Program Optimization Through Loop Vectorization

María Garzarán, Saeed Maleki
William Gropp and David Padua

Department of Computer Science
University of Illinois at Urbana-Champaign

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

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

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
### 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];
```

### 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

**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
  2. Macros or Vector Intrinsics
  3. Assembly Language
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?

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

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

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

The examples use single precision floating point numbers.

Compiler directives

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

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

Compiler directives

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

```c
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;
}
```

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

Intrinsics (SSE)
```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++) {
        c[i] = a[i] * b[i];
    }
}
```
Intrinsics (Altivec)

```c
#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];
```

```c
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

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++) {
    a[i] = b[i] + 1;
    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++) {
    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)

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

Loop independent dependence

For the whole loop
Dependences in Loops (I)

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

```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:...
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”

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

- Loop carried dependence

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"

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

For the whole loop

Dependences in Loops (III)

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

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

<table>
<thead>
<tr>
<th>iteration</th>
<th>i=0</th>
<th>i=1</th>
<th>i=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: a = b[0] + 1</td>
<td>S1: a = b[1] + 1</td>
<td>S1: a = b[2] + 1</td>
<td></td>
</tr>
</tbody>
</table>
Dependences in Loops (III)

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

\[
\text{iteration: } 0 \quad 1 \quad 2 \quad 3 \quad \ldots
\]

\[
\text{instances of } S1:
\]

\[
\text{instances of } S2:
\]

Dependences in Loops (IV)

- Doubly nested loops

\[
\text{for } (i = 1; i < n; i++) \{ \\
\quad \text{for } (j = 1; j < n; j++) \{ \\
\quad \quad S1: a[i][j] = a[i][j-1] + a[i-1][j]; \\
\quad \quad S2: c[i][j] = c[i-1][j] + c[i][j-1]; \\
\quad \}
\}
\]
```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
```

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

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

**Loop carried dependences**

**Dependences in Loops (IV)**
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.

\[
\text{for } (i=0; i<n; i++) \{
\begin{align*}
&C\text{ code example here}
\end{align*}
\}
\]

Data dependences and transformations

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

Distributing

\[
\text{for } (i=1; i<n; i++) \{
\begin{align*}
&C\text{ code example here}
\end{align*}
\}
\]

Removing dependences

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

Freezing Loops

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

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 ($S += A[i]$)
  - Boolean recurrences ($i \land (A[i] > max) \land max = A[i]$)
Stripmining

- Stripmining is a simple transformation.

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

- It is typically used to improve locality.

Stripmining (cont.)

- Stripmining is often used when vectorizing

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

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

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

**Dependence Graphs and Compiler Vectorization**

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

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

This loop cannot be vectorized as it is

Acyclic Dependence Graphs: Backward Dependences (I)

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

This loop cannot be vectorized as it is

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

Reorder of statements

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

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

This loop cannot be vectorized as it is

The IBM XL compiler generated the same code in both cases

The IBM XL compiler generated the same code in both cases

This loop cannot be vectorized as it is
for (int i=0; i<LEN-1; i++) {
    b[i] = a[i] + (float) 1.0;
    a[i+1] = b[i] + (float) 2.0;
}

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

Compiler report: Loop was not vectorized.
Existence of vector dependence
Exec. Time scalar code: 12.1
Exec. Time vector code: --
Speedup: --
The IBM XLC compiler applies forward substitution and reordering to vectorize the code. This loop is not vectorized:

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

This loop is vectorized:

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

The compiler needs to do this to vectorize the code as before:

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

Will the IBM XLC compiler vectorize this code as before?

No, the compiler does not vectorize S215 because it is not cost-effective.
Cycles in the DG (II)

A loop can be partially vectorized

```c
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)

Cycles in the DG (III)

Self-antidependence can be vectorized

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

Self true-dependence cannot be vectorized (as it is)

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

IBM Power 7
Compiler report: Loop was not vectorized. Existence of vector dependence prevents SIMD vectorization
Exec. Time scalar code: --
Exec. Time vector code: --
Speedup: --
for (int i=0; i<LEN-1; i++) {
  a[i] = a[i+1] + b[i];
}

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

---

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 2.0
Exec. Time vector code: 1.0
Speedup: 2.0

---

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

---

Cycles in the DG (III)

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

---

Cycles in the DG (IV)

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

Self true-dependence cannot be vectorized

---

Cycles in the DG (IV)

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

Yes, it can be vectorized because the dependence distance is 4, which is the number of iterations that the SIMD unit can execute simultaneously.
Cycles in the DG (V)

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

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

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

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 (V)

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

Can this loop be vectorized?

```
S1
```

Cycles in the DG (V)

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

Can this loop be vectorized?

```
S1
```

Cycles in the DG (V)

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

Can this loop be vectorized?

```
S1
```

Cycles in the DG (V)

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

Can this loop be vectorized?

```
S1
```

Cycles in the DG (V)

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

Can this loop be vectorized?

```
S1
```

Cycles in the DG (V)

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

Can this loop be vectorized?

```
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.
  ```c
  for (int i=0;i<LEN;i++)
  
  \( a[r[i]] = a[r[i]] \times (\text{float})2.0; \)
  ```

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

---

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 compiler
  - Other examples
Loop Transformations

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

Compiler Directives (I)

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

  #pragma ivdep (ICC compiler)
  #pragma ibm independent_loop (XLC compiler)

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

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

How can the programmer tell the compiler that k >= 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 >= 0.
- Programmer knows that k >= 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

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

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

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

Compiler Directives (II)

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

```c
#pragma novector (ICC compiler)
#pragma nosimd (XLC compiler)
```
Compiler Directives (II)

Vector code can run slower than scalar code

```
for (int i=1;i<LEN;i++)
{  
  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

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);
}
```

- 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

for (i=0; i<LEN; i++) {
    a[i] = (float)sqrt(b[i]) + (float)sqrt(c[i]);
    dummy(a,b,c);
}

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

Reordering Statements

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

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

Reordering Statements

for (i=0; i<LEN; i++) {
    a[i] = (float)sqrt(b[i]) + (float)sqrt(c[i]);
    dummy(a,b,c);
}

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

Reordering Statements

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

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

The IBM XLC compiler generated the same code in both cases
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;

Node Splitting

for (int i=0; i<LEN-1; i++)

S1 temp[i] = a[i+1];
S2 a[i] = b[i] + c[i];
S3 d[i] = (a[i] + temp[i]) * (float)0.5;

Scalar Expansion

for (int i=0; i<n; i++)

S1 t = a[i];
S2 a[i] = b[i];
S3 b[i] = t;

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 3.8
Exec. Time vector code: 1.7
Speedup: 2.2

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

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: 3.8
Exec. Time vector code: 1.7
Speedup: 2.2

Intel Nehalem
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 5.1
Exec. Time vector code: 2.4
Speedup: 2.0

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 3.8
Exec. Time vector code: 1.7
Speedup: 2.2

Intel Nehalem
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 3.8
Exec. Time vector code: 1.7
Speedup: 2.2
Scalar Expansion

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

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];

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.

if (k>0)
    for (i=0; i<LEN; i++)
        A[i] = B[i] + C[i];

Compiler report: Loop was vectorized.
Exec. Time scalar code: 0.28
Exec. Time vector code: 0.14
Speedup: 2.0

Compiler report: Loop was SIMD vectorized.
Exec. Time scalar code: 0.28
Exec. Time vector code: 0.14
Speedup: 2
Loop Peeling

\[
\begin{align*}
\text{for (int } i=0; i<\text{LEN}; i++) & \quad a[0] = a[0] + a[0] ; \\
S1 & \quad a[i] = a[i] + a[0] ; \\
S1 & \quad a[0] = a[0] + a[0] ; \\
\end{align*}
\]

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

Self true-dependence is not vectorized.

Loop Interchanging

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

\[
\begin{align*}
\text{for (int } i=0; i<\text{LEN}; i++) & \quad a[0] = a[0] + a[0] ; \\
S1 & \quad a[i] = a[i] + a[0] ; \\
S1 & \quad a[0] = a[0] + a[0] ; \\
\end{align*}
\]

IBM Power 7

Compiler report: Loop was not SIMD vectorized
Time scalar code: 2.4
Time vector code: 1.2
Speedup: 2.04

Intel Nehalem

Compiler report: Loop was vectorized.
Exec. Time scalar code: 6.7
Exec. Time vector code: 1.2
Speedup: 5.2

Loop Interchanging

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

Inner loop cannot be vectorized because of self-dependence

---

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

Loop interchange is legal
No dependences in inner loop

---

Intel Nehalem
Compiler report: Loop was not vectorized.
Exec. Time scalar code: 0.2
Exec. Time vector code: 0.2
Speedup: 3

---

Intel Nehalem
Compiler report: Loop was vectorized.
Exec. Time scalar code: 0.6
Exec. Time vector code: 0.2
Speedup: 3
Loop Interchanging

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

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. Time scalar code: 0.5
Exec. Time vector code: --
Speedup: --

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;
  }
  ```

Summary

- Reductions
  - Sum Reduction
  - Max Loc Reduction

Intel Nehalem
Compiler report: Loop was vectorized.
Exec. Time scalar code: 9.6
Exec. Time vector code: 2.4
Speedup: 3.9
Reductions

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

```java
float s = (float)0.0;
for (int i=0;i<LEN;i++)
    s += (float)2.0;
```

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 1.1
Exec. Time vector code: 0.4
Speedup: 2.4

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. Time scalar code: 10.2
Exec. Time vector code: --
Speedup: --

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

```c
float s = (float)0.0;
for (int i=0;i<LEN;i++)
    s += (float)2.0;
    a[i] = (float)2.*(i+1)*b[i];
    a[i] = s * b[i];
```
Induction variables

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

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

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

Intel Nehalem
Compiler report: Loop was vectorized.
Exec. Time scalar code: 6.1
Exec. Time vector code: 1.9
Speedup: 3.1

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 2.7
Exec. Time vector code: 1.4
Speedup: 2.6

Induction Variables

• Coding style matters:

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

These codes are equivalent, but …
Induction Variables

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

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

```c
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

```c
void test2(float *a, float *b, float *c) {
    for (int i = 0; i < LEN; i++)
        a[i] = b[i] + c[i];
}
```
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.

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

```c
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).

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 func(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

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

```c
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(){  
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(){  
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(){  
rA = _mm_loadu_ps(&A[i]);  
rB = _mm_loadu_ps(&B[i]);  
rC = _mm_add_ps(rA, rB);  
_mm_storeu_ps(&C[i], rC);  }

Alignment in a struct

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

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

Output:
0x7ffe6765f00, 0x7ffe6765f04, 0x7ffe6766004, 0x7ffe6766008  

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

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

Nanosecond per iteration

<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)</td>
<td>0.689</td>
<td>0.581</td>
<td>0.241</td>
</tr>
<tr>
<td>Unaligned</td>
<td>2.176</td>
<td>0.829</td>
<td>0.243</td>
</tr>
</tbody>
</table>

• 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
Aliasing

• Can the compiler vectorize this loop?

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

float* a = &b[1];
...
void func(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 func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(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
void func(float *a, float *b, float *c)
    { for (int i = 0; i < LEN; i++)
        { a[i] = b[i] + c[i];
        } }
```
Aliasing

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

### Aliasing

1. **__restrict__** keyword

```c
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];
}
```

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

### Aliasing – Multidimensional arrays

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

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

```c
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.

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

__restrict__ only qualifies the first dereferencing of c;
Nothing is said about the arrays that can be accessed through c[i]

Intel ICC compiler, version 11.1 will vectorize this code.
Previous versions of the Intel compiler or compilers from other vendors, such as IBM XLC, will not vectorize it.

Aliasing – Multidimensional arrays

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

```c
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

• 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

```c
__attribute__ ((aligned(16))) float a[N][N];
void t()

    a[i][j]...

int main()

    t();
```
Aliasing – Multidimensional arrays

2. Linearize the arrays

```c
void t(float* __restrict__ A) {
    // Access to Element A[i][j] is now A[i*128+j]
    
} 
```

```c
int main() {
    float* A = (float*) memalign(16, 128*128*sizeof(float));
    
    t(A);
}
```

3. Use compiler directives:

```c
#pragma ivdep (Intel ICC)
#pragma disjoint(IBM XL)
```

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

Outline

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

Non-unit Stride – Example I

- Array of a struct

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

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

```c
point pt[N] x0 x1 x2 x3 y0 y1 y2 y3 z0 z1 z2 z3
```

```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;
  }
  ```

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

**Intel Nehalem**

Compiler report: Loop was not vectorized. Vectorization possible but seems inefficient.
Exec. Time scalar code: 6.8
Exec. Time vector code: 1.3
Speedup: --
Non-unit Stride – Example I

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

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. Time vector code: 2.9
Exec. Time scalar code: 3.7
Speedup: --

Intel Nehalem
Compiler report: Permuted loop was vectorized.
Exec. Time vector code: 2.8
Exec. Time scalar code: 2.9
Speedup: --

Non-unit Stride – Example II

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

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time vector code: 1.8
Exec. Time scalar code: 1.9
Speedup: 2.0

Intel Nehalem
Compiler report: Permuted loop was vectorized.
Exec. Time vector code: 0.4
Exec. Time scalar code: 0.2
Speedup: 2.0

Non-unit Stride – Example II

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

IBM Power 7
Compiler report: Loop was not SIMD vectorized
Exec. Time vector code: 0.4
Exec. Time scalar code: 0.4
Speedup: 0.4

Intel Nehalem
Compiler report: Permuted loop was vectorized.
Exec. Time vector code: 0.4
Exec. Time scalar code: 0.2
Speedup: 2.0
Outline

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

Conditional Statements – I

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

```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];
```

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 not vectorized.
- Exec. Time scalar code: 10.4
- Exec. Time vector code: --
- Speedup: --

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

Conditional Statements – I

```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];
```

Compiler report: Loop was SIMD vectorized.
- Exec. Time scalar code: 4.0
- Exec. Time vector code: 1.5
- Speedup: 2.5
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];

Compilier 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.

<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

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

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

Intel SSE intrinsic Instructions

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

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

Intel SSE intrinsics Instructions – Examples

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

Intrinsics (SSE)

```c
#include <xmmintrin.h>
#define n 1024
_attribate__((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
#include <xmmintrin.h>
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];
    }
}
```

```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);
    }
}
```

---

## Node Splitting

```c
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
}
```

```c
for (int i=0; i<LEN-1; i++) {
    S1 S2 = a[i]+c[i]*float 0.5;
    S0 S3 = a[i]+temp[i]*float 0.5
}
```
Node Splitting with intrinsics

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

Which code runs faster?
Why?
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.

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.

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

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

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

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.
Compiling

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

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

Strip Mining

This transformation improves locality and is usually combined with vectorization

Strip Mining

This transformation improves locality and is usually combined with vectorization

Strip Mining

Strip Mining

Strip Mining

Loop Distribution

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

- Another example

```c
int v[N];
for (int i=0;i<N;i++) {
    Transform (v[i]);
    Light (v[i]);
}
```
### Overview Floating-Point Vector ISAs

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Name</th>
<th>n-ways</th>
<th>Precision</th>
<th>Introduced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>SSE</td>
<td>4-way</td>
<td>single</td>
<td>Pentium III</td>
</tr>
<tr>
<td></td>
<td>SSE2</td>
<td>+ 2-way</td>
<td>double</td>
<td>Pentium 4( Prescott )</td>
</tr>
<tr>
<td></td>
<td>SSE3</td>
<td></td>
<td></td>
<td>Core Duo</td>
</tr>
<tr>
<td></td>
<td>SSE4</td>
<td>+ 4-way</td>
<td>double</td>
<td>Sandy Bridge, 2011</td>
</tr>
<tr>
<td>Intel</td>
<td>IPF</td>
<td>2-way</td>
<td>single</td>
<td>Itanium</td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow!</td>
<td>2-way</td>
<td>single</td>
<td>K6</td>
</tr>
<tr>
<td></td>
<td>Enhanced 3DNow!</td>
<td>4-way</td>
<td>single</td>
<td>K7</td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow! Professional</td>
<td>+ 4-way</td>
<td>double</td>
<td>AthlonXP</td>
</tr>
<tr>
<td>Motorola</td>
<td>AVX</td>
<td>4-way</td>
<td>single</td>
<td>MPC 7400 G4</td>
</tr>
<tr>
<td>IBM</td>
<td>VMX</td>
<td>4-way</td>
<td>single</td>
<td>PowerPC 970 G5</td>
</tr>
<tr>
<td></td>
<td>SPU</td>
<td>+ 2-way</td>
<td>double</td>
<td>Cell BE</td>
</tr>
<tr>
<td></td>
<td>Double FPU</td>
<td>2-way</td>
<td>double</td>
<td>PowerPC 440 FP2</td>
</tr>
</tbody>
</table>

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

Overview Floating-Point Vector ISAs

Based on slide provided by Markus Püschel

### Evolution of Intel Vector Instructions

- **MMX (1996, Pentium)**
  - CPU-based MPEG decoding
  - Integers only, 64bit 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 SSE single-precision

- **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


### Run-Time Symbolic Resolution

If \( t > 0 \) → self true dependence

\[
\begin{align*}
\text{S1} & : a[i+t] = a[i] + b[i] ; \\
\text{S2} & : b[i+t] = b[i] + c[i] ; \\
\text{S3} & : d[i] = a[i] + (\text{float})1.0 ;
\end{align*}
\]

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

\[
\begin{align*}
\text{S1} & : a[i+t] = a[i] + b[i] ; \\
\text{S2} & : b[i+t] = b[i] + c[i] ; \\
\text{S3} & : d[i] = a[i] + (\text{float})1.0 ;
\end{align*}
\]

If \( t = 0 \) → Cannot be vectorized

\[
\begin{align*}
\text{S1} & : a[i+t] = a[i] + b[i] ; \\
\text{S2} & : b[i+t] = b[i] + c[i] ; \\
\text{S3} & : d[i] = a[i] + (\text{float})1.0 ;
\end{align*}
\]

Can be vectorized

\[
\begin{align*}
\text{S1} & : a[i+t] = a[i] + b[i] ; \\
\text{S2} & : b[i+t] = b[i] + c[i] ; \\
\text{S3} & : d[i] = a[i] + (\text{float})1.0 ;
\end{align*}
\]

### Loop Vectorization – Example I

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

The Intel ICC compiler generated the same code in both cases

### Loop Vectorization – Example II

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

The Intel ICC compiler generated the same code in both cases

### 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

- **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

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

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

How do we access the SIMD units?

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

```c
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
  2. Macros or Vector Intrinsics
  3. Assembly Language

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

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

<table>
<thead>
<tr>
<th>Total</th>
<th>159</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Vectorized</td>
<td>74</td>
</tr>
<tr>
<td>Not Vectorized</td>
<td>85</td>
</tr>
</tbody>
</table>

Vectorizable by

| Classic Transformation | 35 |
| New Transformation | 6 |
| Manual Vectorization | 19 |

Impossible to Vectorize

| Non-unit Stride Access | 16 |
| Data Dependence | 5 |
| Other | 4 |

Loops Percentage

- Vectorized | 84.3%
- Classic transformation applied | 46.5%
- New transformation applied | 22.0%
- Manual vector code | 11.9%
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

What are the speedups?

• Speedups obtained by XLC compiler

<table>
<thead>
<tr>
<th>Test Suite Collection</th>
<th>Average Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatically by XLC</td>
<td>1.73</td>
</tr>
<tr>
<td>By adding classic transformations</td>
<td>3.48</td>
</tr>
<tr>
<td>By adding new transformations</td>
<td>3.64</td>
</tr>
<tr>
<td>By adding manual vectorization</td>
<td>3.78</td>
</tr>
</tbody>
</table>

Why did the compilers fail to vectorize?

<table>
<thead>
<tr>
<th>Issues</th>
<th>ICC</th>
<th>XLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorizable but not automatic</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>Cyclic data dependence</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Non-Unit stride access</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Conditional statement</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Aliasing</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Acyclic data dependence</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Reduction</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Unsupported loop structure</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Loop interchange</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Wrap around</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Scalar expansion</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Other</td>
<td>13</td>
<td>12</td>
</tr>
</tbody>
</table>

XLC and ICC Comparison

<table>
<thead>
<tr>
<th>Issues</th>
<th>XLC but not ICC</th>
<th>ICC but not XLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorized</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>Aliasing</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Acyclic data dependence</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Unrolled loop</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Loop interchange</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Conditional statements</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Unsupported loop structure</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Wrap around variable</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Reduction</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Non-unit stride access</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Other</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Another example: matrix-matrix multiplication

```c
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: --

Another example: matrix-matrix multiplication

```c
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 k = 0; k < 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: --

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
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”

Unrolled loop

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

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

Unrolled loop

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

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

Loop independent: dependence is within the loop iteration boundaries

Dependences in Loops

- A slightly more complex example

Unrolled loop

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

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

Unrolled loop

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

Unrolled loop

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

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

Unrolled loop

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

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

Unrolled loop

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

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

Unrolled loop

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

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

Unrolled loop

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

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

Unrolled loop

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

iteration: 1 2 3 4 ...
instances of S1: S1 S1 S1 ...
instances of S2: S2 S2 S2 ...
### Unrolled loop

For (i=1; i<LEN; i++)

\[ S1: a[i] = b[i] + 1; \]
\[ S2: c[i] = a[i-1] + 2; \]

#### Even more complex

For (i=0; i<LEN; i++)

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

#### Two dimensional

For (i=1; i<LEN; i++)

\[ S1: a[i][j] = a[i-1][j] + a[i][j-1]; \]
\[ S2: c[i][j] = a[i][j] + 2; \]

### Dependences in Loops

- A slightly more complex example

Unrolled loop

for (i=1; i<LEN; i++)

\[ S1: a[i] = b[i] + 1; \]
\[ S2: c[i] = a[i-1] + 2; \]

#### Even more complex

Unrolled loop

for (i=0; i<LEN; i++)

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

#### Loop Independent

Unrolled loop

for (i=1; i<LEN; i++)

\[ S1: a[i][j] = a[i-1][j] + a[i][j-1]; \]
\[ S2: c[i][j] = a[i][j] + 2; \]
Dependences in Loops

• Two dimensional

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

Unrolled loop

\[ a[1][1] = a[1][0] + a[0][2] \]
\[ a[1][2] = a[1][1] + a[0][3] \]
\[ a[1][3] = a[1][2] + a[1][4] \]
\[ a[1][4] = a[1][3] + a[1][5] \]
\[ a[2][1] = a[2][0] + a[1][2] \]
\[ a[2][2] = a[2][1] + a[1][3] \]

Loop carried dependence

Dependences in Loops

• Another two dimensional loop

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

Unrolled loop

\[ a[1][1] = a[1][0] + a[0][2] \]
\[ a[1][2] = a[1][1] + a[0][3] \]
\[ a[1][3] = a[1][2] + a[1][4] \]
\[ a[1][4] = a[1][3] + a[1][5] \]
\[ a[2][1] = a[2][0] + a[1][2] \]
\[ a[2][2] = a[2][1] + a[1][3] \]

Loop carried dependence

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.

**Scalar Code**

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

**Scalar Code, loops are distributed**

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

**Scalar Code, loops are strip-mined**

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

**Loop Vectorization**

- Loop Vectorization

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

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

  - Loop distribution will increase the cache miss ratio if array b is large.

  - Fused loop was strip-mined (equivalent to the strip-mined version) in all the cases.

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

  - Loop was strip-mined and distributed.

  - Compiler report:

    - Speedup: 2.7
    - Exec. Time vector code: 1.4
    - Exec. Time scalar code: 2.0
    - SIMD vectorized scalar code: 2.9
    - SIMD vectorized vector code: 1.2
    - SIMD vectorized speedup: 2.07
Loop Vectorization

• 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.

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

I
A loop can be partially vectorized

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
(A loop with a cycle in the dependence graph cannot be vectorized)

The INTEL ICC compiler generates the same code in both cases
Cycles in the DG – Example V

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

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

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

Yes, i = i' + 4

---

Loop Interchanging

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

It is illegal
Loop Interchanging

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

Intel Nehalem
Compiler report: Loop was not vectorized. Vectorization possible, but inefficient
Exec. Time scalar code: 2.2
Exec. Time vector code: --
Speedup: --

IBM Power 7
Compiler report: Loop was not vectorized. Vectorization possible, but inefficient
Exec. Time scalar code: 1.2
Exec. Time vector code: --
Speedup: --

Loop Interchanging

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

Intel Nehalem
Compiler report: Loop was not vectorized. Vectorization possible, but inefficient
Exec. Time scalar code: 2.2
Exec. Time vector code: --
Speedup: --

IBM Power 7
Compiler report: Loop was not vectorized because it is not profitable to vectorize
Exec. Time scalar code: 1.2
Exec. Time vector code: --
Speedup: --

The innermost loop carries a true dependence with itself

Loop Interchanging

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

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

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

It is legal

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

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

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

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

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

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:
  __m256d x, y;
  __m256d z = _mm_add_pd(x, y);

Intel SSE intrinsics
Instructions – Examples III

• Add two vectors of 8 16-bit signed integers using saturation arithmetic (*):
  __m256i r, s;
  __m256i t = _mm_adds_epi16(r, s);

• Compare two vectors of 16 8-bit signed integers
  __m256i a, b;
  __m256i 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)
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 in order to generate parallel code.

DAVID A. PADUA and MICHAEL J. WOLFE

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

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]);
}

Options:
1) Remove the condition.
2) Leave it as is.

Conditional Statements-II

- 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

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

IBM Power 7
Compiler report: Loop was SIMD vectorized
Exec. Time scalar code: 2.1
Exec. Time vector code: 2.06
Speedup: 0.8