Analysis and modeling of

Computational Performance

Classical sequential optimization

Classical software optimization

- Classical optimization concerns mainly the single thread performance and aims primarily at:
  - reducing the number of performed operations
  - proper utilization of vector capabilities of the hardware
  - proper utilization of memory hierarchy
  - removing dependencies between instructions
- Classical optimization techniques can be applied manually
  - most of the techniques are also utilized by the compilers
  - it is important not to inhibit compiler optimizations by manual source code changes
    - it is unfortunately a common case when manually optimized code performs worse than before optimization due to improper interaction with an optimizing compiler
- Classical optimization can speed-up program execution dozens of times in certain situations

Classical optimization techniques

- General techniques for variables and expressions:
  - *constant folding*
    - instead of: `for(i=...) r = 2*PI*r[i];`
    - use: `const double 2_PI = 2*PI;  for(i=...) r = 2_PI*r[i];`
  - *copy propagation*
    - instead of: `y = x; ...; z = f(y); // read-after-write`
    - use: `y = x; ...; z = f(x); // no dependence`
  - *strength reduction*
    - instead of: `y = pow(x,4);`
    - use: `temp = x*x; y = temp*temp;`
  - *common subexpression elimination*
    - instead of: `a = b * c + g; d = b * c * e;`
    - use: `temp = b*c; a = temp + g; d = temp * e;`

Classical optimization techniques

- Loop oriented techniques
  - *induction variable simplification*
  - *loop invariant code motion*

before:

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

after LICM:

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

after LICM+IVS:

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

Classical optimization techniques

- Loop oriented techniques
  - *loop unrolling*
    - instead of:

```
dot = 0.0;
for(i=0; i<N; i++)
{
  dot += X[i]*X[i];
}
```

    - use:

```
dot = 0.0;
for(i=0; i<N; i+=4) // always add another loop with N%4 iterations
{
  dot += X[i]*X[i]+X[i+1]*X[i+1]+X[i+2]*X[i+2]+X[i+3]*X[i+3];
}
```

Classical optimization techniques

- Loop oriented techniques
  - loop *fusion* (e.g. to reduce the number of memory accesses)

- before

```
for(k=0; k<16; k++){
  a_tab[k] += 2*c_tab;
  b_tab[k] += 2*d_tab;
}

for(k=0; k<16; k++){
  a_tab[k] += d_tab;
  b_tab[k] += c_tab;
}
```

- after

```
for(k=0; k<16; k++){
  a_tab[k] += 2*c_tab+d_tab;
  b_tab[k] += 2*d_tab+c_tab;
}
```

Classical optimization techniques

- Loop oriented techniques
  - *loop fission* (e.g. to reduce register pressure)

- before

```
for(i=0;i<1000000;i++){
   for(k=0; k<16; k++){
     a_tab[k] += 1.0;
     b_tab[k] += 1.0;
     c_tab[k] += 1.0;
     d_tab[k] += 1.0;
   }
}
```

- after

```
for(i=0;i<1000000;i++){
   for(k=0; k<16; k++){
     a_tab[k] += 1.0;
     b_tab[k] += 1.0;
   }
 }
 for(i=0;i<1000000;i++){
   for(k=0; k<16; k++){
     c_tab[k] += 1.0;
     d_tab[k] += 1.0;
   }
}
```

Classical optimization techniques

- Loop oriented techniques
  - *loop interchange* (e.g. to correct memory access pattern)
    - before:

```
for( i=0; i<N; i++ ){
  for( j=0; j<N; j++ ) {
    sum += a[i+j*n]; // not optimal memory access, stride n
} }
```

    - after:

```
for( j=0; j<N; j++ ){
  for( i=0; i<N; i++ ) {
    sum += a[i+j*n]; // optimal memory access, stride 1
} }
```

Classical optimization techniques

- Loop oriented techniques
  - *register blocking*
    - before:

```
for(i = 0; i < n, i++){
  for(j = 0; j < n; j++) {
    sum += a[i*n+j] * x[j];
} }
```

    - after (reduced number of memory accesses for x):

```
for(i = 0; i < n, i+=2){
  for(j = 0; j < n; j+=2) {
    t0 = x[j];
    t1 = x[j+1];
    sum += a[i*n+j] * t0 + a[i*n+j+1] * t1;
    sum += a[(i+1)*n+j] * t0 + a[(i+1)*n+j+1] * t1;
} }
```

Classical optimization techniques

- Other techniques
  - *dead code removal*
  - *tail-recursion elimination*
  - *inlining*
  - *software prefetching*
  - *software pipelining*

  software prefetching and pipelining example:

before:

```
for(i = 0; i<n, i++){
  fetch( a[i] );
  process( a[i] );
}
```

after:

```
fetch( a[0] );
for(i = 0; i<n-1, i++){
  fetch( a[i+1] );
  process( a[i] );
}
process( a[n-1] );
```

Optimizing compilers

- Usually in order to get maximal performance for the code on a given complex hardware manual optimization is not sufficient and sophisticated optimizing compilers **have to** be used
- Optimization is performed by compilers usually after syntax analysis and before object code generation
  - some options, e.g. parallelization, can be realized in a preprocessing stage by suitable compiler modules
- Optimization operates on some intermediate form of the code that usually utilizes:
  - registers
  - basic blocks
- Basic block is a fundamental portion of the code for which optimization is performed
  - basic block is a sequence of instructions having the property, that if one of them is executed than all of them are executed
    - it is impossible to jump out of a basic block (jumps end a block)
    - it is impossible to jump into the block (targets of jumps are beginnings of basic blocks)

Optimizing compilers

Source code:

```
while( j < n ) {
     k = k + 2j;
     m = 2j;
     j++;
}
```

Intermediate code:

```
A:   t1 := j;
     t2 := n
     t3 := t1 < t2
     jmp (B) t3
     jmp (C)
B:   t4 := k
     t5 := j
     t6 := t5 * 2
     t7 := t4 + t6
     k := t7
     t8 := j
     t9 := t8 * 2
     m := t9
     ....
     jmp (A)
C: ...
```

Compiler produced assembler

```
.L2
     movl -4(%ebp), %eax    // j -> eax
     cmpl -12(%ebp), %eax // n <> eax ?
     jl .L4
     jmp .L3
.L4
     movl -4(%ebp), %eax    // j-> eax
     movl %eax, %edx        // j->edx
     leal 0(,%edx,2), %eax  // eax=2*edx
     addl %eax, -8(%ebp)  // k+=eax (k+=2*j)
     movl -4(%ebp), %eax    // j->eax
     movl %eax, %edx        // j->edx
     leal 0(,%edx,2), %eax  // eax=2*edx
     movl %eax, -16(%ebp) // m=eax (m=2*j)
     incl -4(%ebp)          // j++
     jmp .L2
.L3
```

Optimizing compilers

Before optimization:

```
.L2
    movl -4(%ebp), %eax       // j -> eax
    cmpl -12(%ebp), %eax      // n <> eax ?
    jl .L4
    jmp .L3
.L4
    movl -4(%ebp), %eax// j-> eax
    movl %eax, %edx           // j->edx
    leal 0(,%edx,2), %eax     // eax=2*edx
    addl %eax, -8(%ebp)       // k += eax
    movl -4(%ebp), %eax// j->eax
    movl %eax, %edx           // j->edx
    leal 0(,%edx,2), %eax     // eax=2*edx
    movl %eax, -16(%ebp)      // m=eax
    incl -4(%ebp)             // j++
    jmp .L2
.L3
```

Compiler optimized version 1:

```
.L4
    leal (%edx, %eax, 2), %edx     // edx+=2*eax
    leal 0(,%eax,2), %ecx          // ecx=2*eax
    incl %eax                      // eax+=1
    cmpl %ebx, %eax                // n<>eax ?
    jl .L4
```

Compiler optimized version 2 (with IVS):

```
.L4:
    addl    $1, %ecx        // j++
    addl    %eax, %edx      // k+=m
    addl    $2, %eax        // m+=2
    cmpl    %r8d, %ecx      // n<>j ?
    jne  .L4
```

Optimizing compilers

- Contemporary compilers can have dozens of optimization options
  - examples (for *gcc*):
    - -fstrength-reduce, -fcse-follow-jumps, -ffast-math, -funroll-loops, -fschedule-insns, -finline-functions, -fomit-frame-pointer
  - important optimizations concern parallelization and vectorization
    - often in order to use particular optimizations for a given hardware (concerning e.g. vectorization) special options have to be passed explicitly to the compiler – e.g. **`-march=core-avx2`** – for cores with AVX2 instructions
    - often directives in source code help compilers to optimize
- In practice, most often compiler optimization is applied using options for optimization levels
  - typical levels and performed optimizations are:
    - -O0 – no optimization
    - -O1 – optimize for execution time and code size
    - -O2 – more optimization options applied, without sacrificing too much time and going into options that can alter the results of code execution
    - -O3 – the most aggressive optimization
    - (some compilers can have more levels, e.g. for vectorization, parallelization)