Compiler optimization and code generation

Compiler Optimization

- The compiler translates programs written in a high-level language to assembly language code
- The assembly language code is translated to object code by an assembler
- Object code modules are linked together and relocated by a linker, producing the executable program
- Code optimization can be done in all three stages, but the most significant optimizations are done in the compilation stage
  - what kind of optimizations are done depends on the compiler / assembler / linker that is used
  - there can be significant differences in optimization capabilities between different compilers
  - it is important to write clear and simple code that the compiler can analyze
  - avoid programming constructs that are difficult for the compiler to analyze, or that are known to be inefficient
The compilation process

- **Preprocessing**
  - simple textual manipulations of the source code
  - include files, macro expansions, conditional compilation

- **Lexical analysis**
  - source code statements are decomposed into tokens
  - (variables, constants, language constructs, ...)

- **Parsing**
  - syntax checking
  - source code is translated into an intermediate language form
  - creates a tree-based representation of the program
  - semantic analysis
  - error messages

- **Optimization**
  - several optimization phases are performed on the intermediate language form of the program

- **Code generation**
  - intermediate language form is translated to assembly language
  - assembler code is optimized

Intermediate language

- Expresses the same computation as the high-level source code
  - represented in a form that is better suited for analysis
  - also includes computations that are not visible in the source code, like address calculations for array expressions

- A high-level language statement is represented by several IL statements
  - IL is closer in complexity to assembly language than to a high-level language

- **Static Single-Assignment form**
  - a variable is assigned exactly once
  - a new version of the variable is created when it is modified
  - variables can be seen as pseudo-registers

```c
while (j<n) {
    k = k+j*2;
    m = j*2;
    j++
}
```
Basic blocks

- Basic blocks are regions of code with one entry at the top and one exit at the bottom
  - there are no branches in a basic block
  - generated from the tree-based representation built by the front end
- A flow graph describes the transfer of control between basic blocks
- Data dependence analysis
  - builds a directed acyclic graph (DAG) of data dependences
- The compiler both optimizes the code within basic blocks and across multiple basic blocks

```
A: t1 = j
t2 = n
t3 = (t1<t2)
jmp (B) t3
```

```
B: t4 = k
t5 = j
t6 = t5*2
t7 = t4+t6
k = t7
t8 = j
t9 = t8*2
m = t9
t10 = j
t11 = t10+1
j = t11
jump (C) TRUE
```

```
C: t1 = j
t2 = n
t3 = (t1<t2)
jmp (B) t3
```

Compiler optimization

- Most compiler optimization techniques optimize code speed
  - sometimes at the expense of code size
  - the user can choose what kind of optimizations to apply by compiler options (-O1, -O2, -O3, -Os)
- The basic optimization techniques are typically very simple
  - operate on code within one or a few basic blocks
  - try to reduce the number of instructions and memory references
- Loop optimizations operate across basic blocks
  - can move code from one basic block to another
- Advanced compilers also do optimization over larger units of code
  - interprocedural optimizations
- Peephole optimizations
  - replaces short instruction sequences (1-4 instructions) with more efficient alternatives
Register allocation

- Register allocation decides which values are stored in registers and which in memory
  - typically we have much more variables in a program than what can be kept in registers
- Register spilling
  - in general, all variables cannot be stored in registers
  - some values have to be stored in memory locations instead of in registers, called register spilling
  - may slow down the code because of frequent memory accesses
  - register allocation is less critical in processors with register renaming
- Register pressure
  - measures how much demand there is for allocating variables to registers

Register allocation (cont.)

- Register allocation is done in several stages
  - local register allocation
    - allocate registers to variables within a basic block
  - global register allocation
    - allocate registers to variables across an entire subprogram
  - interprocedural register allocation
    - allocate registers to variables and constants accessed by more than one subprogram
- A variable will not be stored in a register if there is a pointer to it
- Register storage class in C / C++
  - `register int ix, iy;`
  - advises the compiler that a variable will be heavily used and should be placed in a register
  - the compiler is free to ignore the advice
  - it is better to leave low-level code optimization, like register allocation, to the compiler
Live range

- A variable is live at a point in a program if its value will be used later in
  the program
  - the variable will be read in some subsequent instruction
  - i.e. it is used on the right hand side in an expression
- A variable is dead when the next reference to it is an assignment, or
  when there are no further references to it
- The compiler does a live range analysis of the variables to be able to
  do register allocation
- Rules for register allocation
  - allocate all of the variables to as few registers as possible
  - if two variables are live at the same point in the program, the variables
    must be allocated to different registers

Simple register allocation

- When a variable is seen for the first time
  it is allocated to a free register or a register containing a dead variable
- If no such register exists, select the
  register who’s use is furthest ahead,
  spill that register and allocate it to the
  new variable
  - register spilling means storing the
    variable in memory instead of a register
- We can execute the example code using
  only three registers
  - red – R0
  - green – R1
  - blue – R2
Register allocation by graph colouring

- Build an interference graph of the variables in a basic block
  - nodes represent variables
  - arc between two nodes if they are both live at the same time
- Two nodes that are alive at the same time cannot be allocated to the same register
- The problem is to find a coloring of the interference graph using $N$ colors
  - assign each node (=variable) a color (=register) so that any two connected nodes have different colors
- Optimal graph coloring is an NP-hard problem
  - have to use heuristic algorithms
  - cannot guarantee that we find an optimal solution

Compiler optimization

- The compiler applies transformations to the program code to make it more efficient to execute
  - which optimizations are used depends strongly on the architecture of the processor
- Similar transformations can also be applied to the source code by the programmer
  - high-level code optimization techniques
- The compiler must preserve the correctness of the program
  - it may not do any transformations that can alter the behaviour of the program
  - only applies transformations that are known to always produce similar results as the original code
Compiler optimization techniques

- Optimizations that improve assembly language code
  - reduce the number of instructions and memory references
  - use more efficient instructions or assembly language constructs
  - instruction scheduling to improve pipeline utilization

- Optimizations that improve memory access
  - keep variables in registers instead of memory
  - reduce cache misses
  - prefetching of data

- Loop optimizations
  - build larger basic blocks
  - remove branch instructions

- Function call optimization
  - eliminate function calls
  - pass arguments in registers instead of on stack

Constant folding

- Expressions consisting of multiple constants are reduced to one constant value at compile time

- Example:
  - two constants \( \pi \) and \( d \)
  - \( \text{tmp} = \pi / d \) is evaluated at compile time
  - the compiler uses the value \( \text{tmp} \) in all subsequent expressions containing \( \pi / d \)

- Explicitly declaring constant values as `const` helps the compiler to analyze the code
  - also improves code readability and structure

- Reduces the number of instructions

```c
const double Pi = 3.14159;
... 
d = 180.0;
... 
t = Pi*v/d;
... 
t = v*tmp;
```
Copy propagation and constant propagation

- Propagates values of variables or constants into the expressions where they are used
  - called constant propagation when applied to constants
- Example:
  - the second statement depends on the first
  - copy propagation eliminates the dependency
  - if \( x \) is not used in the subsequent computation, the assignment \( x = y \) can be removed (by dead code elimination)
- Reduces register pressure and eliminates redundant register-to-register move instructions

```
x = y;
z = c+x;
```

```
x = y;
z = c+y;
```

Common sub-expression elimination

- Replace sub-expressions that are evaluated more than once with a temporary variable
  - evaluate the sub-expression and store it in a temporary variable
  - use the temporary variable instead of the sub-expression
  - the sub-expression is computed once and used many times
- The associative order may be important
  - is it always correct to replace \((a+b)+c\) by \(a+(b+c)\)?
  - integer arithmetic is associative, floating-point arithmetic is not
- Often used to simplify address calculations in array indexing or pointer de-referencing

```
d = c*(a+b);
e = (a+b)/2;
tmp=a+b;
d = c+tmp;
e = tmp/2;
```
Dead code removal

- Removes code that does not affect the result of the computation
  - dead code is often produced as a result of other compiler optimizations
  - may also be introduced by the programmer
- Two types of dead code
  - instructions that are unreachable
  - instructions that produce results that are never used
    - for instance, computing a result into a non-global dead variable
- Dead code removal can completely change the behavior of simple synthetic benchmark programs
  - can eliminate a whole loop if the value computed in the loop is never used
- Reduces run time, reduces code size, improves instruction cache usage

```c
#define DEBUG 0
...
if (DEBUG) {
    /* debugging code */
    printf("...");
    ...
}
...
```

Strenght reduction

- Replace slow operations by equivalent faster ones
  - replace multiplication by a constant $c$ with $c$ additions
  - replace power function by multiplications
  - replace division by a constant $c$ with multiplication by $1/c$
  - replace integer multiplication by $2^n$ with a shift operation
  - replace integer division by $2^n$ with a shift operation, for positive values
  - replace integer modulo-2 division by masking out the least significant bit
- Some transformations may affect the precision of floating-point calculations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Replaced by</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x*2$</td>
<td>$x+x$</td>
</tr>
<tr>
<td>$x^2$</td>
<td>$x*x$</td>
</tr>
<tr>
<td>$x^2.5$</td>
<td>$x^{3/2}x$</td>
</tr>
<tr>
<td>$x/n$</td>
<td>$x*(1/n)$</td>
</tr>
<tr>
<td>$k*2^n$</td>
<td>$k&lt;&lt;n$</td>
</tr>
<tr>
<td>$k/2^n$</td>
<td>$k&gt;&gt;n$</td>
</tr>
<tr>
<td>$k%2$</td>
<td>$k&amp;1$</td>
</tr>
</tbody>
</table>
Induction variable optimization

- An induction variable is a variable that is increased or decreased by a fixed amount on every iteration of a loop
  - can also be a linear function of another induction variable
- Simplify expressions that change as a linear function of the loop index
  - the loop index is multiplied with a constant value in each iteration
  - can replace the multiplication with an addition
- Can often be applied on address calculations when iterating over an array
  - for instance when accessing array elements which are located at a fixed distance from each other

```
for (i=0; i<N; i++) {
    k = 4*i+m;
    ... 
}
```

```
k=m;
for (i=0; i<N; i++) {
    ...
    k=k+4;
}
```

Loop invariant code motion

- Move calculations that do not change between loop iterations (i.e. loop invariant code) out of the loop
- Often used to eliminate load- and store operations from loops in generated assembly language code
- Hoisting
  - move invariant code before the loop
  - example: in the generated assembly code, the value of y is loaded into a register before the loop (not in each iteration)
- Sinking
  - move invariant code after the loop
  - example: compute the result in the register and store the computed value of s after the loop (not in each loop iteration)
- Can eliminate unnecessary loads/stores of intermediate results

```
for (i=0; i<N; i++)
{
    y = sqrt(z);
    X[i] = X[i]*y;
}
```

```
y = sqrt(z);
for (i=0; i<N; i++)
{
    s = s*X[i];
}
```
Loop unswitching

- Move loop-invariant conditional constructs out of the loop
  - if- or switch-statements which are independent of the loop index can be moved outside of the loop
  - the loop is instead repeated in the different branches of the if- or switch-statement
  - removes branch instructions from within the loop
- Removes branch instructions, increases instruction level parallelism, improves possibilities to parallelize the loop
  - but increases the amount of code

```c
for (i=0; i<N; i++)
{  if (a>0)
   X[i] = a;
   else
   X[i] = 0;
}
```

Loop fission and loop fusion

- Loop fission breaks up a complicated loop into multiple smaller loops, iterating over the same index range
  - may achieve better locality of reference and reduce register pressure

```c
for (i=0; i<N; i++)
{  A[i] = 0.0;
   B[i] = 1.0;
}
```

- Loop fusion combines multiple loops operating over the same index range into one single loop
  - may increase register pressure
  - may lead to better locality of reference, for instance if both original loops refer to the same memory locations

```c
for (i=0; i<N; i++)
{   A[i] = 0.0;
   for (i=0; i<N; i++)
   {   B[i] = 1.0;
   }
```
Loop peeling

■ A small number of iterations from the beginning and/or end of a loop are removed and executed separately, outside the loop
  – makes the behavior of the loop more regular
■ Often used in handling boundary conditions
  – values along the edges of a matrix may need to be treated differently
■ Example:
  – a is 10 only in the first iteration, after that it is set to i
  – x[a] then refers to the value of a in the previous iteration, i.e. a[i-1]
■ Simplifies the loop body
  – more possibilities to apply further compiler optimizations

```
int a = 10;
for (i=0; i<10; ++i){
y[i] = x[i] + x[a];
a = i;
}
y[0] = x[0] + x[10];
for (int i=1; i<10; ++i){
y[i] = x[i] + x[i-1];
}
```

Loop unrolling

■ Replicate the loop body k times and increase the loop counter with k
  – k is called the unrolling factor
■ Reduces loop overhead and branch instructions
■ Produces larger blocks of linear code (without branches)
  – increases instruction level parallelism
  – more opportunities for instruction scheduling
■ Increases code size and register usage
■ When applied on source code this makes the code difficult to read and maintain

```
#include <limits>

/* Copy Y to X */
for (i=0; i<N; i++) {
  X[i] = Y[i];
}

/* Copy Y to X */
limit = N-4;
for (i=0; i<limit; i+=5) {
  X[i] = Y[i];
  X[i+1] = Y[i+1];
  X[i+2] = Y[i+2];
  X[i+3] = Y[i+3];
  X[i+4] = Y[i+4];
}

/* Do the last elements */
for (; i<N; i++) {
  X[i] = Y[i];
}
```
Procedure inlining

- Replace a function call by the body of the function
  - eliminates the overhead of the function call and return
  - improves possibilities for further compiler analysis and optimization by building larger basic blocks
- Also called in-line expansion
- Increases code size
  - the compiler has an upper limit on the size and complexity of functions that it will inline
- In C/C++, functions can be defined with the `inline` keyword
  - hint to the compiler that the function should be inlined
  - inlining can also be implemented using macros

```c
double max(double a, double b) {
    return ((a>b) ? a : b);
}
```

```c
... for (i=0; i<N; i++) {
    Z[i] = max(X[i], Y[i]);
}
```

- Increases code size
  - the compiler has an upper limit on
  - the size and complexity of functions that it will inline
- In C/C++, functions can be defined with the `inline` keyword
  - hint to the compiler that the function should be inlined
  - inlining can also be implemented using macros

```c
void foo(int a[N], int *b) {
    for (int i=0; i<N; i++) {
        a[i] = *b+1;
    }
}
```

Pointer aliasing

- When variables are accessed through pointers the compiler can not in general know whether there are multiple pointers to the same location
  - called pointer aliasing
  - a value accessed through a pointer may be updated in each loop iteration
- Example:
  - the pointer `b` could point to an element in `a`, which is updated in an iteration
  - the compiler can not assume that the expression `*b+1` is loop invariant
  - thus, it can not move the expression out of the loop, but it has to be re-evaluated in each iteration
- Can use the keyword `restrict` (or `__restrict__`) to specify that a pointer has no aliases
  - it is up to the programmer to design the program so no aliasing occurs
  - often used to enable vectorization of loops
Example: GCC

- The GNU Compiler Collection comprises a number of compilers for different programming languages
  - C, C++, Objective-C, Objective-C++, Java, Fortran and Ada
- Contains several separate components
  - preprocessor cpp
  - compiler gcc
  - assembler as
  - linker ld
- The compiler operates in three stages
  - the front-end is language dependent, the later stages are language independent
  - there is one back-end for each supported architecture
  - transforms the program to different intermediate tree-based representations

![Diagram of GCC process]

GCC front end

- The front-end reads and validates the syntax of the program
  - reads in the source program as a stream of characters
  - lexical analysis and parsing
  - checks the syntax of the program
  - constructs an Abstract Syntax Tree (AST) representation for each function of the program
  - performs data type analysis, data types are attached to tree nodes

Example: Abstract Syntax Tree for the expression x = y-z*3

![Abstract Syntax Tree]
Front end structure

- The front end depends on the language that is used
  - uses different AST representations for each language
  - C, C++, Fortran77, Fortran95, Java, …
- The AST representation is transformed into another intermediate format, called the GENERIC tree representation
  - unified tree representation for all languages

Front end

```
C tree
C++ tree
GENERIC
Gimplify
GIMPLE
```

GCC middle end

- Gimplification
  - the GENERIC representation is converted to another intermediate form, called GIMPLE
  - the GIMPLE AST format is a subset of the GENERIC format, and is in SSE form
  - better suited for compiler optimization
- The middle end performs target-independent compiler optimizations on the intermediate program representation
  - performs control-flow and data-flow analyzes and tries to apply code optimization techniques
- The goal is to transform the program into an equivalent but more efficient version
  - run time optimization
  - space optimization

Middle end

```
GIMPLE
Tree SSA optimization
RTL
```
Compiler optimizations in the middle end

- **Algebraic simplifications**
  - simplifications based on the algebraic properties of data
  - for instance, \( i+1-i \) can be transformed into the constant value 1
  - can also use associativity, commutativity and distributivity to simplify expressions
  - less simplifications for floating-point expressions than integer expressions

- **Constant folding**
  - expressions where all operands are constants are evaluated at compile time and replaced by a constant value
  - for instance \( a = 4+3-8 \) can be replaced with \( a = -1 \)

- **Constant propagation**
  - values of constants are propagated into the expressions where they are used
  - for instance \( x=3; y=x+4; \) can be replaced with \( x=3; y=7; \)

- **Redundancy elimination**
  - common sub-expression elimination
    - sub-expressions that are computed repeatedly, and that are not modified, are replaced with the value of the sub-expression
    - for instance \( a = b\cdot c+g; d = b\cdot c\cdot d; \) can be replaced by \( \text{tmp} = b\cdot c; a = \text{tmp}+g; d = \text{tmp}\cdot d; \)
  - loop-invariant code motion
    - expressions inside loops that are independent of the loop iteration, but always evaluate to the same result, are moved out of the loop
    - for instance
      ```c
      for (i=0; i<n; i++){
          x = y*z;
          a[i] = 6*i+x*x;
      }
      
      x = y*z;
      t1 = x*x;
      for (i=0; i<n; i++){
          a[i] = 6*i+t1;
      }
      ```
Compiler optimizations in the middle end

- Redundancy elimination (continued)
  - partial redundancy elimination
    - eliminates expressions that are computed more than once on some, but not all, execution paths through the program
    - combines both loop-invariant code motion and common sub-expression elimination

- Example:
  - the computation of $x+20$ is partially redundant, since it is computed twice if $\text{cond}$ is TRUE
  - after the optimization it is only computed once

```c
if (cond) {
    y = x+20;
    tmp = y;
} else {
    {other code}
    tmp = x+20;
}
z = tmp;
```

- Dead code removal
  - removes obviously unused code constructs
  - simplify if-statements with constant conditions (TRUE or FALSE)

- Generating code for OpenMP threaded execution
  - only if the compiler flag `-fopenmp` is used

- Build a control flow graph for each function
  - identifies the basic blocks and builds the control flow graph
  - identifies all variables that are referenced
  - warns for uninitialized variables

- Convert the code to Static Single Assignment (SSA) form

- Tail recursion elimination
  - transforms tail recursion into loops
Compiler optimizations in the middle end

- Loop optimizations
  - loop invariant motion
    - moves code that is independent of the loop out of the loop
  - induction variable optimizations
    - optimizations for variables that are increased or decreased by a fixed amount on every iteration of a loop
    - strength reduction, induction variable merging, induction variable elimination
  - loop unswitching
    - moves invariant conditional expressions out of the loop
  - vectorization
    - transforms loops to operate on vector data types (SSE or AVX) instead of scalar data types
  - auto-parallelization
    - divides the iteration space of loops into a number of parallel threads

- Tail call elimination
  - replaces function calls with jumps

- Check and warn for function return without a value
  - identifies and warns for non-void functions without a return value

- Return value optimization
  - tries to avoid unnecessary memory copies for return values

- Loop nest optimization
  - tries to improve data locality for array traversal in loop nests
  - loop interchange, skewing and reversal

- Remove empty loops
  - remove loops with no code body

- Unrolling of small loops
  - loops with few iterations are completely unrolled

- Function inlining
  - function calls are replaced by the body of the function
Compiler optimizations in the middle end

- Predictive commoning
  - tries to reuse computations from previous iteration of a loop, especially values of loads and stores
  - values are stored in a set of temporary variables that is rotated at the end of the loop
  - the loop can then be unrolled and the loop bodies rewritten to use the correct temporary variables

- Array prefetching
  - inserts prefetch instructions for array references in loops

- After the tree optimizations, the program is transformed into the Register Transfer Language (RTL) format
  - RTL is a low-level representation of the program, closer to assembly language than the previous representations
  - abstract assembly language for a machine with an infinite number of registers

Back end

- The back end performs several passes of optimization on the RTL representation
- Finally the program is transformed into assembly language for the target system
  - uses a machine description of the target system
- Performs register allocation and code scheduling
  - maximizes the number of variables assigned to registers
  - rearranges instructions to use the pipeline efficiently
Compiler optimizations in the back end

- Generates code for exception handling
- Control flow graph cleanup
  - removes unreachable code
  - simplifies jumps to the next statement and jumps to other jumps
- Copy propagation and addressing mode selection
- Common sub-expression elimination within basic blocks
  - also optimizes addressing modes based on their costs
- Global common sub-expression elimination
  - different CSE optimizations depending on whether optimizing for speed or memory
- Loop optimizations
  - same kind of loop optimizations as in the middle end

Compiler optimizations in the back end

- Jump bypassing
  - simplifies the control flow graph by propagating constants into conditional branches
- If-conversion
  - replaces conditional branches with compare and conditional move instructions
- Instruction combination
  - tries to combine groups of 2-3 instructions that are related by data flow into a single instruction
- Modulo scheduling
  - examines inner loops and reorders instructions by overlapping different iterations of the loop
- Instruction scheduling
  - reorders instructions within basic blocks to reduce pipeline stalls
  - uses information about instruction latency and throughput
Compiler optimizations in the back end

- **Register allocation**
  - all references to pseudo registers are replaced by physical registers, constant expressions or memory locations on the stack
  - uses a heuristic graph coloring algorithm
  - decisions about register spilling is based on the live range for the variables
  - generates function entry and exit code sequences
  - also inserts code to save and restore registers that have been clobbered by function calls

- **Register-to-stack conversion**
  - arranges references to floating-point registers to operate on the fp register stack (if the x87 fp-unit is used)

- **Final assembly code generation**
- **Output debugging information, if the debug flag is on**

Compiler optimization switches in GCC

- **Switches for controlling the optimization level in gcc**
  - If no optimization switch is used, the compiler will not perform any optimization on the code
  - tries to reduce the cost of compilation and make debugging produce the expected result
  - the generated assembly code will correspond closely to the high-level code, which is good for debugging
  - without any optimization only variables declared with the `register` keyword will be placed in registers

- **-O, -O1**
  - tries to reduce code size and execution time, but does not use any optimizations that make the compilation time longer
  - function call arguments that have been placed on the stack are not popped immediately, but they are accumulated and popped later
  - no frame pointer kept in register for functions that doesn’t need it
Compiler optimization switches in GCC (cont.)

- **-O2**
  - adds compiler optimizations that do not involve space/speed trade-offs
  - combines threaded jumps (branches to further branch instructions)
  - performs function inlining for small and simple functions
  - aligns branches and function calls in memory
  - allows the compiler to assume that two pointers of different data types never point to the same address (i.e. that they are not aliases)
  - increases compile time and performance, compared to -O1

- **-O3**
  - even more optimization, also optimizations that may increase code size
  - uses function inlining and loop vectorization with SSE or AVX instructions

- **-Os**
  - optimize for code size
  - enables all -O2 optimizations that do not increase code size
  - also some optimizations designed to reduce code size

Options controlling optimization in gcc

- gcc can be given options to turn on or off specific compiler optimizations

- Compiler options with a -f prefix are machine-independent and options starting with -m are machine-specific
  - -m32 generates 32-bit code for the IA-32 ISA
  - -m64 generates 64-bit code for the x86-64 ISA

- An -On switch turns on a predefined set of machine-independent optimizations

- For instance, -O1 turns on the following optimization flags:
  - -fauto-inc-dec, -fbranch-count-reg, -fcombine-stack-adjustments, -fcompare-elim,
  - -fcprop-registers, -fcse, -fdelayed-branch, -fdse, -fforward-propagate,
  - -fguess-branch-probability, -ffconverson2, -ffconverson, -fflinel-functions-called-once,
  - -fipa-pure-const, -fipa-profile, -fipa-reference, -fmerge-constants, -fmove-loop-invariants,
  - -fshrink-wrap, -fsplit-wide-types, -ffree-bit-cpp, -ffree-cpp, -fssas-piopt, -ffree-ch, -ffree-copy-prop,
  - -ffree-copyrename, -ffree-dce, -ffree-dominator-opt, -ffree-dse, -ffree-fordprop, -ffree-fre,
  - -ffree-hiprop, -ffree-sink, -ffree-iar, -ffree-ira, -ffree-pla, -ffree-ter, -funit-at-a-time
Options controlling floating-point arithmetic

- **-ffast-math**
  - allows the compiler to violate some aspects of the IEEE fp standard
  - enables some unsafe optimizations for floating-point computations
  - assumes that no floating-point values are NaNs
  - assumes that arguments to square root are non-negative
  - assumes that arithmetic operations are associative
- Not enabled by –O3, because it may cause results in floating-point computations that are not compliant with the IEEE fp-standard

- There is a compiler switch -Ofast, which enables all –O3 optimizations and some other optimizations that may not be compliant with the floating-point standard
  - turns on -ffast-math

Options for pointer validity checking

- **-fsanitize=address**
  - instrument all pointer or array dereferencing operations with code for range and validity checks
  - used to detect out-of-bounds memory accesses and use of deallocated memory
  - causes overhead on the execution, both execution time and memory usage increases
- Prints out error messages for memory accesses that are out of range when the program is executed

- There are other tools that can be used to detect memory references
  - for instance Valgrind
Specifying the processor architecture

- `–march=native | i686 | pentium3 | pentium4 | core2 | corei7 | haswell | k6 | athlon | opteron | barcelona | …`
  - generates instructions for the specified processor type, and also optimizes the generated code for this processor
  - the default value is `–march=native`, which generates code for the same architecture as the compiler runs on
  - if you distribute code compiled with `–march=native` it may not work on older architectures

Programs and libraries that are distributed in binary format may be compiled for an old architecture
- often compiled for the i686 architecture (Pentium Pro with the P6 microarchitecture)
- otherwise they may not run on all processors
- may not for instance use conditional moves, floating-point operation with the SSE/AVX unit or vectorization
- it is often a good idea to compile the library specifically for the architecture it will be used on

Options to specify the architecture

- Some of the options for `–march` to specify the processor architecture
  - check the manual page for `gcc` to see all options (there are a lot of them)

<table>
<thead>
<tr>
<th>Option</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>native</td>
<td>Generate code for the architecture the compiler runs on</td>
</tr>
<tr>
<td>i386, i486, i586</td>
<td>Original Intel 386, 486 or Pentium processor (no SSE/AVX instructions)</td>
</tr>
<tr>
<td>i686</td>
<td>Intel Pentium Pro (P6 microarchitecture)</td>
</tr>
<tr>
<td>corei7</td>
<td>Core i7 with 64-bit extensions, SSE vectorization</td>
</tr>
<tr>
<td>haswell</td>
<td>Intel 64-bit Haswell with SSE2 vectorization and FMA instruction</td>
</tr>
<tr>
<td>k6</td>
<td>AMD K6 processor with MMX vectorization</td>
</tr>
<tr>
<td>opteron</td>
<td>AMD K8 core with 64-bit extensions, SSE vectorization</td>
</tr>
<tr>
<td>barcelona</td>
<td>AMD Tiber family core, 64-bit extensions, SSE vectorization</td>
</tr>
</tbody>
</table>
Options to select floating-point unit

- **mfpmath** = 387 | sse
  - selects either the x87 floating-point unit or the SSE/AVX vector unit for floating-point computations
  - the default for 32-bit code is to use the x87 fp-unit
  - the default for 64-bit code is to use the SSE/AVX unit

Floating-point operations with the SSE/AVX unit use scalar instructions, unless they are also vectorized
- i.e., SSE instructions that only operate on one scalar fp value instead of a vector of values

SSE and AVX support

- **-mmsx**, **-msse**, **-msse2**, **-mavx**, **-mavx2**
  - allow the compiler to generate instructions in the specified extension
  - requires **-march=cputype**, where cputype supports the extension
  - does not perform automatic vectorization of loops, just enables the compiler to generate code for the specified extension
  - when AVX is enabled (by **-mavx** or **-mavx2**) the compiler does not generate any SSE-instructions

- **-ftree-vectorize**
  - perform automatic loop vectorization using SSE or AVX operations
  - this flag is enabled by **-O3**

- **-fopt-info-optimized**
  - print out diagnostic messages output about optimization, including vectorized loops

- **-ftree-parallelize-loops=n**
  - try to parallelize loops, i.e. split their iterations between n threads
  - only supported on systems that support Pthreads
Inspecting compiler-generated assembly language

- The best (and sometimes the only) way to find out what optimizations the compiler does is to look at the generated assembly code.
  - need to be able to read assembly language code
- Compile the code you want to examine with a low optimization switch (-O1 or -O2).
  - the highest optimization level (-O3) may do very aggressive optimizations, including function inlining, loop unrolling and automatic vectorization.
  - compiler optimizations make it difficult to relate the generated assembly code to the original source code.
- Can look up assembly language instructions in Intel’s manuals.
- Gnu compilers use AT&T syntax, the manuals use Intel’s syntax.

Assembly Language Syntax

- There are two different assembly language versions.
  - AT&T syntax – used in the Gnu assembler (as)
  - Intel syntax – used in Intel’s assembler and documentation.
- Opposite order of source and destination operands.
  - Gnu: mov source, dest ; source → dest
  - Intel: mov dest, source
- AT&T and Intel use different ways of specifying operand sizes.
  - Gnu: uses an instruction mnemonic suffix to specify the size
    - movl x, %eax
  - Intel: uses a prefix for memory operands: byte ptr, word ptr, dword ptr
    - mov eax, byte ptr x

<table>
<thead>
<tr>
<th>C declaration</th>
<th>as suffix</th>
<th>Size (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int, unsigned</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long int, unsigned long</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>all pointers</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>d</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>t</td>
<td>16</td>
</tr>
</tbody>
</table>
Assembly Language Syntax (cont.)

- Instructions operate on zero, one or two operands
  - only one of the operand may be a memory reference
- Operands can be
  - immediate – add $0x8,%rsp
  - registers – add %rax,%rbx
  - a memory location – mov 0x8(%rbp),%rsi
- Gnu: registers are prefixed by %, immediate operands prefixed by $
  - add $4, %eax
- Intel: no prefixes for registers or immediate operands
  - add eax, 4
- Different syntax for address expressions
  - Gnu: DISP(BASE, INDEX, SCALE)
    - movl -4(%ebp), %eax
  - Intel: [BASE + INDEX*SCALE + DISP]
    - movl eax, [ebp - 4]

Memory addressing

- Memory addresses can be specified in several ways
- Can be given as a base, index, displacement and a scale
  - mov $0x8(%rsi,%rdx,4),%rax
  - scale can be 1, 2, 4 or 8 bytes
  - used to specify addresses to elements in arrays or structures
  - any of base, index, displacement or scale can be missing
  - the effective address is computed as base + scale*index + displacement

- Example: add 8(%rdi,%rdx,4),%eax

<table>
<thead>
<tr>
<th>Base</th>
<th>Index</th>
<th>Displacement</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effective address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Inspecting assembly code generated by the compiler

- We will look at a number of code structures and see what assembly language code is generated
  - the example code is compiled on an Intel Xeon running Linux with gcc 4.8.1
- We write the code as small procedures, which are separately compiled
  - this avoids including unnecessary code in the assembly language (for instance code for the main program and input/output)
  - compile with gcc -c -g -O2 prog.c
- Use objdump to look at the assembly language code
  - objdump -d -S prog.o
- On Mac OS X you can use otool to examine object code
- On Windows with Microsoft Visual C++ you can use DUMPBIN

Calling conventions

- A set of rules, or conventions, describe the interface between a calling procedure (the caller) and the called procedure (the callee), including:
  - where the parameters are placed when the procedure is called (in registers or on the stack)
  - which registers are used for passing parameters and results
  - the order in which parameters are passed
  - who is responsible for cleaning up registers and the stack after the call
  - name mangling or name decoration – how identifiers are named in assembly
- For 32-bit x86 systems there are many alternative calling conventions
  - differs between operating systems, compilers and processors
  - cdecl – default for C programs and global functions in C++
  - stdcall – used by Win32 API
  - thiscall – used in C++
  - fastcall – many different variants exist
- On 64-bit x64 systems there are two main standards, one for Linux and one for Windows
x86-64 calling conventions on Linux

- For 64-bit x86-64 Unix-based systems the calling conventions are defined in the System V AMD64 Application Binary Interface (ABI)
  - the default calling convention for C/C++ programs on the x86 architecture in Linux is called cdecl
  - calling conventions on Windows is different (uses only 4+4 registers for arguments)
- Procedure arguments and return values can be passed in registers
  - only passed on the stack if a procedure has more arguments then the number of available registers, or if the arguments have a size larger than the registers
- 6 integer registers and 8 XMM registers can be used for passing arguments
  - integer arguments are passed in RDI, RSI, RDX, RCX, R8, R9 (in this order)
  - floating-point and vector arguments are passed in xmm0–xmm7 (or ymm0–ymm7)
  - if a procedure has more arguments than this, or if their types don’t fit in a register, they are passed on the stack
- For procedures where all arguments and local variables can be held in registers, no stack frame is generated
  - avoids copying of arguments between stack and registers
- For more complex procedures, a stack frame is generated
  - for instance if it calls further procedures

Register saving and returned values

- RAX, RCX, RDX, RSI, RDI and XMM0–XMM15 are caller saved
  - the calling procedure saves these values before a procedure is called, and restores them after
  - the called procedure can freely use these registers
- The other registers (RBX, RBP, R12–R15) are callee saved
  - the called procedure has to save these on the stack if it modifies them
- Results are passed back to the calling procedure in RAX and RDX
  - XMM0 and XMM1 for floating-point and vector data types
- Structures and classes are returned in memory
  - RAX contains a pointer to the memory location
Example: adding two integer values

- The code is compiled with optimization on
  - `gcc -O2 -c -g add.c`
- The first argument (a) is passed in the register RDI and the second (b) in RSI
  - the Load Effective Address (LEA) instruction is used for the addition, computes \( RSI + RDI \times 1 \)
  - the result is placed in EAX and returned to the caller

```c
int add (int a, int b) {
    return a+b;
}
```

| 0: 8d 04 37 | lea (%rdi,%rsi,1),%eax | Add a and b into eax |
| 3: c3        | retq                  | Return               |

- If we compile without any optimization, a stack frame is created
  - the arguments a and b are pushed on the stack

```c
int add (int a, int b) {
    0:   55          push   %rbp          # Save old base pointer
    1:   48 89 e5    mov    %rsp,%rbp     # New base pointer
    4:   89 7d fc    mov    %edi,-0x4(%rbp) # Put a on the stack
    7:   89 75 f8    mov    %esi,-0x8(%rbp) # Put b on the stack
    return a+b;
    a:   8b 55 fc    mov    -0x4(%rbp),%edx # Copy a to edx
    d:   8b 45 f8    mov    -0x8(%rbp),%eax # Copy b to edx
    10:  01 d0       add    %edx,%eax    # Add a and b
    }                     
    12:  5d          pop    %rbp          # Remove saved base pointer
    13:  c3          retq              # Return
```

Same example, without optimization

- Compile the code without any optimization
  - `gcc -c -g add.c`
- The arguments are passed in the registers RDI and RSI
  - a stack frame is created and the arguments pushed on the stack
  - the arguments are then copied to registers and added
### Example: comparing two integer values

- The first argument \((a)\) is passed in the register EDI and the second \((b)\) in ESI
  - compare \(b\) to \(a\) and set flags
  - move the value of \(b\) to the result register EAX
  - conditionally move \(a\) to EAX if \(a \geq b\)
  - RETQ transfers control to an address located on the top of the stack
  - the return address has been pushed on the stack by a CALL instruction

```c
int max (int a, int b) {
    if (a>b) return a;
    else return b;
}
```

### Example: adding elements in an array

- The address of \(a\) is passed in EDI and the length in ESI
  - returns a single integer in EAX as the result
  - \(sum\) is in EAX
  - loop counter \(i\) is in EDX
- This code is compiled with –O1
  - with –O3 the compiler vectorizes and unrolls the loop

```c
int addvec (int *a, int len) {
    int i, sum = 0;
    for (i=0; i<len; i++)
        sum +=a[i];
    return sum;
}
```
Example: adding floating-point values

- Function that adds together two arrays of floating-point values
  - generates floating-point code for the SSE unit
- The address of a is in EDI, address of b in ESI, address of c in RDX and length in RCX

```c
void addarray (float *a, float *b, float *c, int length) {
    int i;
    for (i=0; i<length; i++) {
        c[i] = a[i]+b[i];
    }
}
```

Passing parameters on the stack

- If the number of parameters is large, some have to be passed on the stack
- Example: a procedure with 8 integer arguments
  - the 6 first arguments (a–f) are passed in registers
  - the two remaining (g and h) have to be passed on the stack
  - the local variables xx, yy and zz can be kept in registers

```c
int myfunc (int a, int b, int c, int d, int e, int f, int g, int h) {
    int xx, yy, zz;
    xx = a*b*c*d*e*f*g*h;
    yy = a+b+c+d+e+f+g+h;
    zz = xx+yy;
    return zz+3;
}
```
Parameters on the stack

- The 6 first integer parameters are passed in registers
- The two remaining parameters are pushed on the stack in reverse order
  - they can be popped from the stack in order of declaration
  - after that, the return address is pushed on the stack
- The arguments g and h are referred to relative to the stack pointer
  - g is at address RSP+8
  - h is at address RSP +16 (in hexadecimal notation RSP+0x10)
- Red zone
  - 128 bytes of space reserved for the procedure

Generated code

- Compiled with –O1

```
0:   44 8b 54 24 08   mov   0x8(%rsp),%r10d   Move g to r10
5:   8b 44 24 10      mov   0x10(%rsp),%eax  Move h to eax
9:   41 89 fb         mov   %edi,%r11d   Move a to r11
c:   44 0f af de      imul  %esi,%r11d   r11 = r11*b
10:  44 0f af da      imul  %edx,%r11d   r11 = r11*c
14:  44 0f af d9      imul  %ecx,%r11d   r11 = r11*d
18:  45 0f af d8      imul  %r8d,%r11d   r11 = r11*e
1c:  45 0f af d9      imul  %r9d,%r11d   r11 = r11*f
20:  45 0f af da      imul  %r10d,%r11d   r11 = r11+g
24:  44 0f af d8      imul  %eax,%r11d   r11 = r11+h
28:  01 f7            add   %esi,%edi   edi = a+b
2a:  01 fa            add   %edi,%edx   edx = a+b+c
2c:  01 d1            add   %edx,%ecx   ecx = a+b+c+d
2e:  41 01 c8          add   %ecx,%r8d   %r8 = a+b+c+d+e
31:  45 01 c1          add   %r8d,%r9d   %r9 = a+b+c+d+e+f
34:  45 01 ca          add   %r9d,%r10d   %r10 = a+b+c+d+e+f+g
38:  44 01 d0          add   %r10d,%eax   eax = a+b+c+d+e+f+g+h
3a:  41 8d 44 03 03 lea   0x3(%r11,%rax,1),%eax   eax = eax+r11+3
3f:  c3             retq
```
Stack frames

- If no optimization switch is given, the compiler generates a stack frame for procedures
  - the register RBP is used as a frame pointer and can not be used for anything else
- Omitting the frame pointer frees the RBP register for general use
  - a stack frame is not strictly needed, arguments can be addressed relatively to the stack pointer (as we have seen)
  - can force the compiler to generate stack frames with the flag `-fno-omit-frame-pointer`
- Example of a procedure entry and exit code with a stack frame

```
pushq %rbp  # Push old base pointer
movq %rsp,%rbp # Set new base pointer
...
popq %rbp   # Restore old base pointer
ret        # Return
```