Rechnerarchitektur (RA)

Sommersemester 2017

Architecture-Aware Optimizations
-Software Optimizations-

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

2017/07/18
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
  - 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

IWL=3

FWL

S 1 0 0 . . . 0 0 0 0 1 0

(b) Fixed-Point

hypothetical binary point

© Ki-II 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

- Floating-Point C Program
  - Range Estimator
  - Floating-Point to Fixed-Point C Program Converter
  - Fixed-Point C Program

Range Estimation

- Execution
- Manual specification
- IWL information

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

Cycles

ADPCM

<table>
<thead>
<tr>
<th>Format</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>
Performance Comparison - SNR -

ADPCM

SNR (dB)

A  B  C  D

Fixed-Point (16b)
Fixed-Point (32b)
Floating-Point

© Ki-II 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)

Column major order (FORTRAN)
Best performance of innermost loop corresponds to rightmost array index

Two loops, assuming row major order (C):

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

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

Same behavior for homogeneous memory access, but:

For row major order

↑ Poor cache behavior       Good cache behavior ↑

Memory architecture dependent optimization
Example:
...#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];}}}}...
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 reduction to [%]</th>
<th>Sun SPARC</th>
<th>Intel Pentium</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>35%</td>
<td>3.2%</td>
</tr>
</tbody>
</table>

Ti C6xx ~ 57%
Intel Pentium 3.2%
Sun SPARC 35%

Not always the same impact..

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

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

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

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

void ss1() {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 ms1() {int i,j;
  for (i=0;i<size;i++){
    for (j=0;j<size;j++){
      a[i][j]+=17;
    }
  }
  for (j=0;j<size;j++){
    for (i=0;i<size;i++){
      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

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

```c
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(N^3) references for Z).
Loop tiling/loop blocking
- tiled version -

```c
for (kk=1; kk<= N; kk+=B)
  for (jj=1; jj<= N; jj+=B)
    for (i=1; i<= N; i++)
      for (k=kk; k<= min(kk+B-1,N); k++)
        r=X[i][k]; /* to be allocated to a register*/
        for (j=jj; j<= min(jj+B-1, N); j++)
          Z[i][j] += r* Y[k][j]
    }
```

Reuse factor of B for Z, N for Y

O(N³/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
Inter-array folding

Intra-array folding

depth
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