</td>
<td>75</td>
<td>32</td>
</tr>
<tr>
<td>Not vectorized</td>
<td></td>
<td>85</td>
<td>84</td>
<td>127</td>
</tr>
<tr>
<td>Average Speed Up</td>
<td></td>
<td>1.73</td>
<td>1.85</td>
<td>1.30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loops</th>
<th>Compiler</th>
<th>XLC but not ICC</th>
<th>ICC but not XLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorized</td>
<td></td>
<td>25</td>
<td>26</td>
</tr>
</tbody>
</table>

![Venn Diagram showing the distribution of vectorizable loops between XLC, ICC, and both](image)
How well do compilers vectorize?

<table>
<thead>
<tr>
<th>Loops</th>
<th>Compiler</th>
<th>XLC</th>
<th>ICC</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
<td>159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectorized</td>
<td></td>
<td>74</td>
<td>75</td>
<td>32</td>
</tr>
<tr>
<td>Not vectorized</td>
<td></td>
<td>85</td>
<td>84</td>
<td>127</td>
</tr>
<tr>
<td>Average Speed Up</td>
<td></td>
<td>1.73</td>
<td>1.85</td>
<td>1.30</td>
</tr>
</tbody>
</table>

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

• Compilers can vectorize for us, but they may fail:

1. Code cannot be vectorized due to data dependences: vectorization will produce incorrect results.

2. Code can be vectorized, but the compiler fail to vectorize the code in its current form

   1. Programmer can use compiler directives to give the compiler the necessary information

   2. Programmer can transform the code
Example

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

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

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

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

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

Power 7
Compiler report: Loop was not vectorized.
Exec. Time scalar code: 2.3
Exec. Time vector code: --
Speedup: --

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

Power 7
Compiler report: Loop was vectorized.
Exec. Time scalar code: 1.6
Exec. Time vector code: 0.6
Speedup: 2.7
Vectorization is not always legal

- Vectorization of some codes could produce incorrect results
- Compilers (and programmers) can compute data dependences to determine if a program can be vectorized
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.
Tour on Data Dependences

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.
Data Dependences

S1: \( A = B + D \)
S2: \( C = A + T \)
S3: \( Z = P + T \)
Data Dependences

\[ S1: A = B + D \]
\[ S2: C = A + T \]
\[ S3: Z = P + T \]
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++){
  a[i] = b[i] + 1;
  c[i] = a[i] + 2;
}
```
Dependences in Loops (I)

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

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

S1: $a[1] = b[1] + 1$

Dependences in Loops (I)

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: S1 S1 S1 S1 ...

instances of S2: S2 S2 S2 S2 ...
Dependences in Loops (I)

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

iteration: \hspace{1cm} 0 \hspace{0.5cm} 1 \hspace{0.5cm} 2 \hspace{0.5cm} 3 \hspace{1cm} \ldots

instances of S1: \hspace{1cm} S1 \hspace{0.5cm} S1 \hspace{0.5cm} S1 \hspace{0.5cm} S1 \hspace{1cm} \ldots

instances of S2: \hspace{1cm} S2 \hspace{0.5cm} S2 \hspace{0.5cm} S2 \hspace{0.5cm} S2

\[\rightarrow \text{Loop independent dependence}\]
Dependences in Loops (I)

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

iteration: 0 1 2 3 ...
instances of S1: S1 \rightarrow S2 \rightarrow S1 \rightarrow S2 ...
instances of S2: 

For the whole loop
Dependences in Loops (I)

```c
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: | S1 | S1 | S1 | S1 | ...
| instances of S2: | S2 | S2 | S2 | S2 | ...

For the whole loop
Dependences in Loops (I)

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

iteration: 0 1 2 3 ...

instances of S1: S1 S1 S1 S1 ...

instances of S2: S2 S2 S2 S2 ...

For the whole loop

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

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

Dependences in Loops (II)

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

<table>
<thead>
<tr>
<th>iteration</th>
<th>instances of S1</th>
<th>instances of S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>2</td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>4</td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Dependences in Loops (II)

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

| iteration: | 1 | 2 | 3 | 4 | ...
|------------|---|---|---|---|---
| instances of S1: | S1 | S1 | S1 | S1 | ...
| instances of S2: | S2 | S2 | S2 | S2 | ...

Loop carried dependence
Dependences in Loops (II)

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

iteration: 1 2 3 4 ...

instances of S1: S1 S1 S1 ...
instances of S2: S2 S2 S2 ...

For the whole loop
Dependences in Loops (II)

for (i=1; i<n; i++)
    S1: \( a[i] = b[i] + 1; \)
    S2: \( c[i] = a[i-1] + 2; \)

iteration: 1 2 3 4 ...
instances of S1: S1 S1 S1 S1 ...
instances of S2: S2 S2 S2 S2 ...

For the whole loop

1
Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

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

![Diagram showing dependences between S1 and S2 statements over iterations]
for (i=0; i<n; i++){
  a = b[i] + 1;
  c[i] = a + 2;
}
Dependences in Loops (III)

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

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 (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  S1: a = b[1] + 1  S1: a = b[2] + 1

→ Loop independent dependence
→ Loop carried dependence
Dependences in Loops (III)

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

iteration: 0 1 2 3 ...

instances of S1: S1 S1 S1 S1

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

iteration: 0 1 2 3 ...

instances of S1: S1 S1 S1 S1

instances of S2: S2 S2 S2 S2
Loop Vectorization

• Loop Vectorization is not always a legal transformation.
  – Compilers can vectorize when there are only forward dependences
  – Compilers cannot vectorize when there is a cycle in the data dependences (with the exception of self-antidependence), unless a transformation is applied to remove the cycle
  – Codes with only backward dependences can be vectorized, but need to be transformed
Simple Example

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

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

Diagram:
- **Register File**
  - Register 1 (X1): 32 bits
  - Register 2 (Y1): 32 bits
  - Register 3 (Z1): 32 bits

- **Scalar Unit**
  - lw $t0, 0($a0)
  - lw $t1, 0($a1)
  - add $t3, $t0, $t1
  - sw $t3, 0($a3)

- **Vector Unit**
  - lwv $vt0, 0($a0)
  - lwv $vt1, 0($a1)
  - addv $vt3, $vt0, $vt1
  - swv $vt3, $0($a3)

Diagram shows the processing of a loop with vectorization compared to scalar processing.
Loop Vectorization

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

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

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

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

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

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

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

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

```c
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;
}
```
Acyclic Dependence Graphs: Forward Dependences

```java
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

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

backward dependence
Acyclic Dependendes Graphs
Backward Dependences (I)

Reorder of statements

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

backward
dependence

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

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

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

**Intel Nehalem**

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

**Exec. Time scalar code:** 12.6

**Exec. Time vector code:** --

**Speedup:** --

**Intel Nehalem**

**Compiler report:** Loop was vectorized

**Exec. Time scalar code:** 10.7

**Exec. Time vector code:** 6.2

**Speedup:** 1.72

**Speedup vs non-reordered code:** 2.03
Cycles in the DG (III)

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

Self-antidependence can be vectorized

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

Self true-dependence can not vectorized (as it is)
Cycles in the DG (III)

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

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

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

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

• Overcoming limitations to SIMD-Vectorization
  – Data Dependences
  – Data Alignment
  – Aliasing
  – Non-unit strides
  – Conditional Statements

• 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

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

Is &b[0] 16-byte aligned?

vector load loads b[0] ... b[3]
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.

```c
__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.

```c
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];
    }
```
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

```c
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);  
  }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Core 2 Duo</th>
<th>Intel i7</th>
<th>Power 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligned</td>
<td>0.577</td>
<td>0.580</td>
<td>0.156</td>
</tr>
<tr>
<td>Aligned (unaligned ld)</td>
<td>0.689</td>
<td>0.581</td>
<td>0.241</td>
</tr>
<tr>
<td>Unaligned</td>
<td>2.176</td>
<td>0.629</td>
<td>0.243</td>
</tr>
</tbody>
</table>
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-bytes (notice the 0 in the 4 least significant bits of the address)
- Compiler automatically does padding
Aliasing

• Can the compiler vectorize this loop?

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

• Can the compiler vectorize this loop?

```c
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]
```
Aliasing

• Can the compiler vectorize this loop?

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

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

```c
__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

```c
void func1(float* __restrict__ 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];
}

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);
}
```
Non-unit Stride – Example I

- Array of a struct

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

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

<table>
<thead>
<tr>
<th>point pt[N]</th>
<th>x₀</th>
<th>y₀</th>
<th>z₀</th>
<th>x₁</th>
<th>y₁</th>
<th>z₁</th>
<th>x₂</th>
<th>y₂</th>
<th>z₂</th>
<th>x₃</th>
<th>y₃</th>
<th>z₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pt[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pt[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pt[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Non-unit Stride – Example I

- Array of a struct

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

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

```plaintext
point pt[N] x_0 y_0 z_0  x_1 y_1 z_1  x_2 y_2 z_2  x_3 y_3 z_3
```
Non-unit Stride – Example I

- Array of a struct

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

  ![Diagram of data loading and processing]
Non-unit Stride – Example I

- Array of a struct

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

- Arrays

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

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

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

**Conditional Statements – I**

Intel Nehalem

**Compiler report:** Loop was not vectorized. Condition may protect exception

**Exec. Time scalar code:** 10.4

**Exec. Time vector code:** --

**Speedup:** --

Intel Nehalem

**Compiler report:** Loop was vectorized.

**Exec. Time scalar code:** 10.4

**Exec. Time vector code:** 5.0

**Speedup:** 2.0
Conditional Statements

- Compiler removes *if conditions* when generating vector code

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

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

<table>
<thead>
<tr>
<th>rC</th>
<th>rS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rCmp</th>
<th>rThen</th>
<th>rElse</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>True</td>
<td>3.2</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rCmp</th>
<th>rThen</th>
<th>rElse</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>False</td>
<td>3.2</td>
<td>0.0</td>
</tr>
</tbody>
</table>

// 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

```c
vector bool char = rCmp
vector float r0={0.0,0.0,0.0,0.0};
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

<table>
<thead>
<tr>
<th>Compiler Hints for Intel ICC</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma ivdep</td>
<td>Ignore assume data dependences</td>
</tr>
<tr>
<td>#pragma vector always</td>
<td>override efficiency heuristics</td>
</tr>
<tr>
<td>#pragma novector</td>
<td>disable vectorization</td>
</tr>
<tr>
<td><strong>restrict</strong></td>
<td>assert exclusive access through pointer</td>
</tr>
<tr>
<td><strong>attribute</strong>((aligned(int-val)))</td>
<td>request memory alignment</td>
</tr>
<tr>
<td>memalign(int-val, size);</td>
<td>malloc aligned memory</td>
</tr>
<tr>
<td>__assume_aligned(exp, int-val)</td>
<td>assert alignment property</td>
</tr>
</tbody>
</table>
Compiler Directives

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

<table>
<thead>
<tr>
<th>Compiler Hints for IBM XLC</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma ibm independent_loop</td>
<td>Ignore assumed data dependences</td>
</tr>
<tr>
<td>#pragma nosimd</td>
<td>disable vectorization</td>
</tr>
<tr>
<td><strong>restrict</strong></td>
<td>assert exclusive access through pointer</td>
</tr>
<tr>
<td><strong>attribute</strong>((aligned(int-val)))</td>
<td>request memory alignment</td>
</tr>
<tr>
<td>memalign(int-val, size);</td>
<td>malloc aligned memory</td>
</tr>
<tr>
<td>__alignx (int-val, exp)</td>
<td>assert alignment property</td>
</tr>
</tbody>
</table>
Outline

• Overcoming limitations to SIMD-Vectorization
  – Data Dependences
  – Data Alignment
  – Aliasing
  – Non-unit strides
  – Conditional Statements

• 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:
  
  ```c
  #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.
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

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

- Intrinsic operate on these types and have the format:
  \_mm_instruction_suffix(…)

- Suffix can take many forms. Among them:
  - ss scalar single precision
  - ps packed (vector) single precision
  - sd scalar double precision
  - pd packed double precision
  - si\# scalar integer (8, 16, 32, 64, 128 bits)
  - su\# scalar unsigned integer (8, 16, 32, 64, 128 bits)
Intel SSE intrinsics
Instructions – Examples

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

```c
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:

```c
__m128 a, b;
__m128 c = _mm_add_ps(a, b);
```
#include <xmmintrin.h>
#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);
    }
}
#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];
    } 
}

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

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

```c
#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

```c
#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);
    }
}
```
#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];
    }
}

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

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

```c
going (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;
}
```

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

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

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

Which code runs faster?

Why?

#include <xmmintrin.h>
#define n 1000

int main() {
    __m128 rA1, rA2, rB, rC, rD;
    __m128 r5=_mm_set1_ps((float)0.5)
    for (i = 0; i < LEN-4; i+=4) {
        rA2= _mm_loadu_ps(&a[i+1]);
        rB= _mm_load_ps(&b[i]);
        rC= _mm_load_ps(&c[i]);
        rA1= _mm_add_ps(rB, rC);
        rD= _mm_mul_ps(_mm_add_ps(rA1,rA2),r5);
        _mm_store_ps(&a[i], rA1);
        _mm_store_ps(&d[i], rD);
    }
}
Node Splitting with intrinsics

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;
}

#include <xmmintrin.h>
define n 1000

int main() {
    __m128 rA1, rA2, rB, rC, rD;
    __m128 r5=_mm_set1_ps((float)0.5)
    for (i = 0; i < LEN-4; i+=4) { 
        rA2= _mm_loadu_ps(&a[i+1]);
        rB= _mm_load_ps(&b[i]);
        rC= _mm_load_ps(&c[i]);
        rA1= _mm_add_ps(rB, rC);
        rD= _mm_mul_ps(_mm_add_ps(rA1,rA2),r5);
        _mm_store_ps(&a[i], rA1);
        _mm_store_ps(&d[i], rD);
    }
}
Node Splitting with intrinsics

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

Intel Nehalem
Compiler report: Loop was vectorized.
Exec. Time scalar code: 13.2
Exec. Time vector code: 9.7
Speedup: 1.3

Intel Nehalem
Exec. Time intrinsics: 6.1
Speedup (versus vector code): 1.6
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

  – 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.
Data Dependences

• The correctness of 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.
Compiler Optimizations

For a longer discussion see:


 Algorithms


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

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

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

```c
    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);
```
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
- \textit{vec-report0}, no report is generated
- \textit{vec-report1}, indicates the line number of the loops that were vectorized
- \textit{vec-report2 .. 5}, gives a more detailed report that includes the loops that were not vectorized and the reason for that.
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
Compiler Directives (I)

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

#pragma ivdep (ICC compiler)

#pragma ibm independent_loop (XLC compiler)
Compiler Directives (I)

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

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

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

K = 1

Can be vectorized

```c
a[0]=a[1]+b[0]
```

K = -1

Cannot be vectorized

```c
a[1]=a[0]+b[0]
```
Compiler Directives (I)

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

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

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

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

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

Wrong results will be obtained if loop is vectorized when $-3 < k < 0$
Compiler Directives (I)

S124

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

S124_1

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

S124_2

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

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence
Exec. Time scalar code: 6.0
Exec. Time vector code: --
Speedup: --

Intel Nehalem

Compiler report: Loop was vectorized
Exec. Time scalar code: 6.0
Exec. Time vector code: 2.4
Speedup: 2.5
Compiler Directives (I)

IBM Power 7

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

Exec. Time scalar code: 2.2
Exec. Time vector code: --
Speedup: --

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

This transformation improves locality and is usually combined with vectorization
Strip Mining

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

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

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];
}

for (i=1; i<LEN; i++)
a[i] = b[i];
for (i=1; i<LEN; i++)
c[i] = c[i-1] + a[i];
int v[N];
...
for (int i=0; i<N; i++){
    Transform (v[i]);
}
for (int i=0; i<N; i++){
    Light (v[i]);
}
Compiler Directives (I)

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

  #pragma ivdep (ICC compiler)

  #pragma ibm independent_loop (XLC compiler)
Compiler Directives (II)

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

```c
#pragma novector (ICC compiler)
#pragma nosimd (XLC compiler)
```
Vector code can run slower than scalar code

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];
}

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

Less locality when executing in vector mode
S116

```c
#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];
}
```

Intel Nehalem

**Compiler report:** Loop was partially vectorized

- **Exec. Time scalar code:** 14.7
- **Exec. Time vector code:** 18.1
- **Speedup:** 0.8