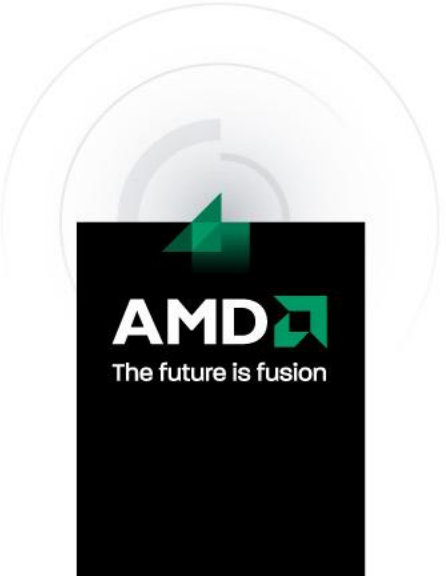
# if-Conversion & loop-flattening for AMD Bulldozer Processors

Sebastian Pop  Reza Yazdani  Quentin Neill | October 25, 2010



# Outline

- Motivation
- If-conversion
- Results
- Loop flattening by elimination of back-edges
- Loop flattening
- Optimizations
- Conclusion

# Motivation

- Vectorization of loops with irregular control flow.
- Uniform application of vectorization on inner and outer loops.
- Removing inner loops with small number of iterations.
- Avoiding branch miss-prediction.
- Avoiding bubbles in scheduled-code. Technique is very effective in SIMD environment.
- Improving instruction level parallelism (ILP) by having independent execution paths.
- Avoiding branch diversion in Single-Program Multiple-Data (SPMD) models or SIMD wavefronts supported by GPU's.

# if-conversion

- Conversion of control flow to data flow.
- Evaluate both expression, and generate "? :" operation.

```
for (i = 0; i < N; i++) {
  i f (cond)
    a = x;
  else
    a = y;
  b = a + 2;
}
```

```
for (i = 0; i < N; i++) {
  a_0 = x;
  a_1 = y ;
  a_2 = cond ? a_0 :a_1;
  b_3 = a_2 + 2;
}
```

# if-conversion Restrictions

- Without if-conversion only one case is evaluated.
- In the following example:

```
for (i = 0; i < N; i++) {
  i f ( p )
    *p = 3;
}
```

- An exception is raised if p = 0 after if-conversion:

```
t = *p;
*p = (p ? 3 : t);
```

# Examples that if-conversion may cause traps

- writes in a read only memory,
- accessing out-of-range memory,
- accessing invalid pointers,
- division by zero.

# if-conversion with safe memory accesses

- Data references in if-converted branches occur in parts of the loop unconditionally executed.

- There exists at least one unconditional write to the same memory (e.g. an array access).

# if-conversion results

- Code extracted from Ffmpeg which is used in encoding and decoding of video streams:

```
for (i = 0 ; i <= nCoeffs ; i++)
  i f (block[i] < 0)
    block[i] = block[i] * qmul - qadd;
  else
    block[i] = block[i] * qmul + qadd;
```

- After if-conversion and optimization:

```
for (i = 0 ; i <= nCoeffs ; i++) {
  t = block[i] < 0 ? – qadd : qadd;
  block[i] = block[i] * qmul + t;
}
```

- This DCT kernel is vectorized after if-conversion.
- Two versions were considered, one targeting the SSE2 vector instructions and another targeting XOP.
- We used a simulator for the upcoming Bulldozer processors to evaluate the effectiveness of the vectorization.
- The XOP version executed 2.51 times faster than the SSE2 version.
- Part of difference is due to XOP executing fewer instructions, 163882 instructions, compared to the SSE2 version that executes 540714 instructions.

# If-conversion without restrictions

- compiler generates valid code and free of trapping by:
  - Creating dummy objects that are written to or read from to avoid illegal access.
- Dummy objects are read or write in cases access conditions not met.
- Following examples show the concept.

# Solution to null pointer

```
for (i = 0; i < N; i++) {
  i f ( p )
    *p = 3;
}
int t1;
int *q = &t1;
for (i = 0; i < N; i++) {
   t = p ? p : q;
  *t = 3;
}
```

# If-conversion without restrictions examples cont.

```
for ( i = 0 ; i < N; i ++) {
  i f (cond)
    A[ i ] = expr ;
}
```

- Converted to:

```
int t1;
int *q = &t1;
for ( i = 0 ; i < N; i ++) {
   t = cond ? &A[ i ] : q ;
  *t = expr;
}
```

# if-conversion without restrictions cont.

- Similar techniques apply to writes/reads from memory, out-of-range accesses, and access of invalid pointers.
- In division by zero case, we assume the conditional operations exists for all operations including the divide in Gimple.
- "conditional statements" are generated to replace if-conversion for operations not supported in hardware in later phases.

# Loop Flattening

- Two approaches:
  - Loop flattening by elimination of back-edges.
  - Loop flattening on a high level semantic representation of loops.

# Loop flattening by elimination of back-edges

- Transforms control dependences into data dependences by using a boolean.
- Back-edge of the outer loop controls the execution of the flattened inner loop.
- A boolean variable controls the execution of the basic blocks between the outer loop and the inner loop.

# back-edge removal example

```
int foo ( voi d )
{
   int i , j;
   f o r ( i = 0 ; i < N; i ++) {
    f o r ( j = 0 ; j < N; j ++)
      A[ i ] [ j ] = i + j ;
   }
    return A[ 12 ] [ 3 ] ;
}
```

```
int foo ( void )
{
  int i , j , f lag = true;
  for ( i = 0 ; i < N;) {
    if ( flag)
       j = 0, flag = false;
    if ( j < N) {
       A[ i ] [ j ] = i + j ;
       j++;
    } e l s e {
       i++ , f lag = true ;
    }
    return A[ 12 ] [ 3 ] ;
  }
```

# back-edge removal example after if-conversion

```
int bar ( v oi d ) {
  int i , j , t;
  int *q = &t; bool flag = true, cond;
  for ( i = 0 ; I < N; i = cond ? i : i + 1) {
    j = f lag ? 0 : j ;
    flag = false, cond = (j < N);
    p = &A[ i ] [ j ] ;
    p = cond ? p : q ;
   *p = i + j;
   j = cond ? j + 1 : j ;
    flag = cond ? flag : true;
  }
}
```

# Loop flattening on a high level semantic representation of loops

- Transforms loop nests to a single loop.
- Auto vectorization applied on the loop body.
- Canonical Example:

```
for ( i = 1 ; i <= 6 ; i++)
  f o r ( j = 1 ; j <= 6 ; j++)
     S1 ( i , j )
```

- Loop flattening linearizes the iteration space using the function x = 6 * i + j.
- GCC with Graphite would produce this code:

```
f o r ( x =7; x <=42; x++) {
  i = floord ( x - 1 ,6) ;
  S1 ( i , x – 6 * i ) ;
}
```

# Vectorization

- Currently code generated by Graphite would not vectorize flattened loops, because:
  - old induction variables contain modulo and division expressions,
  - Scalar evolution framework does not handle complex expressions,
  - new data reference accesses cannot be analyzed.

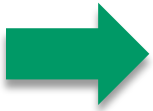
# Optimizations

- Loop flattening and if-conversion transforms nested loops containing well formed structures to a loop with one basic block.
- These transformations can simplify complicated optimizations such as partial redundancy elimination (PRE) to simple basic block optimizations.
- Following example shows how PRE is done by common subexpression elimination.

```
for (…) { …
  if (exp) {…
  }
  else {….
       n = x + y; …
  }
  k = x + y;
```

- After if conversion:

```
for (…) { …
  t = x + y;
  n = exp ? n : t;
  ….
  K = x + y;
}
```



```
for (…) { …
  t = x + y;
  n = exp ? n : t;
  …
  K = t;
}
```

- Putting if's back

```
for (…) { …
  t = x + y;
  if (exp) {…
   }
   else {….
        n = t; …
   }
   k = t;
}
```

# Conclusion

- if-conversion and loop-flattening boosts performance in SIMD and SPMD execution models.
- They have substantial performance effects on nested loops with small inner loop and software pipelining of nested loops.
- Some more work is needed to make flattened loop to vectorize.
  - Induction variables information needs to be kept after loop-flattening and passed to the compiler phases requiring such information for optimization.

# Vectorization with if-conversion and loop flattening

```
for (i = 0; i < 1000; i++) {
    if (i & 1) a[i] = b[i] + 1;
    for (j = 0; j < 50; j++) {
      if (j & 1) c[i,j] = d[j] + 1;
    }
}

// After loop flattening
  flag = true;
  for (i = 0; i < 1000; i++) {
    if (flag) {
      if (i & 1) {
         a[i] = b[i] + 1; flag = false; j = 0; }
    }
    if (j & 1) c[i,j] = d[i] + 1;
    i = (j < 50) ? (j++, i) : (flag = true, i + 1);
}
```

```
// After if-conversion and vectorization.
  flag = true;
  for (i = 0; i < 1000;) {
    //if (flag) {
    t0 = (i & 1) & flag;
    t1 = (i + 1) & 1 & flag;
    x[0:2] = b[i:2] + 1;
    a[i:2] = (t0, t1) x[0:2]; // (a[i] = x[0]) <- (t0 == 1),
                              // (a[i+1] = x[1]) <- (t1 == 1)
    flag = flag ? (j = 0, false): flag;
    //}
    t2 = j & 1;
    y3 = (j + 1) & 1;
    y[0:2] = d[j:2] + 1;
    c[j:2] = (t2, t3) y[0:2];
    i = (j < 50) ? (j += 2, i) : (flag = true, i + 1);
   }
```