Rechnerarchitektur (RA)

Sommersemester 2016

Architecture-Aware Optimizations
-Software Optimizations-

Jian-Jia Chen
Informatik 12
Jian-jia.chen@tu-..
Tel.: 0231 755 6078

2016/07/13
High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding
Fixed-Point Data Format

• Floating-Point vs. Fixed-Point
  - \textit{exponent}, mantissa
  - Floating-Point
    • automatic computation and update of each exponent at run-time
  - Fixed-Point
    • implicit exponent
    • determined off-line

• Integer vs. Fixed-Point

(a) Integer
\begin{equation}
\begin{array}{cccccccc}
S & 1 & 0 & 0 & . . . & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\end{equation}
\text{IWL}=3 \quad \text{FWL}

(b) Fixed-Point
\begin{equation}
\begin{array}{cccccccc}
S & 1 & 0 & 0 & . . . & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\end{equation}

hypothesetical binary point

\text© Ki-Il Kum, et al
Floating-point to fixed point conversion

Pros:
- Lower cost
- Faster
- Lower power consumption
- Sufficient SQNR, *if properly scaled*
- Suitable for portable applications

Cons:
- Decreased dynamic range
- Finite word-length effect, *unless properly scaled*
  - Overflow and excessive quantization noise
- Extra programming effort

© Ki-Il Kum, et al. (Seoul National University): A Floating-point To Fixed-point C Converter For Fixed-point Digital Signal Processors, 2nd SUIF Workshop, 1996
Development Procedure

1. Floating-Point C Program
2. Range Estimator
3. Execution
4. Manual specification
   - Floating-Point to Fixed-Point C Program Converter
   - Range Estimation C Program
   - IWL information

© Ki-II Kum, et al
Performance Comparison - Machine Cycles -

Cycles

ADPCM

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-Point (16b)</td>
<td>26718</td>
</tr>
<tr>
<td>Fixed-Point (32b)</td>
<td>61401</td>
</tr>
<tr>
<td>Floating-Point</td>
<td>125249</td>
</tr>
</tbody>
</table>

© Ki-Il Kum, et al
Performance Comparison - SNR -

ADPCM

SNR (dB)

<table>
<thead>
<tr>
<th></th>
<th>Fixed-Point (16b)</th>
<th>Fixed-Point (32b)</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Ki-Il Kum, et al
High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding
Impact of memory allocation on efficiency

Array $p[j][k]$

Row major order (C)

<table>
<thead>
<tr>
<th></th>
<th>k=0</th>
<th>k=1</th>
<th>k=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=1</td>
<td>k=0</td>
<td>k=1</td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>k=0</td>
<td>k=1</td>
<td>...</td>
</tr>
</tbody>
</table>

Column major order (FORTRAN)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>k=0</th>
<th>k=1</th>
<th>k=2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j=0</td>
<td>j=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>j=0</td>
<td>j=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>j=0</td>
<td>j=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Best performance of innermost loop corresponds to rightmost array index

Two loops, assuming row major order (C):

\[
\begin{align*}
\text{for } (k=0; k \leq m; k++) & \quad \text{for } (j=0; j \leq n; j++) \\
\text{for } (j=0; j \leq n; j++) & \quad \text{for } (k=0; k \leq m; k++) \\
p[j][k] = \ldots & \quad p[j][k] = \ldots
\end{align*}
\]

Same behavior for homogeneous memory access, but:

\[\begin{align*}
\text{For row major order}
\end{align*}\]

↑ Poor cache behavior \quad \text{Good cache behavior} ↑

\[\begin{align*}
\text{memory architecture dependent optimization}
\end{align*}\]
Program transformation “Loop interchange”

Example:

```c
#define iter 400000
int a[20][20][20];
void computeijk() {int i,j,k;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            for (k = 0; k < 20; k++) {
                a[i][j][k] += a[i][j][k];
            }
        }
    }
}
void computeikj() {int i,j,k;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            for (k = 0; k < 20; k++) {
                a[i][k][j] += a[i][k][j];
            }
        }
    }
}
```

Improved locality

```c
start = time(&start);
for(z = 0; z < iter; z++) computeijk();
end = time(&end);
printf("ijk =%16.9f \n", 1.0* difftime(end, start));
```
Results:
strong influence of the memory architecture

Loop structure: i j k

Dramatic impact of locality

<table>
<thead>
<tr>
<th>Processor</th>
<th>Ti C6xx</th>
<th>Sun SPARC</th>
<th>Intel Pentium</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduction to [%]</td>
<td>~ 57%</td>
<td>35%</td>
<td>3.2 %</td>
</tr>
</tbody>
</table>

Not always the same impact..

Transformations
“Loop fusion” (merging), “loop fission”

\[
\text{for}(j=0; j<=n; j++) \\
p[j] = \ldots; \\
\text{for } (j=0; j<=n; j++) \\{ \\
p[j] = p[j] + \ldots \}
\]

Loops small enough to allow zero overhead
Better locality for access to p.
Better chances for parallel execution.

Which of the two versions is best?
Architecture-aware compiler should select best version.
Example: simple loops

```c
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];

void ssl1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j] += 17;   
        }
    }
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            b[i][j] -= 13;   
        }
    }}

void msl1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j] += 17;
            b[i][j] -= 13;
        }
    }}

void mm1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j] += 17;
            b[i][j] -= 13;
        }
    }}
```

Results: simple loops

Merged loops superior; except Sparc with –o3
Loop unrolling

```
for (j=0; j<=n; j++)
p[j] = ...
```

```
for (j=0; j<=n; j+=2)
{p[j] = ... ; p[j+1] = ...
}
```

factor = 2

**Better locality** for access to p.
Less branches per execution of the loop. More opportunities for optimizations.
Tradeoff between code size and improvement.
Extreme case: completely unrolled loop (no branch).
High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding
Program transformation
Loop tiling/loop blocking: - Original version -

for (i=1; i<=N; i++)
    for (k=1; k<=N; k++)
        r=X[i,k]; /* to be allocated to a register*/
        for (j=1; j<=N; j++)
            Z[i,j] += r* Y[k,j]
} % Never reusing information in the cache for Y and Z if N is large or cache is small (O(N³) references for Z).
Loop tiling/loop blocking
- tiled version -

\[
\begin{align*}
\text{for } (\text{kk}=1; \text{kk} \leq N; \text{kk}+=B) \\
\text{for } (\text{jj}=1; \text{jj} \leq N; \text{jj}+=B) \\
\text{for } (i=1; i \leq N; i++) \\
\quad \text{for } (k=\text{kk}; k \leq \min(\text{kk}+B-1,N); k++)
\quad r=X[i][k]; /* to be allocated to a register*/ \\
\quad \text{for } (j=\text{jj}; j \leq \min(\text{jj}+B-1, N); j++)
\quad Z[i][j] += r \times Y[k][j]
\end{align*}
\]

Reuse factor of \(B\) for \(Z\), \(N\) for \(Y\)
\(O(N^3/B)\) accesses to main memory

Compiler should select best option

High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding
Transformation “Loop nest splitting”

Example: Separation of margin handling

- many if-statements for margin-checking
- no checking, efficient
- only few margin elements to be processed
High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding
Array folding

Initial arrays

&A

&B

&C

&D

&E
Array folding

Unfolded arrays

addresses

memory size

t
Intra-array folding

Inter-array folding

addresses

memory size

addresses

memory size

t
Summary

- Floating-point to fixed point conversion
  - Range estimation
  - Conversion
  - Analysis of the results
- High-level loop transformations
  - Fusion
  - Unrolling
  - Tiling
  - Loop nest splitting
  - Array folding