Pick up ICES, feel free to leave off gender.

Program Optimization Through Loop Vectorization

Exam 6 tonight
Sheet of notes back?
Name Tuesday 2pm
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

```c
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 xlc compiler, version 11.0
OS Red Hat Linux Enterprise 5.4

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

```
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
  2. Macros or Vector Intrinsics
  3. C code and a vectorizing compiler

```c
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);
    }
}
```

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

```c
for (i=0; i<LEN; i++)
    c[i] = a[i] + b[i];
```
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>
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></td>
<td></td>
<td>159</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>

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++) {
    }
}
```

Can this code be vectorized?

a) Yes  
b) No
void test(float* A, float* B, float* C, float* D, float* E) {
    for (int i = 0; i < LEN; i++) {
    }
}

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 not vectorized.
**Exec. Time scalar code:** 5.6
**Exec. Time vector code:** --
**Speedup:** --

**Intel Nehalem**

**Compiler report:** Loop was vectorized.
**Exec. Time scalar code:** 5.6
**Exec. Time vector code:** 2.2
**Speedup:** 2.5
void test(float* A, float* B, float* C, float* D, float* E) {
    for (int i = 0; i < LEN; i++) {
    }
}

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 not vectorized.

**Exec. Time scalar code:** 2.3

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

**Speedup:** --

**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 of 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: $Z = P + T$

S1: $A = B + D$
S3: $Z = P + T$
S2: $C = A + T$

a) Safe
b) Unsafe
Dependences in Loops (I)

- Dependences in loops are easy to understand if the 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;
}
```
Dependences in Loops (I)

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


a) RAW
b) WAR
c) WAW
d) RAW and WAR
e) All of the above
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)

```cpp
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
Dependences in Loops (I)

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

For the whole loop
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 ...

For the whole loop

0
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 ...

For the whole loop
distance 0
Dependences in Loops (I)

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

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

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

S1
S2

a) RAW
b) WAR
c) WAW
d) RAW and WAR
e) All of the above
Dependences in Loops (II)

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

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


a) RAW
b) WAR
c) WAW
d) RAW and WAR
e) All of the above
Dependences in Loops (II)

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

iteration: \[1\] \[2\] \[3\] \[4\] \[\ldots\]

instances of S1: \[S1\] \[S1\] \[S1\] \[S1\] \[\ldots\]

instances of S2: \[S2\] \[S2\] \[S2\] \[S2\]
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 ...

Loop carried dependence
Dependences in Loops (II)

```
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
- ...

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

For the whole loop
Dependences in Loops (III)

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

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

i=0
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

a) RAW
b) WAR
c) WAW
d) RAW and WAR
e) All of the above
Dependences in Loops (III)

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

- **Loop independent dependence**
- **Loop carried dependence**
Dependences in Loops (III)

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

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++){  
    a = b[i] + 1;  
    c[i] = a + 2;  
}
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];
```

```
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)
```

Scalar Unit

Vector Unit

Register File

Vector Register File
Loop Vectorization

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

\[
\begin{align*}
\text{for (i=0; i<LEN; i++)} \{ \\
S1 & \quad a[i]=b[i]+(\text{float})1.0; \\
S2 & \quad c[i]=b[i]+(\text{float})2.0; \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{for (i=0; i<LEN; i+=\text{strip\_size})} \{ \\
\text{for (j=i; j<i+\text{strip\_size}; j++)} \{ \\
& \quad a[j]=b[j]+(\text{float})1.0; \\
& \quad c[j]=b[j]+(\text{float})2.0; \\
\}
\}
\end{align*}
\]
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;
}
```
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

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

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

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

a) RAW
b) WAR
c) WAW
d) RAW and WAR
e) All of the above
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 Dependendenden Graphs
Backward Dependences (I)

Reorder of statements

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

backward dependence

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

forward dependence
Acyclic Dependendent Graphs
Backward Dependences (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: --

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)

Self-antidependence can be vectorized

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

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];
}

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

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