Loop Optimizations - II

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Loop optimization techniques - Unrolling

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

⇨

```
for(int i=0;i<n;i+=2) {
   sum1 += a[i]
   sum2 += a[i+1]
}
..
sum = sum1 + sum2
```

- Why do this?
  - Avoid branches, which are costly ops
  - Expose Instruction-Level Parallelism (ILP)
    - Modern systems have hardware that can utilize available ILP
- Pros and Cons?
  - Cons: increases code, register pressure, sometimes complicates CFG
  - Pros: may avoid branching, expose invariants, allow parallelization and other optimizations

Loop optimization techniques - Unrolling

- Loop Peeling

```
void bar(int);
void foo() {
    int a = 0;
    int y = 0;
    int x = 0;
    for(int i = 0; i <100; ++i) {
        bar(a);
        x = y;
        y = a + 1;
        a = 5;
    }
    bar(x);
}
```

```
//iteration0
bar(a);
x = y;
y = a + 1;
++i;
//check if i < 100;
//iteration1
bar(a);
x = y;
y = a + 1;
++i;
//check if i < 100;
//iteration2
bar(a);
x= y
y = a + 1;
++i;
//check if i < 100;

for(int i = 3; i <100; ++i) {
```

Loop optimization techniques - Unrolling

- Loop Peeling

Instn/iter	0	1	2	3
bar(a)	bar(0)	bar(5)	bar(5)	bar(5)
x=y+1	x=1	x=2	x=7	x=7
y=a+1	y=1	y=6	y=6	y=6
a=5	a=5	a=5	a=5	a=5

```
void bar(int);
void foo() {
  int a = 0;
  int y = 0;
  int x = 0;
  for(int i = 0; i <100; ++i) {
    bar(a);
    x = y;
    y = a + 1;
    a = 5;
  }
  bar(x);
}
```

- Discover invariants and avoid phi nodes
  - Demo: `make looppeel`
  - Try it yourself: change `x=y` to `x=y+1` and observe .png files. Why do you see that phi node for x has not been eliminated?
- Eliminate If conditions that exist inside loops
  - Demo: `make looppeel2`

Loop optimization techniques - Unrolling

- Full Unroll
  - Unroll the loop fully or don't unroll
- Loop Unroll
  - First try Full Unroll, if not successful, do partial unroll / runtime unroll
- Partial Unroll
  - Cannot do this before inlining (demo: `make unroll1`)
  - Runtime unroll (demo: `make unroll2`)
    - Try it yourself: remove the `-unroll-runtime -unroll-count=4` arguments and observe the .png files

Loop optimization techniques - Unrolling

- Unroll and Jam (fuse)

```
for i..
  ForeBlocks(i)
  for j..
    SubLoopBlocks(i, j)
  AftBlocks(i)
```

⇨

```
for i... i+=2
  ForeBlocks(i)
  ForeBlocks(i+1)
  for j..
    SubLoopBlocks(i, j)
    SubLoopBlocks(i+1, j)
  AftBlocks(i)
  AftBlocks(i+1)
Remainder
```

  - Make sure that ForeBlocks(i+1) can actually execute before SubLoopBlocks(i, j) and AftBlocks(i)
    - Need to use dependency analysis.

Loop optimization techniques

- Loop Flatten

```
for (int i = 0; i < N; ++i)
  for (int j = 0; j < M; ++j)
    f(A[i*M+j]);
```

⇨

```
for (int i = 0; i < (N*M); ++i)
  f(A[i]);
```

- The collapse() clause in OpenMP

```
#pragma omp parallel for collapse(2)
for(int i=0;i<N;i++)
    for(int j=0;j<M;j++)
        f(A[i*M+j]);
```

- Some of the pragmas supported in Clang

```
#pragma unroll(n)
#pragma nounroll
#pragma clang loop unroll(enable)
#pragma clang loop unroll(disable)
#pragma clang loop unroll(full)
#pragma clang loop unroll_count(4)
```

Try it yourself

```
for(i=0;i<n;i++) {
    for(j=0;j<n;j++) {
        X[2*i+1][j] = Y[i+1][j];
        Y[i][j] = X[2*i-1][j];
    }
}
```

- Describe the dependences that exist on the X and Y arrays.

(For each of the arrays, if no dependence exists say “No dependence”. If dependence exists, say what kind of dependence exists (“Anti”, “Flow”, “Output”), Give the distance and the direction vectors.)

- Which loop can be parallelized above? Assume no other optimizations are performed.
- Which of the following transformations i) strength reduction ii) identifying invariants and factoring/moving out of the loop iii) loop fusion iv) loop distribution / loop splitting increases the number of loops that can be executed in parallel OR that makes it more easier to exploit existing parallelism? Show the resulting code after transformation and explain how it enhances parallelism

• 4)

a) X Array: "Flow", (1,0), (+,0) RAW

Y Array: "Anti", (1,0), (4,0) WAR

−1 −1

b) Only the j loop can be parallelized

c) Loop distribution / Loop splitting increases the number of loops that can be executed in parallel.

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++) {
        X[2*i+1][j] = Y[i+1][j];
        Y[i][j] = X[2*i-1][j];
    }
}
```

here each i,j node can be executed in parallel

⇓

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        X[2*i+1][j]=Y[i+1][j];
```

followed by

⇓

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        Y[i][j]=X[2*i-1][j]
```

	i=0, j=0	i=0, j=1	i=0, j=2	i=0, j=3	i=0, j=4
W:	X_{10}, Y_{00}	X_{11}, Y_{01}	X_{12}, Y_{02}	X_{13}, Y_{03}	X_{14}, Y_{04}
R:	X_{-10}, Y_{10}	X_{-11}, Y_{11}	X_{-12}, Y_{12}	X_{-13}, Y_{13}	X_{-14}, Y_{14}
	i=1, j=0	i=1, j=1	i=1, j=2	i=1, j=3	i=1, j=4
W:	X_{30}, Y_{10}	X_{31}, Y_{11}	X_{32}, Y_{12}	X_{33}, Y_{13}	X_{34}, Y_{14}
R:	X_{10}, Y_{20}	X_{11}, Y_{21}	X_{12}, Y_{22}	X_{13}, Y_{23}	X_{14}, Y_{24}
	i=2, j=0	i=2, j=1	i=2, j=2	i=2, j=3	i=2, j=4
W:	X_{50}, Y_{20}	X_{51}, Y_{21}	X_{52}, Y_{22}	X_{53}, Y_{23}	X_{54}, Y_{24}
R:	X_{30}, Y_{30}	X_{31}, Y_{31}	X_{32}, Y_{32}	X_{33}, Y_{33}	X_{34}, Y_{34}

W: $X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{30}, X_{31}, X_{32} \dots$

R: $Y_{10}, Y_{11}, Y_{12}, Y_{13}, Y_{14}, Y_{20}, Y_{21}, Y_{22} \dots$

followed by

W: $Y_{00}, Y_{01}, Y_{02}, Y_{03}, Y_{04}, Y_{10}, Y_{11}, Y_{12}, Y_{13}, Y_{14} \dots$

R: $X_{-10}, X_{-11}, X_{-12}, X_{-13}, X_{-14}, X_{10}, X_{11}, X_{12}, X_{13}, X_{14} \dots$

"Flow" remains "flow" after splitting.

"Anti" remains "anti" after splitting.

Parallel Functional Units

- Fused Multiply Add (FMA)
  - Fuses multiply and an add into one functional unit (c=c+a*b)
  - The functional unit consists of 3 independent subunits
    - The units are Pipelined
  - Example:

```
sum=0.0
for (i=0;i<n;i++)
  sum=sum+a[i]*b[i]
```

  - Try it yourself: Suppose the FMA unit takes 3 cycles to complete, how many cycles do you need to execute the above code snippet?
  - With loop unrolled 4 times? Assume n is divisible by 4.

Vectorization in LLVM

- Let the compiler do it automatically (Auto vectorization)
- Let programmer give hints (to compiler) via pragmas, keywords
- Let the programmer use intrinsics (APIs that allow to manipulate vector registers via special instructions)

Auto-Vectorization in LLVM

- Loop Vectorizer

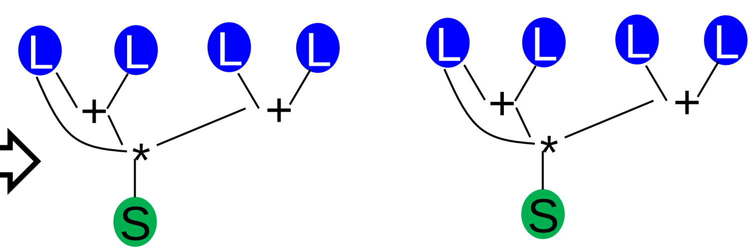
```
sum=0.0
for (i=0;i<n;i++)
  sum=sum+a[i]
```

```
for(int i=0;i<n;i++) {
   sum1 += a[i]
   sum2 += a[i+1]
}
```

  - Expands the body of the loop to operate on instructions across consecutive iterations

- SLP Vectorizer
  - Merges scalars into vectors

```
void foo(int *B, int *A){
      A[0]=(B[0]+B[1])*(B[2]+B[3]);
      A[1]=(B[3]+B[4])*(B[5]+B[6]);
}
```



Auto-Vectorization in LLVM - Method

- Loop Vectorizer and SLP vectorizer
  - Checks if legal to vectorize
  - Builds a vectorizable tree (SLP vectorizer only)
  - Estimates cost
  - Vectorizes
    - Unroll for ILP (Loop vectorizer)
    - Identify consecutive stores, loads, reductions, and patterns (SLP vectorizer)

Auto-Vectorization in LLVM - Features

- Loop Vectorizer can handle:
  - Loops with unknown trip counts `for(int i=start;i<end;i++)`
  - Reverse iterators `for(int i=end;i>start;i--)`
  - Insert runtime checks for memory overlap

```
void foo(int *B, int *A){
        A[0]=(B[0]+B[1])*(B[2]+B[3]);
        A[1]=(B[3]+B[4])*(B[5]+B[6]);
}
```

  - Perform reductions

```
sum=0.0
for (i=0;i<n;i++)
  sum=sum+a[i]
```

  - If-conversion, partial vectorization, mixed types, well-known functions etc.

Auto-Vectorization in LLVM - Features

- Inlining with restrict

```
void foo(int *restrict B, int *restrict A){
        for(int i=0;i<n;i++)
                A[i+1]=B[i]
}

void bar(){
        ...
        foo(ptrB, ptrA)
}
```

- After inlining:

```
void bar(){
...
        for(int i=0;i<n;i++)
                ptrA[i+1]=ptrB[i]
}
```

Auto-Vectorization in LLVM – Restrict keyword

```
void updatePtrs(size_t *ptrA, size_t *ptrB, size_t *val)
{
  *ptrA += *val;
  *ptrB += *val;
}
```

```
; Hypothetical RISC Machine.
ldr r12, [val]
ldr r3, [ptrA]
add r3, r3, r12
str r3, [ptrA]
ldr r3, [ptrB]
ldr r12, [val]
add r3, r3, r12
str r3, [ptrB]
```

Why do we need to load from val again?

```
void updatePtrs(size_t *restrict ptrA, size_t *restrict ptrB, size_t *restrict val);
```

```
ldr r12, [val]
ldr r3, [ptrA]
ldr r4, [ptrB]
add r3, r3, r12
add r4, r4, r12
str r3, [ptrA]
str r4, [ptrB]
```


source: restrict - Wikipedia