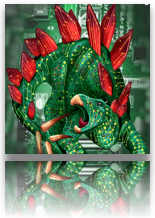
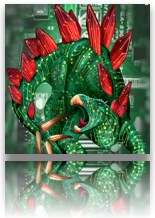


Compiler & Loop Transformation

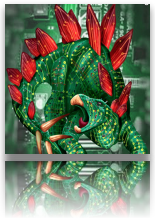
Lecture 6



Compiler & Loop Transformation

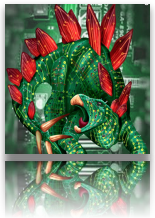
no one write low-level programming today

- ★ How does compiler work? (Compiler Revisit)
- ★ Basic Block
- ★ Optimizations
- ★ Loop Unroll & Software Pipelining
- ★ Loop Transformation
  - Common transformations
  - Polyhedral model



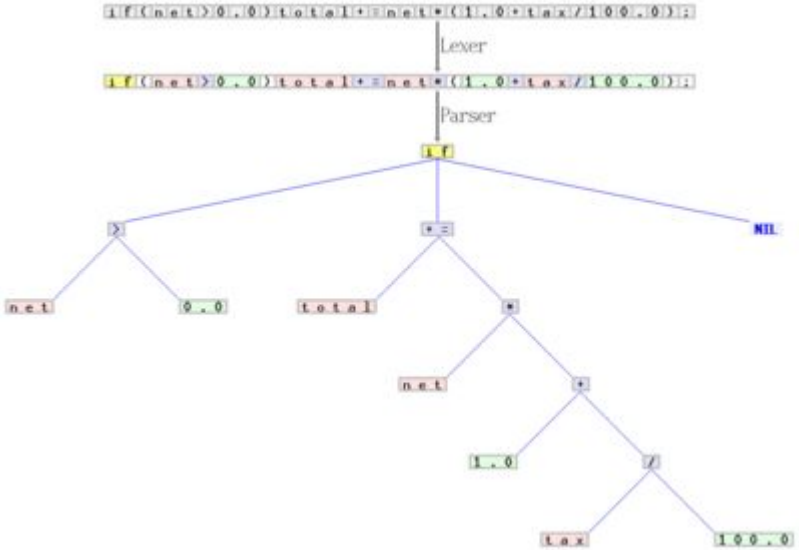
How does compiler work?

- ★ Historically, early operating systems and software were written in assembly language. (We even do hand assembler.)
- ★ 60s and early 70s, several languages (BCPL, BLISS, B) has been developed.
- ★ Notably are:
  - 1952 - A-0 was developed (and coined the term compiler)
  - 1954-1957 - IBM develop FORTRAN (first high-level language.) first complier
  - 1959 - COBAL design was drawn from A-0. It was later compiled on multiple architectures.
  - 1958-1962 - MIT designed LISP
- ★ 1969 - Dennis Ritchie and Ken Thompson creates bootstrapping compiler for B and wrote Unics (later spelled UNIX.). Later (1973) C was developed.

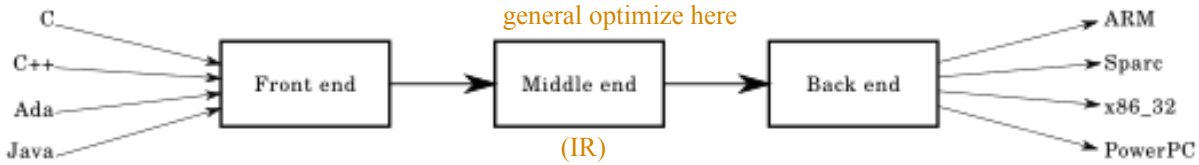


How does compiler work? (ctd.)

Lexer : tokenize string into grouping
Parser : create parse tree (make that group meaningful)

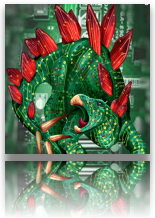


- ★ Single pass vs. Multi-pass compilers
- ★ Modern compilers
  - Front end - transform input into Intermediate Representation (IR)
    - Lexer, Parser
  - Middle end - Optimization
  - Back end - target-dependent optimizations, assembly/code generator



* ILP happen here!!

Picture taken from wikipedia.

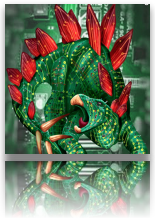


Basic Block and Optimization

- ★ Basic block is a block with one entry point and one exit point
- ★ Translate, once a first instruction in the block is executed, the rest are necessarily to be executed in order.
- ★ What ends a basic block?
  - ○ ?
- ★ What starts a basic block?
  - ○ ?
- ★ Why do we care about basic block?

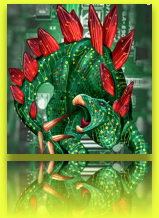
There is the trends that bigger basic block gives more ILP

* How to decrease amount of if/else in code



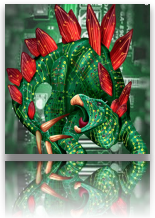
Optimization (ctd.)

- ★ Peephole optimizations
  - (A window optimization)
- ★ Local / Global optimizations
  - Local - within a basic block
  - Global - beyond a basic block
- ★ Loop optimizations
- ★ Data-flow optimizations
- ★ Interprocedural/link-time optimizations
- ★ Machine code optimizations
- ★ etc...



How many basic blocks do we have in a loop?

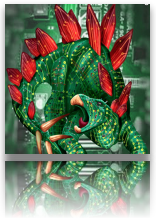
no if/else in basic block — 1 basic block in general



Loop Unrolling & Software Pipelining (revisit)

- ★ Loop unrolling (aka loop unwinding)
- ★ Space-Time tradeoff ? Why it is faster?

make basic block bigger —> more ILP
and
1. use more space
2. take less time



Loop Transformation

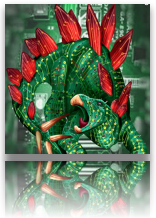
- ★ Compilers are generally good with simple code.
- ★ Loop Transformation can trigger compiler to performance better optimizations.
- ★ Here is an example project from CERN/CMS software.

Table 2. Number of instructions executed for a selection of functions from CMSSW_10_2_3 compiled with GCC 7.3.1

Function name	Initialization(-O2)	Optimized loops
fillCovariance	49,617,744	13,250,193
TrackExtra	10,953,225	6,761,250
TrackBase	9,801,004	5,183,644
VertexCompositePtrCandidate	11,995,424	2,249,142

Table 3. Average execution time of repeatedly selected loop patterns in nanoseconds of CMSSW_10_2_3 compiled with GCC 7.3.1

Optimization description	Initialization(-O2)	Optimized loops	Speed up
Index set splitting	125.34	79.67	1.57
Unrolled loop with directive	122	76.34	1.59
Loop reordering	112	64.67	1.73



Common loop transformations

- ★ Fission (aka. Distribution) transform from fig. 1 to fig. 2
- ★ Fusion (aka. Combining)
- ★ Improve locality

has the chance that a[i] and b[i] store in the same cache block(conflict miss)

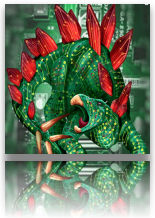
```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
    a[i] = 1;
    b[i] = 2;
}
```

fig. 1

has more spatial locality (close addr. work together)

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
    a[i] = 1;
}
for (i = 0; i < 100; i++)
{
    b[i] = 2;
}
```

fig. 2



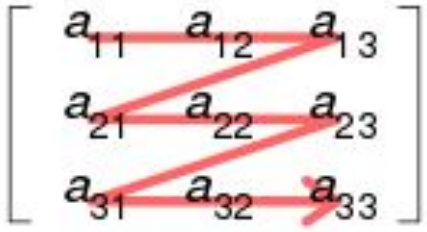
Common loop transformations

- ★ Loop interchange
- ★ Improve locality

this is faster (with cache)
bs. in mem its like a[0, 0],…,a[0, 99],a[1, 0],…,a[1, 99], ...
so the path of access is like in memery

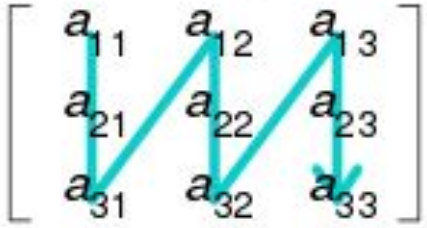
```
for i in 1..n:
  for j in 1..n:
    a[i, j] = …
```

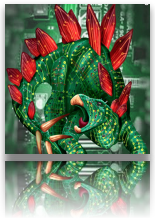
Row-major order



Column-major order

```
for j in 1..n:
  for i in 1..n:
    a[i, j] = …
```





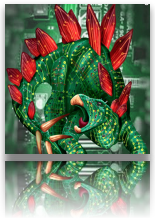
Common loop transformations

- ★ Loop inversion
- ★ Convert *while* to *if* and *do..while*
- ★ Eliminate pipeline stall?

```
int i, a[100];
 i = 0;
 while (i < 100) {
   a[i] = 0;
   i++;
 }
```

```
i = …
a[100] = …

if i < 100:
  do:
    a[i] = ...
    i += 1
  while(i<100)
```

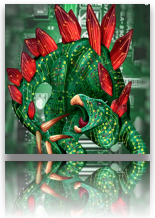


Common loop transformations

- Loop-invariant (code motion)

```
int i = 0;
while (i < n) {
     x = y + z;
     a[i] = 6 * i + x * x;
     ++i;
}
```

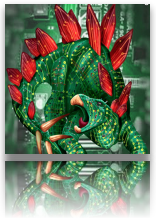
```
int i = 0;
if (i < n) {
     x = y + z;
     int const t1 = x * x;
     do {
     a[i] = 6 * i + t1;
     ++i;
     } while (i < n);
}
```



Common loop transformations

- ★ Loop splitting
- ★ Eliminate dependencies by breaking into multiple loops

```
 int p = 10;
 for (int i=0; i<10; ++i)
 {
   y[i] = x[i] + x[p];
   p = i; //add i to
variable p
 }
```



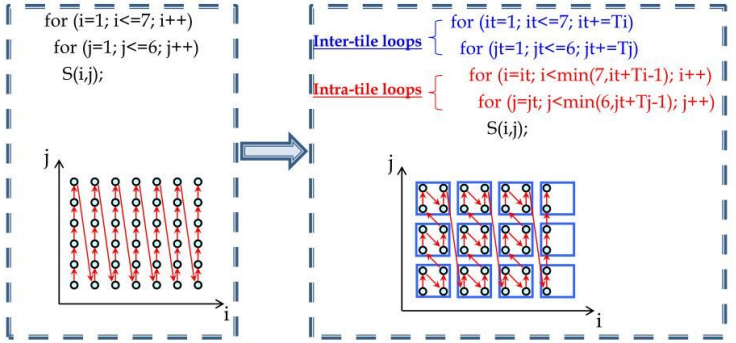
Common loop transformations

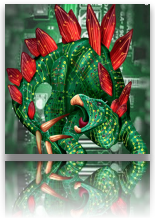
- ★ Tiling / blocking / Loop Nest Optimizations
- ★ Cache optimizations (reused data in cache)

```
int i, j, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i++) {
   c[i] = 0;
   for (j = 0; j < n; j++) {
        c[i] = c[i] + a[i][j] * b[j];
   }
}
```

Loop Tiling

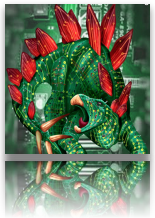
- A key loop transformation for:
  - Efficient coarse-grained parallel execution
  - Data locality optimization





Tiling (ctd.)

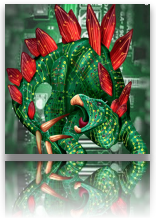
```
int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i += 2) {
   c[i] = 0;
   c[i + 1] = 0;
   for (j = 0; j < n; j += 2) {
    for (x = i; x < min(i + 2, n); x++) {
     for (y = j; y < min(j + 2, n); y++) {
       c[x] = c[x] + a[x][y] * b[y];
     }
    }
   }
}
```



Common loop transformations

- ★ Loop unswitching
- ★ Remove control dependency, increase chances for parallelization/vectorization

```
int i, w, x[1000], y[1000];
for (i = 0; i < 1000; i++) {
  x[i] += y[i];
  if (w)
    y[i] = 0;
}
```

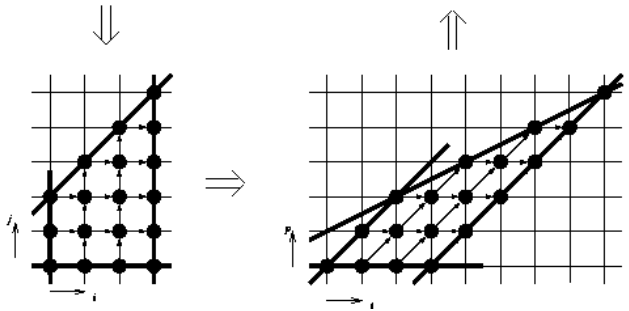


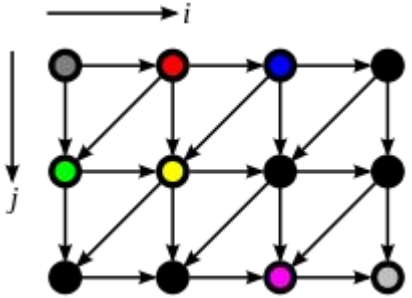
Polyhedral model/Polytope

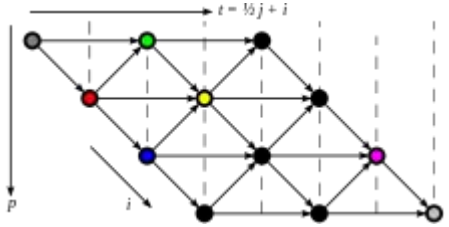
- ★ Skewing (reduce dependencies)
- ★ Consider a rectangular with different shapes, but same areas

```
for i := 0 to n do
    for j := 0 to i + 2 do
        A(i, j) := A(i − 1, j)
                 +A(i, j − 1)
    end
end
```

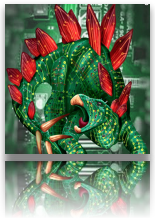
```
for t := 0 to 2n + 2 do
    forall p := max(0, t−n) to min(t, ⌊t/2⌋ + 1) do
        A(t−p, p) := A(t−p−1, p)
                   +A(t−p, p−1)
    end
end
```





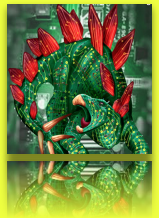


https://www.infosun.fim.uni-passau.de/cl/loopo/doc/loopo_doc/node3.html

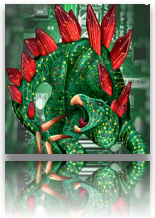


Common loop transformations

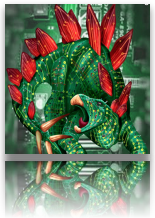
- ★ Index splitting ?



There exists several techniques that are not mentioned here.



Exercises

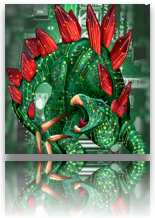


Exercises

- Try code the matrix (eg. 8x8) multiplication in a straight forward algorithm. Compile it with various optimizations.
  See if the compiler is able to perform loop interchange.

  Try this simple code. See if the compiler is able to perform any optimization. Please provide your analysis.

```
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
       A[i,j] = a[i,j] + 2.5;
```



End of Lecture 6