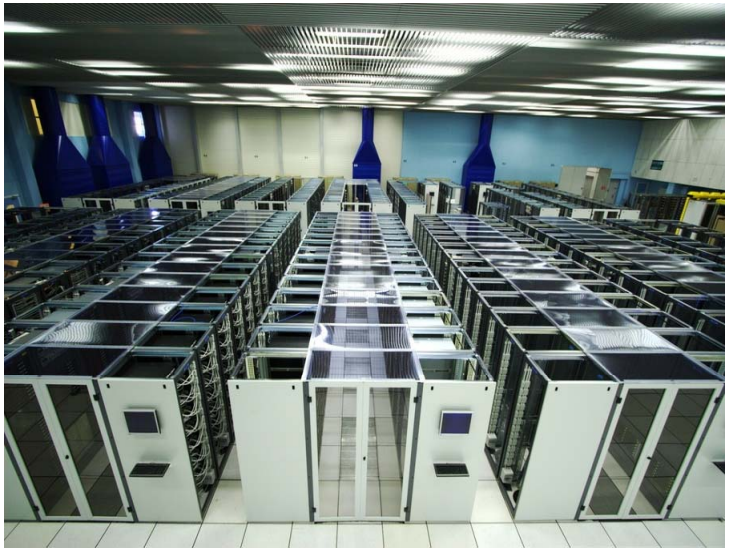
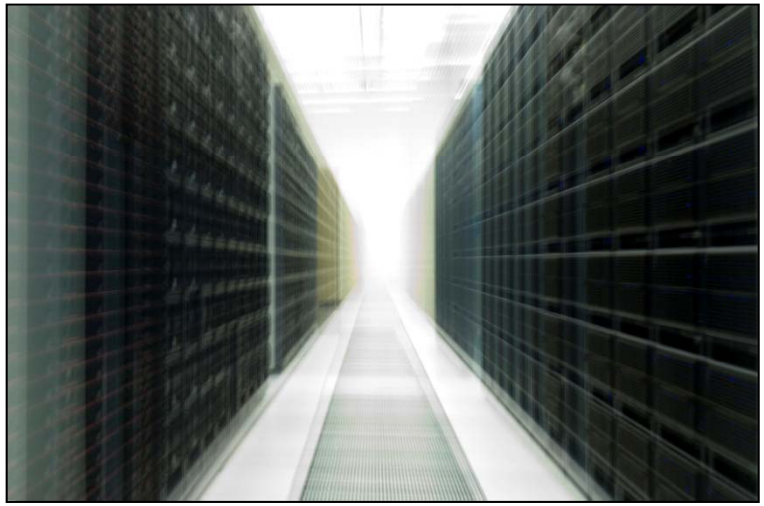
Compilers

Performance optimization
[Floating-point representation]
Vectorization



Sverre Jarp
CERN
openlab
CTO



Bertinoro – 12-17 October 2009

Overview

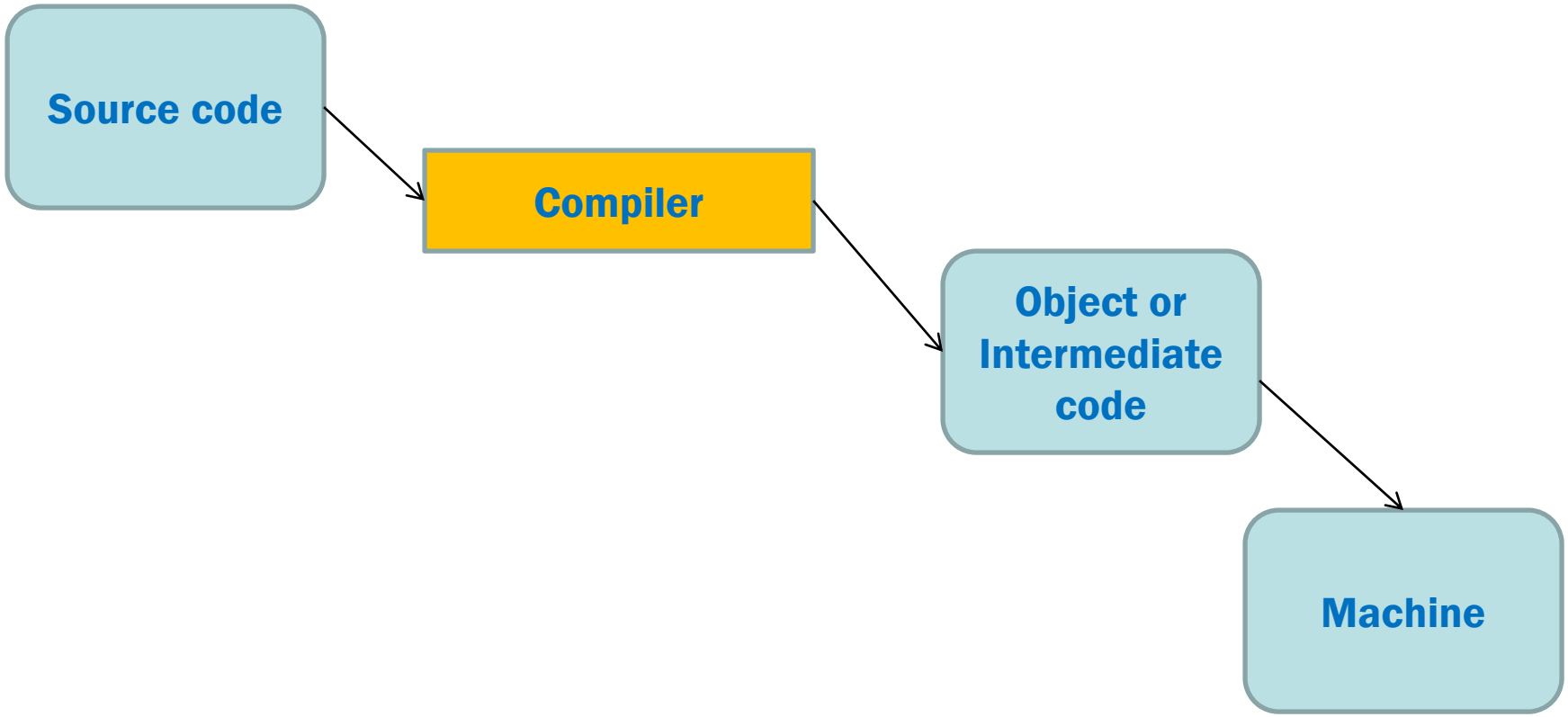
- Compiler theory
  - Front-end, Back-end
- Compilers in practice
  - Manufacturers (open source or not)
  - Working with compilers
    - Correctness and performance
- [Floating-point representation]
- Vectorization

Why are compilers important?

- "Why should we care about compilers? The compiler is just a tool…"
  - The compiler is NOT just a tool
  - It has the entire responsibility for telling the computer what you are trying to do…
    - ..using an archaic language:
      - movsd .LC0(%rip), %xmm0
      - movapd %xmm0, %xmm1
- Knowledge of the compilation process can help programmers produce better code
- Very important to know what the compiler can do for you (and what it can't)

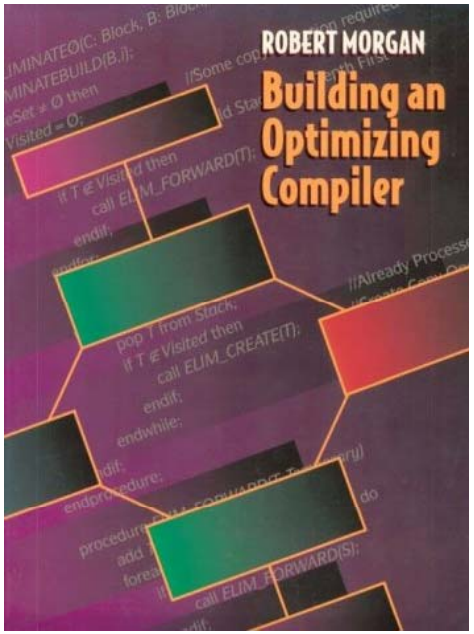
`const vs. #define`

What is a compiler?



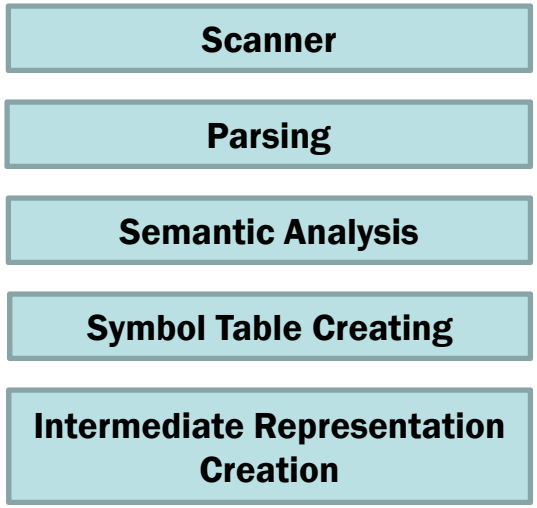
Compiler Theory

from “Building an Optimizing Compiler” by Robert Morgan



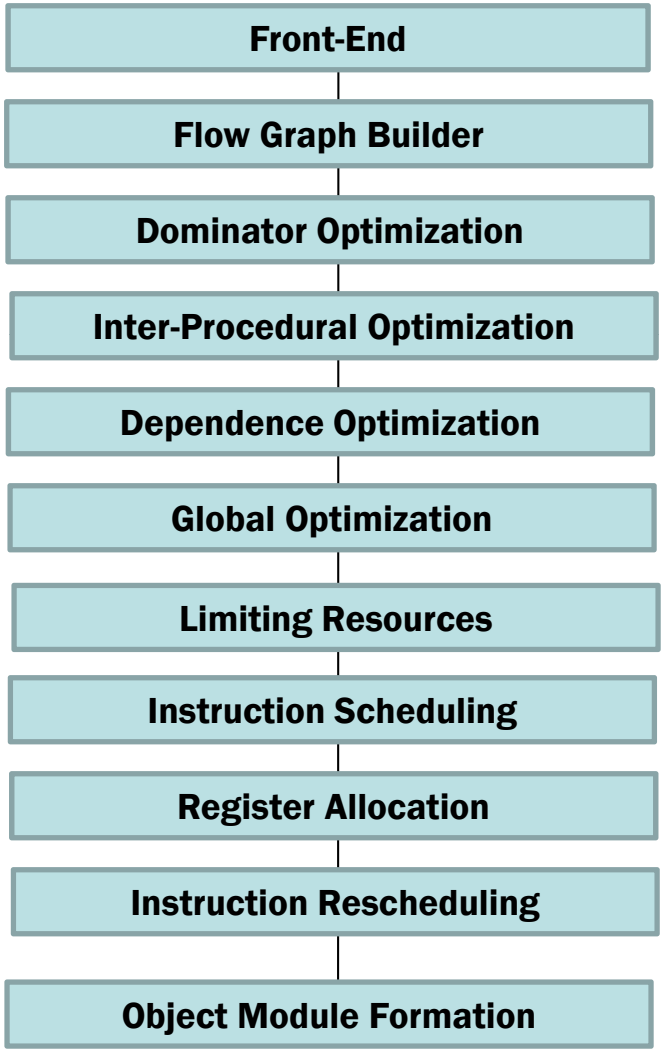
Compiler Front-End

- Language-specific
- Performs all lexical analysis, parsing, and semantic checks.
- Builds an abstract syntax tree and symbol table
- The initial optimization phase builds the flow graph or intermediate representation (IR)
  - Each node in the flow graph represents a “basic block”
    - Straight-line piece of code



Compiler Back-End

- Backend is comprised of several phases that will gradually lower the intermediate representation to assembler code

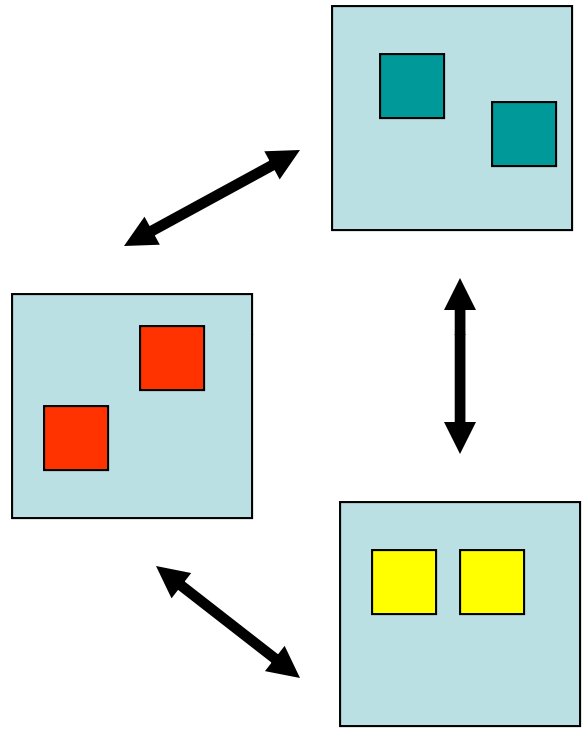


Actions by the optimization phases

- Typical optimization steps
  - Identify
    - Values that are constants
    - Computations known to have the same value
    - Instructions having no effect on the results of the program
  - Identify and eliminate
    - Redundant computations
  - Optimize
    - load and store operations
    - Loops:
      - Interchange indexes or unroll loops
      - Other advanced loop transformations
  - Perform
    - Code motion, strength reduction and dead-code elimination

Optimization phases (cont'd)

- Interprocedural Optimization phase
  - Analyses procedure calls within all the flow graphs of all procedures within the whole program (or library)
  - Identifies
    - which variables might be modified by each procedure call
    - which ones might be referencing the same memory locations
    - which parameters are known to be constants



Optimization phases (cont'd)

- In order to create the object module
  - Optimize the use of the physical registers
  - Save remaining temporaries in memory
  - Schedule all the instructions
  - Write out the assembly language

What affects the optimizer's capabilities?

- In general: Too little knowledge of the programmer's intentions!
- Pointer aliasing: lack of knowledge of which locations are being referenced
- Functions called through function pointers
- Branches, switch statements, etc.
  - Lack of knowledge of what is important
- Inefficient math expressions
- …

Compilers in Practice

Linux compilers on the market (1)

- Open source:
  - GNU compiler suite. C/C++/Fortran
    - http://gcc.gnu.org/
  - LLVM (C/C++) compiler framework
    - Originated from U. of Illinois
    - Now supported by Apple
    - http://www.llvm.org/
  - Open64 compiler suite. C/C++/Fortran
    - Derived from the SGI MIPS, IA-64 compiler
    - Now also supported by AMD
    - http://www.open64.net/

Linux compilers on the market (2)

- Commercial:
  - Intel's compiler suite (C/C++/Fortran for IA-32, Intel64, and IA-64); http://www.intel.com/
  - ST Microelectronics/Portland Group (PGI) (C/C++/Fortran) compilers; http://www.pgroup.com/
  - Pathscale compilers (Now owned by NetSyncro.com) Also derived from SGI's compilers) (C/C++/Fortran); http://www.pathscale.com/
  - Microsoft C/C++ compilers; http://www.microsoft.com/
  - Lahey/Fujitsu Fortran 95/90/77 compiler; http://www.lahey.com/

Approaching a new compiler (version)

- Quality/Ease of use:
  - Does my code compile straight out of the box?
- Correctness:
  - Do I get correct results ?
    - Don’t ignore warnings;
      - Always have good tests for checking correctness, especially of floating-point calculations
- Performance:
  - Do I enjoy the same performance ?
    - Another (or the same) set of tests for performance
      - Always risks of performance regression
  - Are there new performance capabilities?
    - Can I add new flags and get even better performance
      - Without changing the code?

How to best influence the compiler

- Multiple ways:
  - Use flags
    - Problem: There are lots of them
  - Pre-processor definitions
    - allowing executable code to be generated differently depending on each case
  - Define pragmas:
    - *#pragma vector aligned*
  - Improved use of syntactical keywords
    - *const*, *inline*, etc.
    - *__declspec(align(16))*
  - Improved visibility
    - Compile bigger chunks in one go
    - Use Interprocedural Optimization

Getting lost in flags?

- gcc performance flags:

-falign-functions=n -falign-jumps=n -falign-labels=n -falign-loops=n -fbranch-probabilities -fprofile-values - fvpt -fbranch-target-load-optimize -fbranch-target-load-optimize2 -fcaller-saves -fcprop-registers -fcse-follow-jumps-fcse-skip-blocks -fdata-sections -fdelayed-branch -fdelete-null-pointer-checks -fexpensive-optimizations -ffast-math -ffloat-store -fforce-addr -fforce-mem -ffunction-sections -fgcse -fgcse-lm -fgcse-sm -fgcse-las -floop-optimize -fcrossjumping -fif-conversion -fif-conversion2 -finline-functions -finline-limit=n -fkeep-inline-functions -fkeep-static-consts -fmerge-constants -fmerge-all-constants -fmove-all-movables -fnew-ra -fno-branch-count-reg -fno-default-inline -fno-defer-pop -fno-function-cse -fno-guess-branch-probability -fno-inline -fno-math-errno -fno-peephole -fno-peephole2 -funsafe-math-optimizations -ffinite-math-only -fno-trapping-math -fno-zero-initialized-in-bss -fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls -fprefetch-loop-arrays -fprofile-generate -fprofile-use -freduce-all-givs -fregmove -frename-registers -freorder-blocks -freorder-functions -frerun-cse-after-loop -frerun-loop-opt -frounding-math -fschedule-insns -fschedule-insns2 -fno-sched-interblock -fno-sched-spec -fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns=n -sched-stalled-insns-dep=n -fsched2-use-superblocks -fsched2-use-traces -fsignaling-nans -fsingle-precision-constant -fstrength-reduce -fstrict-aliasing -ftracer -fthread-jumps -funroll-all-loops -funroll-loops -fpeel-loops -funswitch-loops -fold-unroll-loops -fold-unroll-all-loops --param name=value -O -O0 -O1 -O2 -O3 -Os

What should you expect?

- Understand how the compiler “behaves”
- Case 1:

```
#include <math.h>
double test()   { return pow(2.1,2) ;   }
```

- Case 2:

```
#include <math.h>
double test()   { return pow(2.1,2.1) ; }
```

- Case 3:

```
#include <math.h>
double test(double x)  { return x/x ; }
```

For the enthusiasts:
Reference: Table 7.1 in “Optimizing software in C++”
(Agner Fog: *www.agner.org/optimize/optimizing_cpp.pdf*)

Inspect your assembly code

- From the previous example (Case 2)
  - When compiling with “-S”:

- gcc 4.1.2

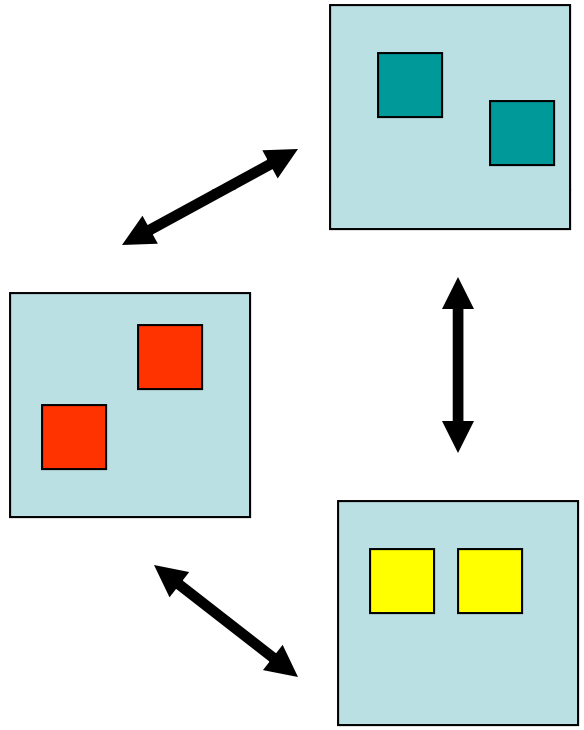
```
test:
.LFB2:
        movsd   .LC0(%rip), %xmm0
        movapd  %xmm0, %xmm1
        jmp     pow
```

- gcc 4.4.1

```
test:
.LFB0:
        .cfi_startproc
        movsd   .LC0(%rip), %xmm0
        ret
```

Interprocedural Optimization

- What is can do:
  - Function inlining
  - Optimizing calls and argument passing
  - Constant propagation
  - Alias analysis
  - Address-taken analysis
  - Unreferenced valiable removal
  - …



Profile-Guided Optimization

- Compilers have normally no clue as to what will happen during execution
  - If/else, switch statements, etc.
- With PGO, a compiler can analyze your software at run-time and choose the “best” optimization techniques dependent on the path actually taken
  - gcc (as of 4.1)
    - -fprofile-generate + test run + -fprofile-use
  - icc
    - -prof-gen + test run + -prof-use
- Main problems:
  - Time consuming
  - Must make the test run representative

Using more than one compiler?

- Multiple reasons:
  - You get more faith in your own code base
    - When it builds with different compilers
      - Some compilers give more and better warnings as well
  - You get more faith in your calculations
    - Again, this is especially true for FLP calculations
  - You could see different performance results
  - You could get acquainted with new and revealing performance flags
    - Autovectorization; Profile-Guided Optimization
    - Etc.

Floating-Point Representation

(See Peter Elmer's talk)

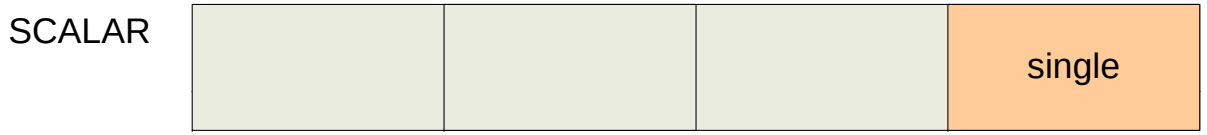
"What Every Computer Scientist Should Know About Floating-Point Arithmetic"
David Goldberg, 1991, 48 pages (http://portal.acm.org/citation.cfm?id=103163)

Vectorization

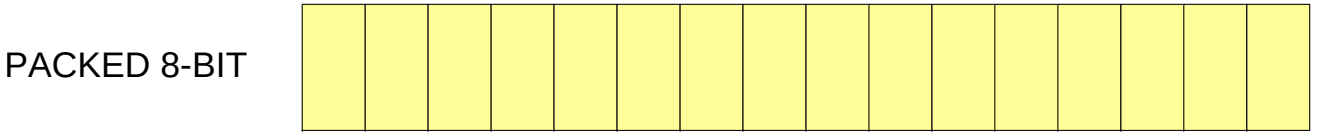
Back to SSE data types

In 128 bits:

SINGLE PRECISION



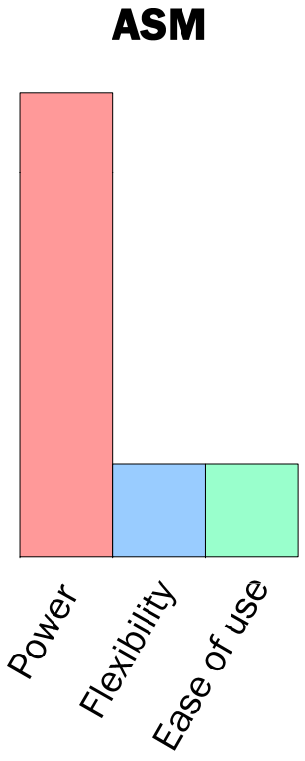
DOUBLE PRECISION



Programming levels with SSE

- Classical tradeoff: code manageability vs. speed
- Available levels
  - Assembly
  - Intrinsics (C/C++)
  - Autovectorization (C/C++)
- Focus on HOTSPOTS!

Vectors in the x86 assembly (1)



- There are more pleasant languages... but not powerful enough
- All vector operations can be controlled directly
- Pros:
  - Full, fine-grained control
  - Useful for inner loops within higher level code
- Cons:
  - Large development overhead
  - Poor code manageability
  - Low flexibility
  - Tied directly to a particular architecture

Vectors in the x86 assembly (2)

- High level code (straight block)

```
a[0] = b[0] * c[0];
a[3] = b[3] * c[3];
a[2] = b[2] * c[2];
a[1] = b[1] * c[1];
```

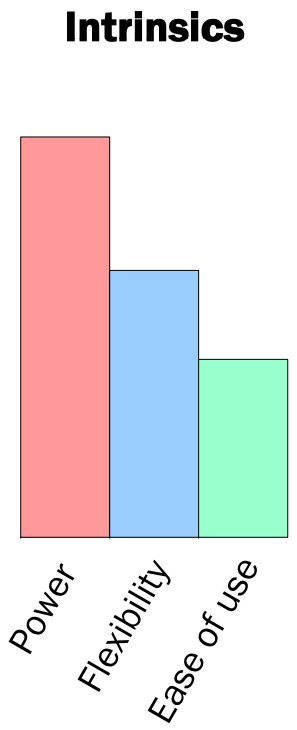
- Assembly

```
movaps %xmm0, b          ; load 4 elem
mulps  %xmm0, c          ; multiply
movaps a, %xmm0          ; store
```

Example from “The Software Vectorization Handbook”, A. J. C. Bik, Intel Press

Vector intrinsics (C/C++)

- Most SSE related operations in assembly can be invoked using intrinsics
- Pros:
  - Much easier to write than inline assembly
    - Access to instructions without the need to manage registers or code scheduling
  - Good performance and fine-grained control still possible
  - Best for inner loops within higher level code
    - (possibility to combine effectively with C/C++)
- Cons:
  - Some additional development overhead
  - Medium flexibility
  - Typically tied to a particular architecture

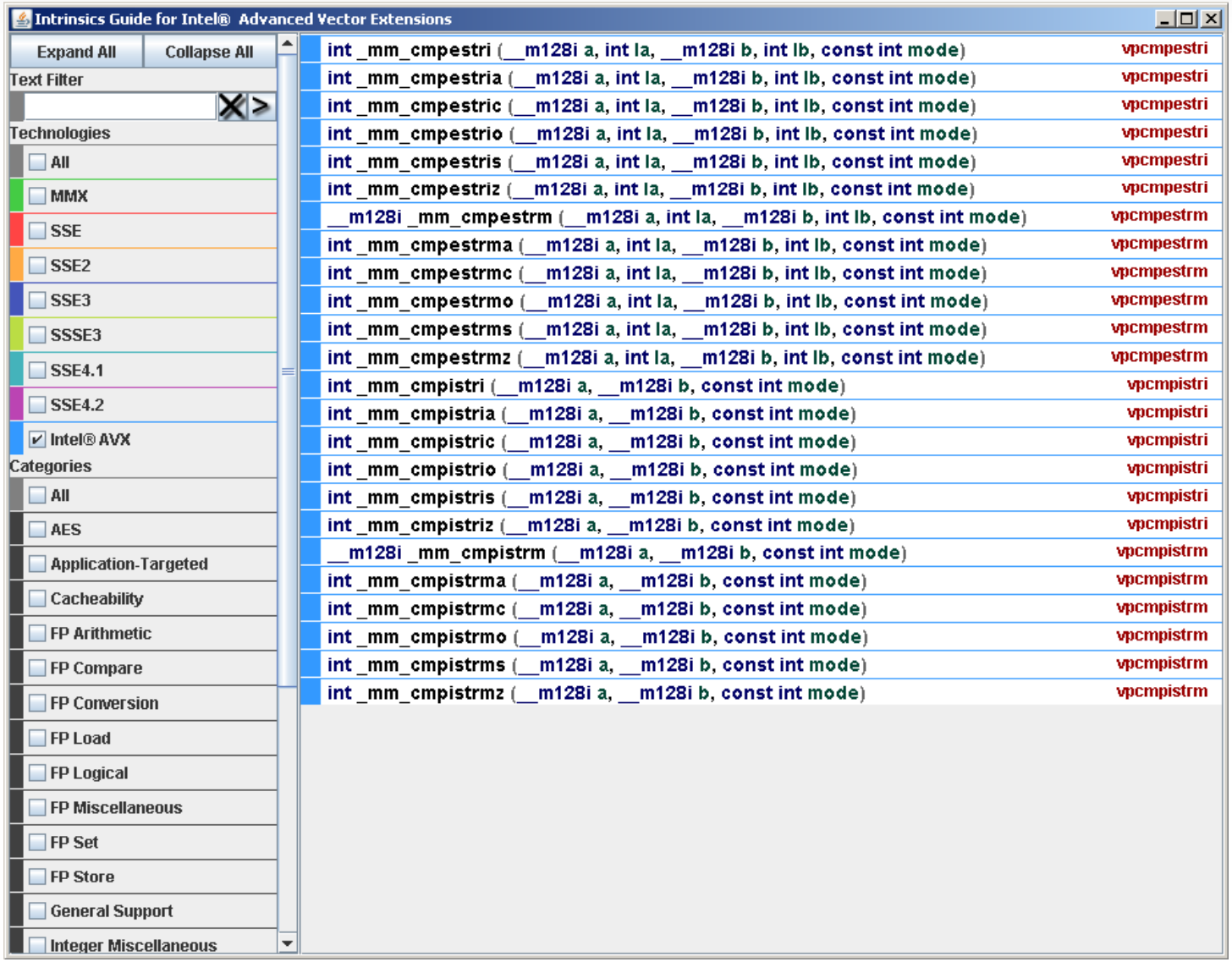


SSE* Intrinsics with C++
(HLT example)

```
F64vec2 operator +(const F64vec2 &a, const F64vec2 &b)
                          { return _mm_add_pd(a,b); }

F64vec2 min( const F64vec2 &a, const F64vec2 &b )
                          { return _mm_min_pd(a, b); }

F64vec2 sqrt ( const F64vec2 &a )
                          { return _mm_sqrt_pd (a); }

F64vec2 operator<( const F64vec2 &a, const F64vec2 &b )
                          { return _mm_cmplt_pd(a, b); }
```

Source: HLT demo
(CERN openlab / Intel / Univ. Heidelberg)

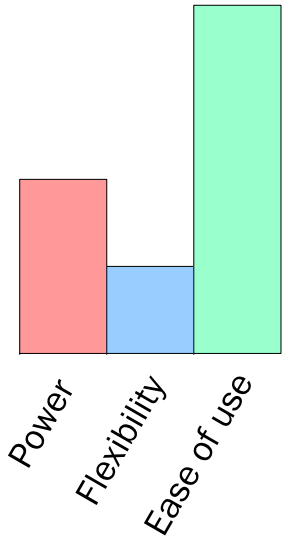
Intrinsics browser



Autovectorization (C/C++)

- Heavily compiler and code dependent
  - Although the principle is the same, GCC and ICC differ
- Numerous benefits, numerous pitfalls
  - Speedups of 2x are not uncommon
  - Delicate: for example, one data type change in your loop variable can derail all compiler efforts to vectorize the loop
- Pros:
  - Speedups can often be achieved with virtually no effort on the programmer's part
  - Compiler reports make it easier
  - Architecture independent on the source level
- Cons:
  - Difficult to control, many pitfalls
  - Heavy dependencies
  - Gains not as significant as with direct techniques – only as good as the compiler

Autovectorization



Autovectorization techniques

- Basic premise: simple loops are automatically transformed into vectors by the compiler
  - Only as smart as the compiler
- Conventional autovectorization
  - Vectorizing inner loops
  - Data dependencies break this scenario
- Loop unrolling to match cache line size
- Loop peeling to align data
- Basic block autovectorization
  - A bigger block of code is autovectorized
  - Also applicable to smaller loops/vectors
- Branch statements are sometimes handled well using predication
- Numerous caveats
  - Example: ICC autovectorizes only the first inner loop in a block of code
  - Data alignment issues
  - Changing just one line might have huge consequences

Autovectorization example

```
/* 01 */ #define N 32
/* 02 */ float a[N], b[N], c[N], d[N];
/* 03 */
/* 04 */ doit() {
/* 05 */   int i;
/* 06 */   for (i = 0; i < N/2; i++) a[i] = i;
/* 07 */   for (i = 0; i < N; i++) b[i] = i;
/* 08 */   for (i = 0; i < N; i++) c[i] = a[i] + b[i];
/* 09 */   for (i = 0; i < N; i++) a[i] = b[i] + c[i];
/* 10 */   }
/* 11 */   d[0] = 10;
/* 12 */   d[1] = 10;
/* 13 */   d[2] = 10;
/* 14 */   d[3] = 10;
/* 15 */ }
```

```
main.c(6)  : (col. 11) remark: LOOP WAS VECTORIZED.
main.c(7)  : (col. 11) remark: PARTIAL LOOP WAS VECTORIZED.
main.c(11) : (col. 11) remark: BLOCK WAS VECTORIZED.
```

Example from “The Software Vectorization Handbook”, A. J. C. Bik, Intel Press

You versus the Compiler

- You expect the compiler always to do the right things for you
  - This is obviously the best, but does not always happen
    - May even be a regression issue: “It used to work !”
- The compilers expect you to do the right thing
  - Good programmers may do it right; Others may “forget”
    - Give strong hints as to the intentions
    - Give maximum visibility

Conclusions

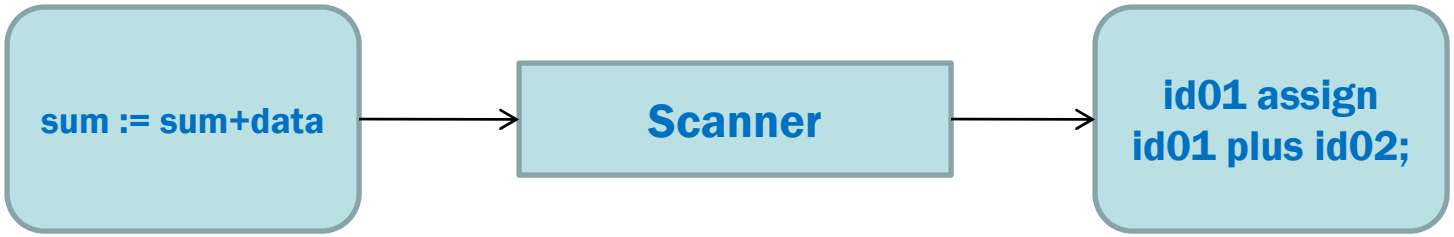
- For everybody with millions of source lines:
  - The compiler must be considered a “close ally”
    - Let it know your intentions; Find out how it reacts
  - Obtain trust through stringent testing
    - Correctness; Speed
  - Consider using more than one compiler
- Floating-point (See Peter’s talk)
  - Understand how data is:
    - Represented
    - Manipulated during calculations
- Vectorization
  - Understand the potential
    - Today and tomorrow

Q & A

CERN openlab

BACKUP

Scanning or Lexical Analysis (1)

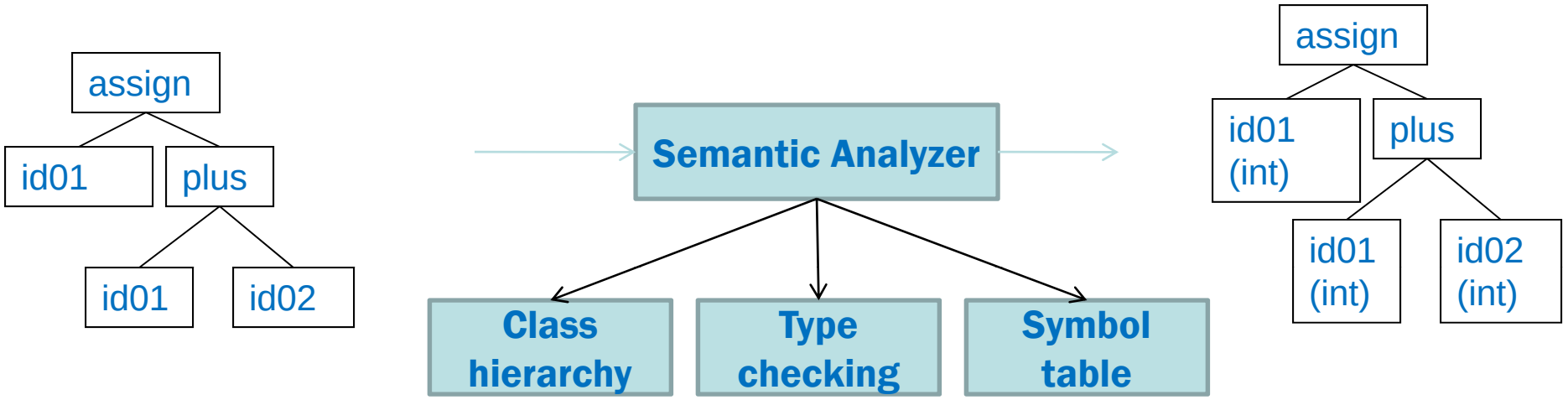


- Partition of the text into tokens (smallest meaningful unit)
- Remove comments, white spaces, etc.
- Track line numbers
- The scanner is basically a recognizer of a regular language

Parsing or Syntactic Analysis (2)

- Build Abstract Syntax Tree (AST)
- The parser is a recognizer of a context-free language

Semantic Analysis (3)



- Symbol Table creation (debugging)
- Class inheritance hierarchy
- Type checking
- Static semantic checking (def before use)

Floating-Point Representation

“What Every Computer Scientist Should Know About Floating-Point Arithmetic”
David Goldberg, 1991, 48 pages (http://portal.acm.org/citation.cfm?id=103163)

A few words on floating point

- IEEE754 as a standard
- Numbers are represented in a binary notation:
  - $S * 2^e * M$
  - For instance, in double precision (64-bit)
    - Sign: 1 bit (0 – positive, 1 – negative)
    - Exponent: 11 bits
    - Mantissa: 52 bits (for the fraction): 1.fffffffff
- Decimal number are often not 100% accurate:
  - 1.0 is OK, 0.1 is not
- Accuracy can be destroyed in one line:
    - d = a + b – c;
    - a = 2.0; b = 3.333333e-17; c = 2.0;
    - What happens to d ?

IEEE 754

- Quickly summarized:

	Sign	Exponent	Mantissa	Max exponent	Precision (10^{xxx})
Single	**1**	**8**	**23**	**+127**	**7.2**
Double	**1**	**11**	**52**	**+1023**	**16.0**
Extended	**1**	**15**	**64**	**+16383**	**19.3**

Note that x87 uses 80-bit registers, whereas SSE uses 64 bits only (in a DP calculation).

Some FP rules

- Understand the IEEE standard
- Understand if the compiler follows the standard strictly or not
  - Rules often change with O3
  - For instance,
    - Use reciprocal rather than division
    - Math libraries with less accuracy
- Understand the ranges of your numbers
  - What precision do you really need?
- Sum up from smallest to largest
- Do not mix single and double precision
  - Especially bad for SSE