MAE 5032 High Performance Computing: Methods and Practices

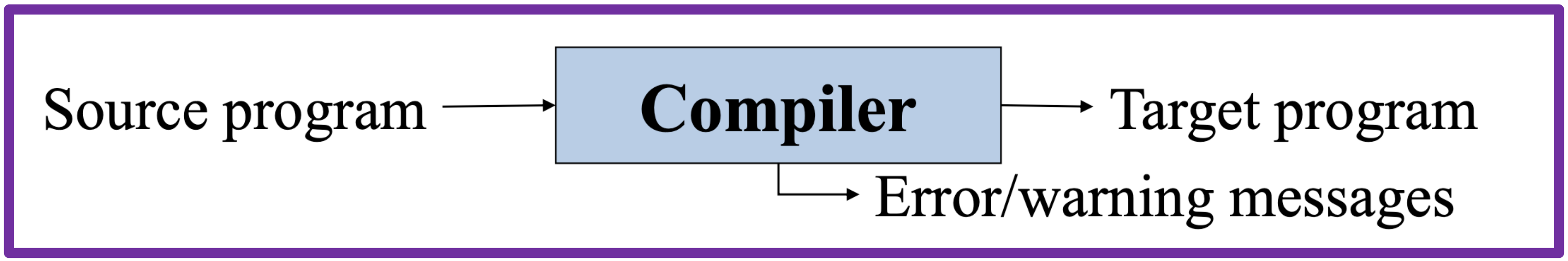
Lecture 5: Compilers

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn

Introduction

What is a compiler

- A program, which can be executed by a processor, is written by machine languages.
- A programer writes in some human-readable higher-level languages (C, C++, Fortran).
- A Compiler translates programs written in one language into "equivalent" programs in another language.



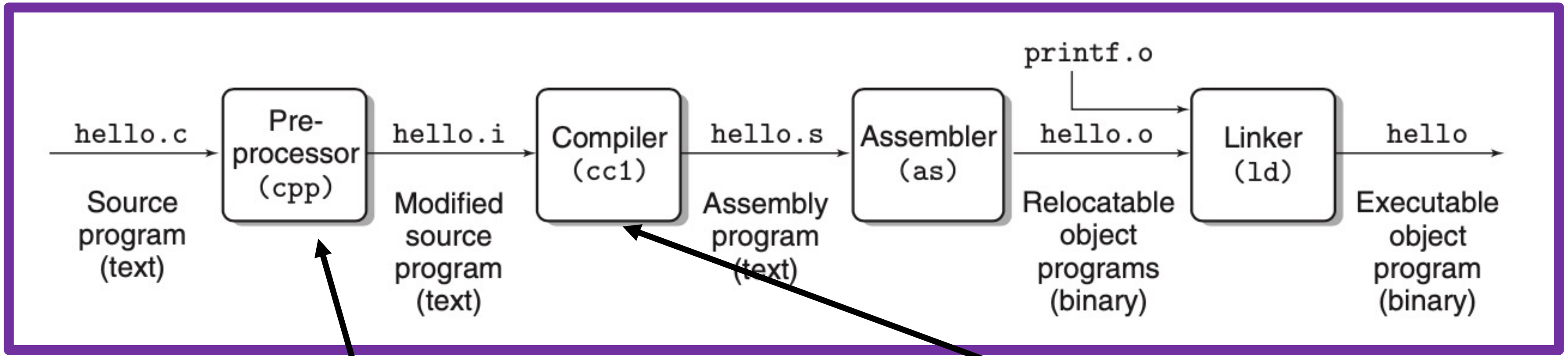
UNIX and C

- UNIX originally written in B (C’s predecessor)
- Computer architecture changes necessitated language changes
  - ➢ developer’s machine went from word-oriented memory to byte oriented memory
  - ➢ B -> C
- UNIX quickly rewritten in C as it developed
- Simplicity of C enabled easier portability
- Easier portability let UNIX jump architectures
- C Standard Library became an integral part of UNIX

UNIX and C

- Compilers (with assemblers and linkers) turn human-readable programming languages into machine-executable programs
- HPC tends towards
  - C (and its dialects)
    - UNIX is written in C
    - supercomputer runs UNIX
  - Fortran (and its dialects)
    - predates UNIX
    - designed so that compiler can know easily what optimization it can do

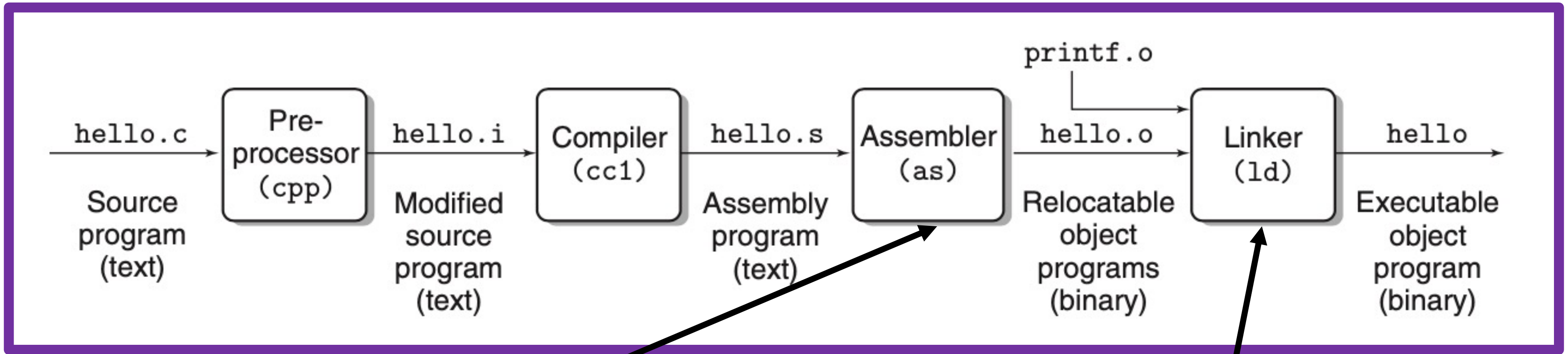
Invocation



Preprocessor phase: preprocessor modifies the C program according to directives that begin with '#' character. For example, $include<stdio.h> tells the preprocessor to read the contents in the system header file stdio.h and insert it directly into the program text.

Compilation phase: The compiler translates the text file hello.i into the text file hello.s, which contains an assembly language program (low-level machine language instruction in a textual form). It is a common language for different high-level languanges. For example, C and Fortran compilers both generate output files in the same assembly language.

Invocation



Assembly phase: The assembler translages hello.s into machine instructions, package them into relocatable object program hello.o. This is a binary file.

Linking phase: The linker handles the merging of printf.o and hello.o, and generate the executalbe hello binary file, which is ready to be loaded into memory for executation.

Compilers and Liners

- Traditionally, *cc* is the compiler and *ld* is the linker on UNIX systems.

```c
#include "hello.h"

int main()
{
  hello("world");
  return 0;
}
```

- Every object file contains
  - executable machine code
  - symbol table: functions, variables, types
- The linker coordinates symbols amongst object files and system libraries to create the executable.
  - usually no symbol table in the result
  - debuggers need the symbols
  - compiler/linker flags to add them to the executable

Compilers and Liners

- GCC: GNU Compiler Collection
  - support C, C++, Objective C, Fortran 77/95, Java, Ada
  - Free software
  - Highly portable.

GNU	gcc, g++, g77, gfortran/g95
Intel	icc, icpc, ifort
IBM XL	xlc, xlC, xlf, xlf95
Sun Studio	cc, CC, f77, f95
Portland Group	pgcc, pgCC, pgf77, pgf90, pgf95
Pathscale	pathcc, pathCC, pathf77, pathf95

Compilers and Liners

- Intel
  - support C, C++, Fortran 77/95
  - Available for Linux and Windows for x86 architecture, optimized for intel chips
  - Free for non-commercial use on Linux

GNU	gcc, g++, g77, gfortran/g95
Intel	icc, icpc, ifort
IBM XL	xlc, xlC, xlf, xlf95
Sun Studio	cc, CC, f77, f95
Portland Group	pgcc, pgCC, pgf77, pgf90, pgf95
Pathscale	pathcc, pathCC, pathf77, pathf95

Compiling and Linking Codes

Invocation

- Most basic

```
gcc -Wall hello.c
```

  - Creates an executable a.out
  - Links in default libraries only (libc for certain, others vary by compiler/architecture/OS)

```c
#include <stdio.h>

int main()
{
  printf("Hello world!\n");
  return 0;
}
```

- Name the executable

```
gcc -Wall -o hello hello.c
gcc -Wall hello.c -o hello
```

  - Creates executalbe hello

Invocation

- option **–o** : specifies the output file name.
- option **–Wall** : turns on all the most commonly-used compiler warnings.
  - Code that does not produce any warning messages is said to be **compiled cleanly**.
  - -w disable warning messages
  - -W<foo> turns on warnings about foo

```
#include <stdio.h>

int main()
{
  printf("2 + 2 = %f\n", 4);
  return 0;
}
```

Invocation

- option **–o** : specifies the output file name.
- option **–Wall** : turns on all the most commonly-used compiler warnings.
  - Code that does not produce any warning messages is said to be

```c
#include <stdio.h>

int main()
{
  printf("2 + 2 = %f\n", 4);
  return 0;
}
```

```
-> gcc warning.c
mae-liuj::login01
le }
-> ./a.out
2 + 2 = 0.000000
```

```
-> gcc -Wall warning.c
warning.c: In function ‘main’:
warning.c:5:3: warning: format ‘%f’ expects argument of type ‘double’, but argument 2 has type ‘int’ [-Wformat=]
   printf("2 + 2 = %f\n", 4);
   ^
```

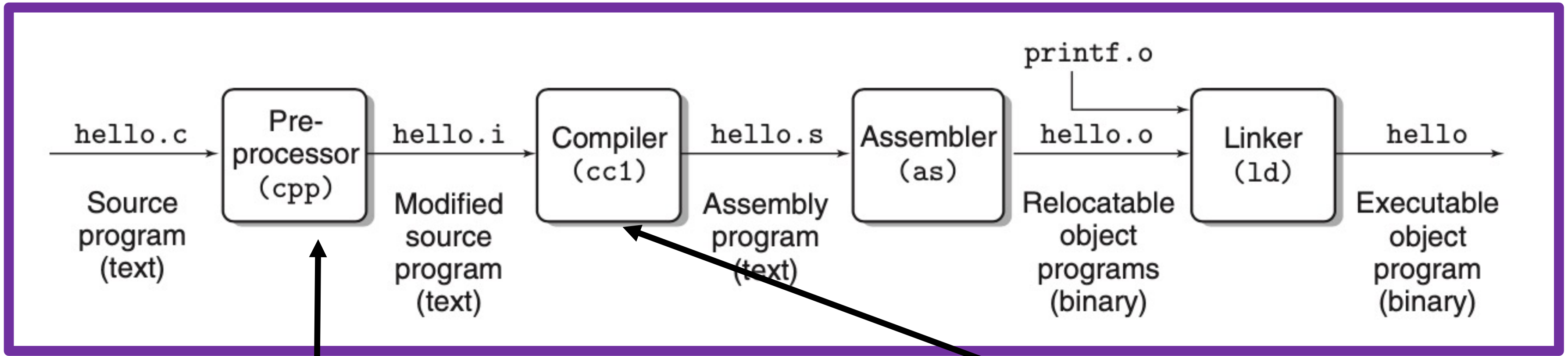
Invocation

- Compile without linking `gcc –Wall –c hello.c`
  - ➢ Creates object file hello.o

- Compile only with name `gcc –o hello.o -c hello.c`
  - ➢ Create object file with name hello.o

- Creating executables from object files `gcc hello.o –o hello`
  - ➢ Here you do not need –Wall (why?)

Invocation



Preprocessor phase:
gcc –E
cpp hello.c > hello.i

- gets rid of comments
- includes the code in the header files
- replace all macros

Compilation phase:
gcc –S

produce assembly code at this step.

Invocation

- Compile multiple source files `gcc main.c hello.c -o newhello`

```c
#include "hello.h"

int main()
{
  hello("world");
  return 0;
}
```

main.c

```c
void hello( const char * name );
```

hello.h

searches the system header file directory

```c
#include <stdio.h>
#include "hello.h"

void hello (const char * name)
{
  printf("Hello %s!\n", name);
}
```

hello.c

search the current directory before looking in the system header file directories.

Invocation

- Compile without linking

```
gcc -c hello.c
gcc -c main.c
```

- Creating executables from object files

```
gcc hello.o main.o -o hello
```

  - link order may be important. Traditionally, object file which contains the definition of a function should appear after any files which call that function. Otherwise you may see linking error like “undefined reference to function”.

  - most current linkers will search all object files regardless of order.

Invocation

- Compile without linking

```
gcc -c hello.c
gcc -c main.c
```

- Creating executables from object files

```
gcc hello.o main.o -o hello
```

- You may make some changes to hello.c and recompile hello.c to hello.o and relink it to hello without compiling main.c.
  - Linking is faster than compiling. This can be critical in large projects.

Invocation

- Compile multiple source files

```
gcc foo.c bar.c baz.c -o foo
```

- Linking multiple object files into one executable

```
gcc foo.o fun1.o fun2.o -o foo
```

- Linking a library by hand (static)

```
gcc foo.c -o foo /home/user/mylib/libmylib.a
```

  - Works just like object file linking

Library

Library

- Libraries contained compiled codes which can be used by other programs
  - it is easy to keep the code modular
  - it is easy to reuse others' (well-written) code
  - save your programming time and ease the process of large software development
  - the compiled library can be roughly viewed as an archived object files with a table of functions
  - the library names are typically
    - libname.a or libname.so.x.y.z
  - you may refer to the header file for the ways of calling the library functions
  - you have used C libraries already
    - GNU C library is the core libraries in GNU system, including printf, malloc, write, etc

Some useful libraries

Linear algebra: MKL, LAPACK, ScaLapack, BLAS
PETSc, Trilinos, Hypre, MUMPS SLEPc

Parititioning: METIS, ParMETIS, Zoltan

Input/Output: HDF5, NetCDF, Globus

Diagnostics: TAU, PAPI

Applications: OpenFOAM, FEniCS, … …

Static vs. Dynamic Libraries

- Static
  - puts all the external routines into the created executable
  - no dependencies at run time
  - leads to larger binaries
  - takes longer to build the executable

- Dynamic
  - contains only a samll table of functions it requires instead of complete machine code
  - loads the routines at run time
  - decreases the build time and the binary size
  - dynamic libraries can be shared by different executables in memory.

Static vs. Dynamic Libraries

- Static
  - usually called `libfoo.a’
  - created as an archive of object files
  - no special options needed when building

- Dynamic
  - usually called `libfoo.so’
  - more complicated to build than static libraries
  - compilers prefer linking shared libraries by default

Making static libraries

- Common code that is useful between programs or that changes frequently can be put in a library with `ar`
- `ar` is more than an object file archiver
  - you can use it for any kind of files
  - similar to tar, but tar maintains a directory structure while ar just archive a plain set of files

Making static libraries

- ar r libfoo.a foo.o bar.o baz.o
  - creates/adds to libfoo.a
  - inserts foo.o bar.o baz.o
  - overwrites members of the same name
- ar s libfoo.a
  - creates or updates the object-file symbol table libfoo.a
  - may be combined with `r’ to do it all at once
  - ar rs libfoo.a foo.o bar.o baz.o
  - ranlib libfoo.a is a synonym
- ar t libfoo.a
  - prints the list of files contained in libfoo.a

Making shared libraries

- Compiling with position independent code
  - `gcc -c -Wall -fpic hello.c`

- Creating a shared library from an object file
  - `gcc -shared -o libhello.so hello.o`

Invocation

- By default, gcc searches the following directories for header files:

```
/usr/local/include
/usr/include
```

  and the following directories for libraries

```
/usr/local/lib
/usr/lib
```

  E.g. C standard math library is in /usr/lib/libm.a and its header file is in /usr/include/math.h

- If the they are not found, error messages will be given

```
FILE.h: No such file or directory
/usr/bin/ld: cannot find library
```

Invocation

- By default, gcc searches the following directories for header files:

```
/usr/local/include
/usr/include
```

  and the following directories for libraries

```
/usr/local/lib
/usr/lib
```

  E.g. C standard math library is in /usr/lib/libm.a and its header file is in /usr/include/math.h

- On systems supporting 64-bit executables, the 64-bit versions will often be stored in /usr/lib64 and /lib64.

Invocation

- Link a library

```
gcc calc.c -o calc -lm
```

  - looks for libm.a or libm.so in the compiler/linker search path
  - most compilers choose *.so over *.a

- Link a library in a non-standard path

```
gcc calc.c -o calc -L/home/user/lib/ -lmylib
```

  - adds /home/user/lib to the library search path
  - looks for libmylib.a or libmylib.so in the search path.

Invocation

- A typical error message at the linking stage

```
ccbR60jm.o: In function `main'
ccbR60jm.o: undefined reference to `sqrt'
```

The above message means that the function sqrt is not defined in the program or in the default library. One need to specify explicitly the link path for the sqrt function.

```
gcc -Wall calc.c /home/user/lib/libmylib.a -o calc
```

The linker searches through libmylib.a, which contains object files for all mathematical functions (sin, cos, exp, log, sqrt, etc) and locate the object file of the sqrt function.

Invocation

- Link order may be critical in certain linkers: they search from left to right. A library containing the definition of a function should appear after any source files or object files which use it.
  - `gcc -Wall calc.c -lm -o calc` (correct)
  - `gcc -Wall -lm calc.c -o calc` (wrong) -> undefined reference error

- A library calling an external function defined in another library should appear before the library containing the function.
  - `gcc -Wall calc.c -lglpk -lm`

  `glpk uses a math function in libm.a.`

Setting search paths

- By default, gcc searches the following directories for header files:

  `/usr/local/include` and `/usr/include`

  and the following directories for libraries

  `/usr/local/lib` and `/usr/lib`

  which are often called as the include path and library search path/link path.

- When additional libraries in other directories are needed, it is necessary to extend the search paths with the –I and -L options.

Setting search paths

- A typical error message

```
gcc -Wall calc.c -lgdbm
main.c:1: gdbm.h: No such file or directory
```

suggests that the gdbm library is installed under a non-standard directory, which is not in the default gcc include path.

```
gcc -Wall -I/home/user/lib/gdbm-1.8.3/include calc.c -lgdbm
/usr/bin/ld: cannot find -lgdbm
collect2: ld returned 1 exit status
```

Now the header file is found, but the library is still missing from the link path.

```
gcc -Wall -I/home/user/lib/gdbm-1.8.3/include -
L/home/user/lib/gdbm-1.8.3/lib calcc.c -lgdbm
```

Setting search paths

- The additional search path for header files and libraries can also be controlled by environment variables in the shell.

  C_INCLUDE_PATH

  LIBRARY_PATH

```
export C_INCLUDE_PATH=/home/user/lib/gdbm-1.8.3/include:$C_INCLUDE_PATH
export LIBRARY_PATH=/home/user/lib/gdbm-1.8.3/lib:$LIBRARY_PATH
gcc -Wall main.c -lgdbm
```

Setting search paths

- The additional search path for header files and libraries can also be controlled by environment variables in the shell.

  C_INCLUDE_PATH

  LIBRARY_PATH

- Compilers search the directoires in the following order
  1. command-line options –I and –L from left to right
  2. directories specified by environment variables such as C_INCLUDE_PATH and LIBRARY_PATH
  3. default system directories

Setting search paths

```
gcc -Wall -I/home/user/lib/gdbm-1.8.3/include -L/home/user/lib/gdbm-
1.8.3/lib main.c -lgdbm
./a.out
error while loading shared libraries: libgdbm.so cannot open shared object
file: No such file or directory
```

This is because the gdbm package provides a shared library as the compiler searches for the shared library first. When the executable a.out gets loaded to run, the loader searches for shared libraries only in a predefined set of system directory, such as /usr/local/lib and /usr/lib.

One may add the search directory to LD_LIBRARY_PATH.

Setting search paths

```
gcc -Wall -I/home/user/lib/gdbm-1.8.3/include -L/home/user/lib/gdbm-
1.8.3/lib main.c -lgdbm
./a.out
error while loading shared libraries: libgdbm.so cannot open shared object
file: No such file or directory

export LD_LIBRARY_PATH=/home/user/lib/gdbm-1.8.3/lib
./a.out
```

You may want to set LD_LIBRARY_PATH in .bashrc or .bash_profile file.

Paths are separated by :

Forcing static linking

- Dynamic linking preferred on most systems when both `libfoo.a' and `libfoo.so' are available
  - the in memory sharing a be a big win for certain libraries that everyone uses
- Most compilers can be forced to link statically
  - GNU and intel: -static
- Sometimes a static version of the library is not available and using –static will cause error
  - use the by hand linking method in these cases

```
gcc -Wall -I/home/user/lib/gdbm-1.8.3/include main.c
/home/user/lib/gdbm-1.8.3/lib/libgdbm.a
```

Adding to the executable's search path

- You can add to the search path embedded in the executable

```
gcc calc.c -o calc -L/home/user/lib/ -lmylib
      -Wl,-rpath=/home/user/lib/mylib
```

-Wl used to pass command line arguments directly to the linker

-rpath linker option to add to the executable's search path

Locate dynamic libraries

- At run time, the Linux loader tries to resolve the shared library dependencies of an executable before it runs it.

- It looks in
  - paths listed in its configuration file: /etc/ld/so/conf
  - LD_LIBRARY_PATH in your environment, which separate a list of paths to look by colons just like PATH
  - the search path built-in to the executable
  - Note that MacOS uses a different variable DYLD_LIBRARY_PATH

Adding to the executable's search path

- You can add to the search path embedded in the executable

```
gcc calc.c -o calc -L/home/user/lib/ -lmylib
          -Wl,-rpath=/home/user/lib/mylib
```

-Wl used to pass command line arguments directly to the linker

-rpath linker option to add to the executable's search path

- The *ldd* command can be used to investiage the shared library dependencies of an executable.

ldd example

- The *ldd* command can be used to investiage the shared library dependencies of an executable.

```
lslogin1$ ldd foo
        libm.so.6 => /lib64/tls/libm.so.6 (0x0000003ee3d00000)
        libc.so.6 => /lib64/tls/libc.so.6 (0x0000003ee3a00000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003eeb400000)
        libdl.so.2 => /lib64/libdl.so.2 (0x0000003ee3f00000)
        /lib64/ld-linux-x86-64.so.2 (0x0000003ee3800000)
```

nm example

- The nm command can be used to list symbols in object files, libraries, and executables.

foo.c:

```
#include "bar.h"
int c=3;
int d=4;
int main()
{
  int a=2;
  return(bar(a*c*d));
}
```

bar.c:

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return(b*a);
}
```

bar.h:

```
int bar(int);
```

```
gcc -g -c -o foo.o foo.c

gcc -g -c -o bar.o bar.c

gcc foo.o bar.o -o foo
```

nm example

```
lslogin1$ nm foo.o bar.o
foo.o:
         U bar
00000000 D c
00000004 D d
00000000 T main
bar.o:
00000000 T bar
```

- U means the symbol “bar” is unknown in foo.o
- T means the symbol is listed in the text section of the object file. (useful for checking if a function is defined in a library)
- D means the symbol defines the location of global, initialized data

nm example

```
lslogin1$ nm foo.o bar.o
foo.o:
         U bar
00000000 D c
00000004 D d
00000000 T main
bar.o:
00000000 T bar
```

- Useful options
  - -a show all symbols
  - -u show only undefined symbols
- Uppercase letter for global symbols, lowercase for local symbols
- Other codes
  - C uninitialized data
  - N debugging symbol
  - R read-only data

Compile for debugging

- You can add –g option to store additional information in object files and executbles, which allows tracking errors.
- You may use gdb to get the diagnostic information.
- The compiled executable also can be traced in a debugger GDB.

```
gcc -Wall -g calc.c -o calc -L/home/user/lib/ -lmylib
    -WI, -rpath,/home/user/lib/mylib
```

Preprocessor

- Consider the following code

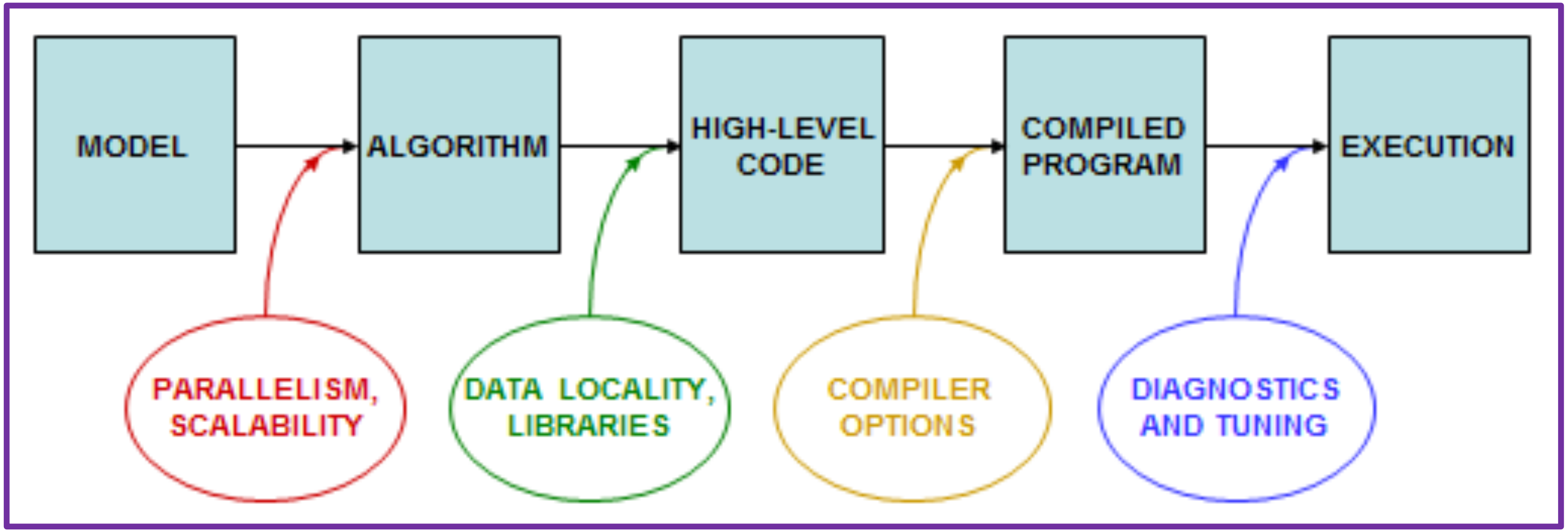
```
#include <stdio.h>
int main()
{
    #ifdef TEST
    printf("Test mode.\n");
    #endif
    printf("Running.\n");
    retrun 0;
}
```

gcc –Wall –DTEST main.c will define a preprocessor macro TEST from the command line.

If the same code is compiled without the –D option, TEST will be undefined.

Compile optimization

A big picture



Improve single-core performance

1. write a code with good locality
2. Employ optimized HPC libraries wherever possible
3. Using appropriate compiler flags when building your code

Compile optimization

- By default, compilers try to
    - reduced compilation time
    - execute code faithfully
    - make debugging make sense

- Compiler optimization can
    - increase compilation time (dramatically)
    - reduce run time
    - increase or decrease executable size
    - change the order of operations
    - eliminate some code completely
    - introduce new code

Source level optimization

- Common subexpression elimination
  - Computing an expression in the source code with fewer instructions, by reusing already-computed results.

```
x = cos(v) * (1+sin(u/2)) + sin(w) * (1-sin(u/2))
```

is rewritten to

```
t = sin(u/2);
x = cos(v) * (1+t) + sin(w) * (1-t);
```

  - Compiler will do this when optimization flag is turned on.

Source level optimization

- Function inlining
  - When a function is used, CPU need to store the function arguments in the registers and memory locations, jump to the start of the function, execute the code and return to the original point of execution when the function call is completed. The above work is called function-call overhead.

```
double sq (double x)
{
   return x*x;
}

for(i=0; i<1000000; ++i)
{
   sum += sq(i+0.5);
}
```

inlining

```
for(i=0; i<1000000; ++i)
{
   sum += (i+0.5) * (i+0.5);
}
```

Source level optimization

- Function inlining
  - When a function is used, CPU need to store the function arguments in the registers and memory locations, jump to the start of the function, execute the code and return to the original point of execution when the function call is completed. The above work is called function-call overhead.

  - Compiler may optimize for function inlining using a number of heuristics.

  - Hint compiler by using the inline keyword in C.

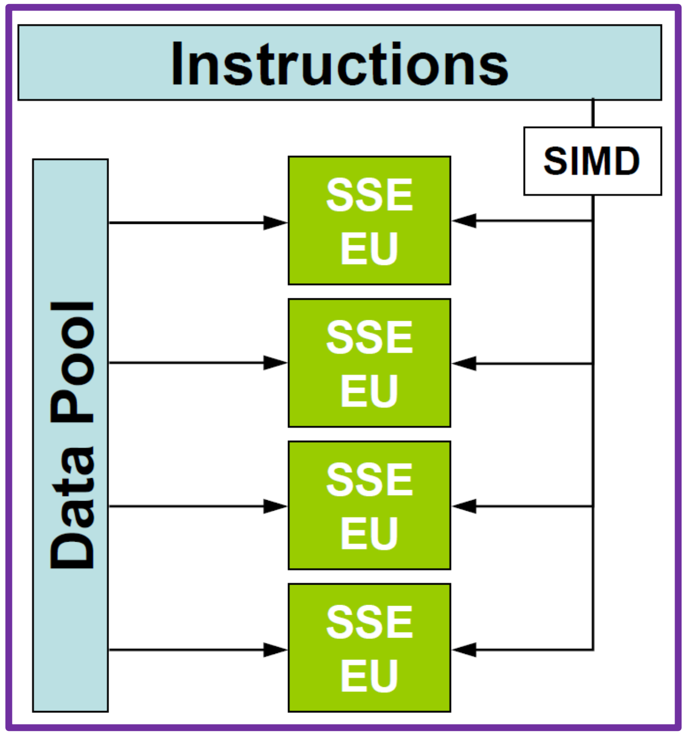
Source level optimization

- Loop unrolling allows SIMD for maximizing efficiency
    - ➢ Increase the spped and increase the executable size.
    - ➢ Streaming SIMD Extensions (SSE)
    - ➢ Advanced Vector Extensions (AVX)

```
for(i=0; i<8; ++i)
{
    y[i] = i
}
```

unrolling

```
y[0] = 0;
y[1] = 1;
y[2] = 2;
y[3] = 3;
y[4] = 4;
y[5] = 5;
y[6] = 6;
y[7] = 7;
```



Source level optimization

- Loop unrolling allows SIMD for maximizing efficiency
  - Increase the spped and increase the executable size.
  - Compilers are smart to unroll the loops, but not always
  - You may want to avoid I/O, braches, function calls insider inner loops whenever possible.

Compiler optimization

- Basic

```
$cc -O -o foo foo.c
```

- More

```
$cc -O# -o foo foo.c
```

  - # usually in the rate [1-3]
  - each level inclusive of previous levels
  - optimization levels represent a suite of options which may be enabled/disabled individually

Compiler optimization

- -O0 or no –O (default)
  - ➢ Does no optimization and compiles the code in the most straightforward way. Good for debugging.

- -O1 or –O
  - ➢ Turns on the most common forms of optimization that does not require any speed-space tradeoffs; no inlining. The code should be smaller and faster than –O0.

Compiler optimization

- -O2
  - Turns on further optimizations, in addition to those used by O1
  - Vectorization (with the Intel compilers)
  - Debugging support is retained with -g
  - Optimizations that do not require any speed-space tradeoffs
  - Generally the best choice for deployment of a program because it provides the maximum optimization without increasing the executable size.
- -O3
  - Turns on more aggresive expensive optimizations. May change code semantics and occasionaly results.
  - May increase the executable size.
  - May make the executable slower.

Compiler optimization

- -O2 default and often preferred
  - Instruction scheduling: rearranging instructions to avoid stalls due to lack of data
  - Copy propagation: replacing variables in expressions with their numerical values
  - Software pipelining: execute several stages of a loop simultaneously
  - Common subexpression elimination: finding identical expressions and calculating them only once
  - Prefetching: explicitly requesting data before it is needed
  - Loop transformations: tiling, unrolling, interchange, reversal, etc.

Compiler optimization

- For GCC, level 1 turns on
  - *-fdefer-pop*
  - *-fdelayed-branch -fguess-branch-probability*
  - *-fcprop-registers*
  - *-floop-optimize*
  - *-fif-conversion -fif-conversion2*
  - *-ftree-ccp -ftree-dce -ftree-dominator-opts -ftree-dse -ftree- ter -ftree-lrs -ftree-sra -ftree-copyrename -ftree-fre -ftree-ch*
  - *-fmerge-constants*
  - *-fomit-frame-pointer*

so O1/2/3 is really shorthand of a bunch of detailed flags that you do not need to memorize.

Compiler optimization

- `-funroll-loops`
  - turns on loop-unrolling and is independent of the other optimization flags.

- -Os
  - selects optimizations which reduce the size of an executable. Produce the smallest possible executable for systems constrained by memory or disk space.

More aggresive

- `-ffast-math`
    - `-fassociative-math`: allow reordering of instructions to something which is mathematically identical but not exactly the same in floating point operations (axb + axc => ax(b+c)).
    - `-ffinite-math-only:` assume all math are finite, which means no checking for NaN
    - `-freciprocal-math:` enables reciprocal approximations of division and reciprocal square root
    - and many others
- You may tune the behavior by calling specific flags: -fno-trapping-math, -fno-signed-zeros, -ffinite-math-only, -no-rounding-math, etc.
- https://kristerw.github.io/2021/10/19/fast-math/
- ICC has similar flags known as `-fp-model=fast`

More aggresive

x / x ⇔ 1.0	x could be 0.0, ∞, or NaN
x – y ⇔ - (y – x)	If x equals y, x – y is +0.0 while – (y – x) is -0.0
x – x ⇔ 0.0	x could be ∞ or NaN
x * 0.0 ⇔ 0.0	x could be -0.0, ∞, or NaN
x + 0.0 ⇔ x	x could be -0.0
(x + y) + z ⇔ x + (y + z)	General reassociation is not value safe
(x == x) ⇔ true	x could be NaN

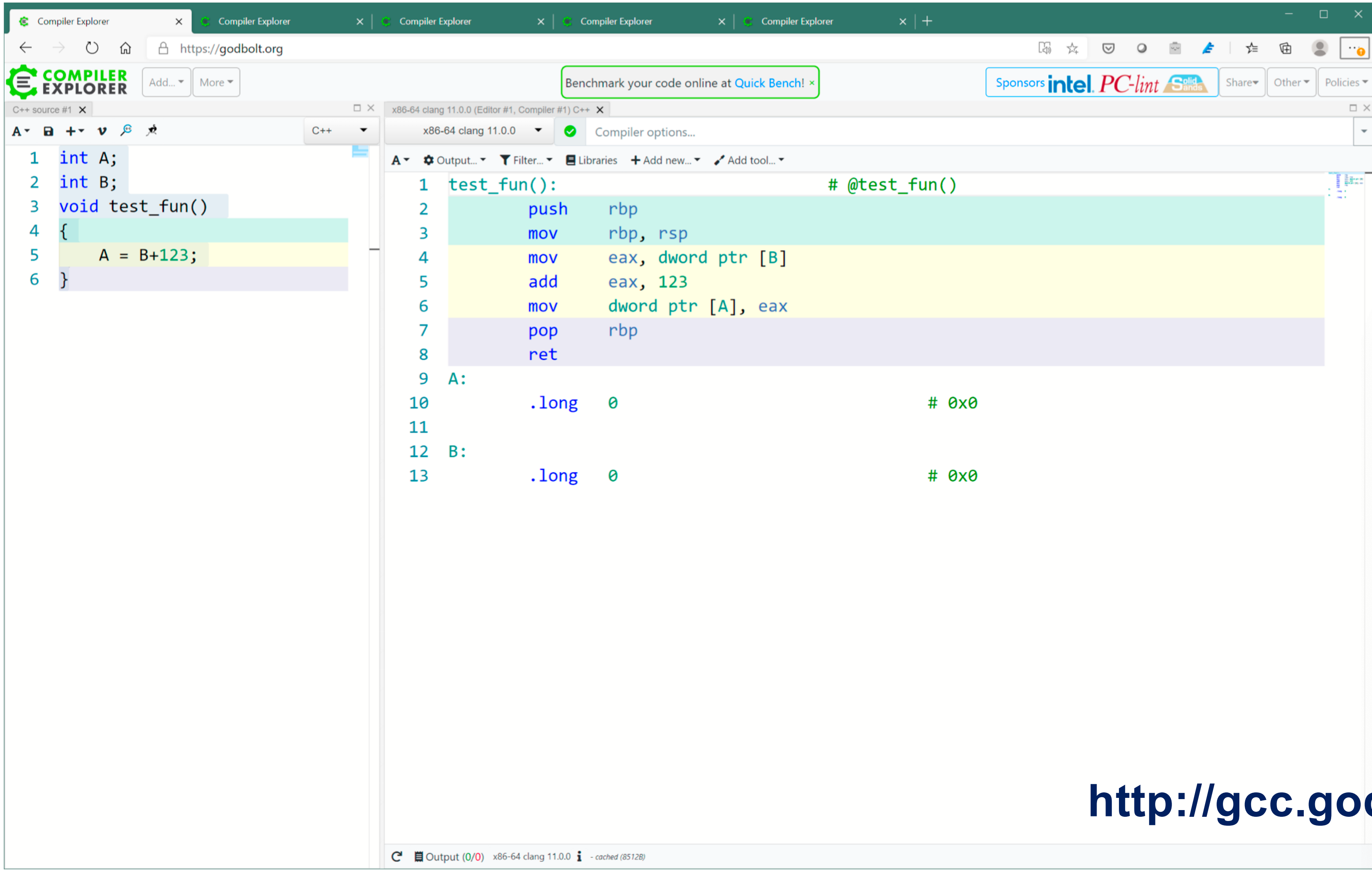
Sometimes SIMD needs fastmath

```
double s;
for(int i=0; i<256; ++i) s= s + arr[i];
```

Compiler generally does not allow this.
You need to enable –fassociative-math

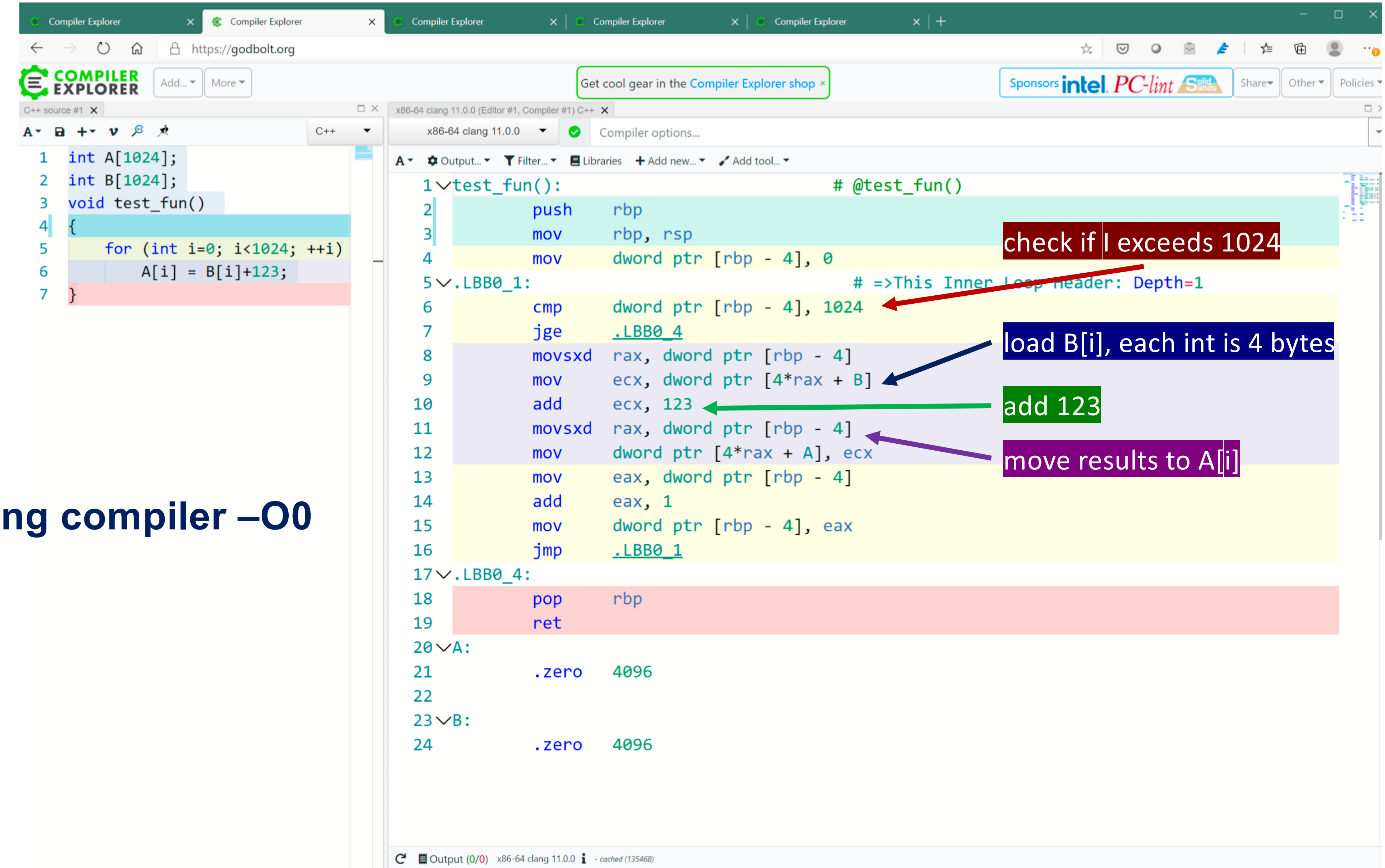
```
s0 = arr[0] + arr[4]; s1 = arr[1] + arr[5];
s2 = arr[2] + arr[6]; s3 = arr[3] + arr[7];

s0 = s0 + arr[8]; s1 = s1 + arr[9]; s2 = s2 + arr[10]; s3 = s3 + arr[11]); ...
s0 = s0 + arr[252]; s1 = s1 + arr[253]; s2 = s2 + arr[254]; s3 = s3 + arr[255]);

sa = s0 + s1;
sb = s2 + s3;

s = sa + sb;
```

Compiler optimization

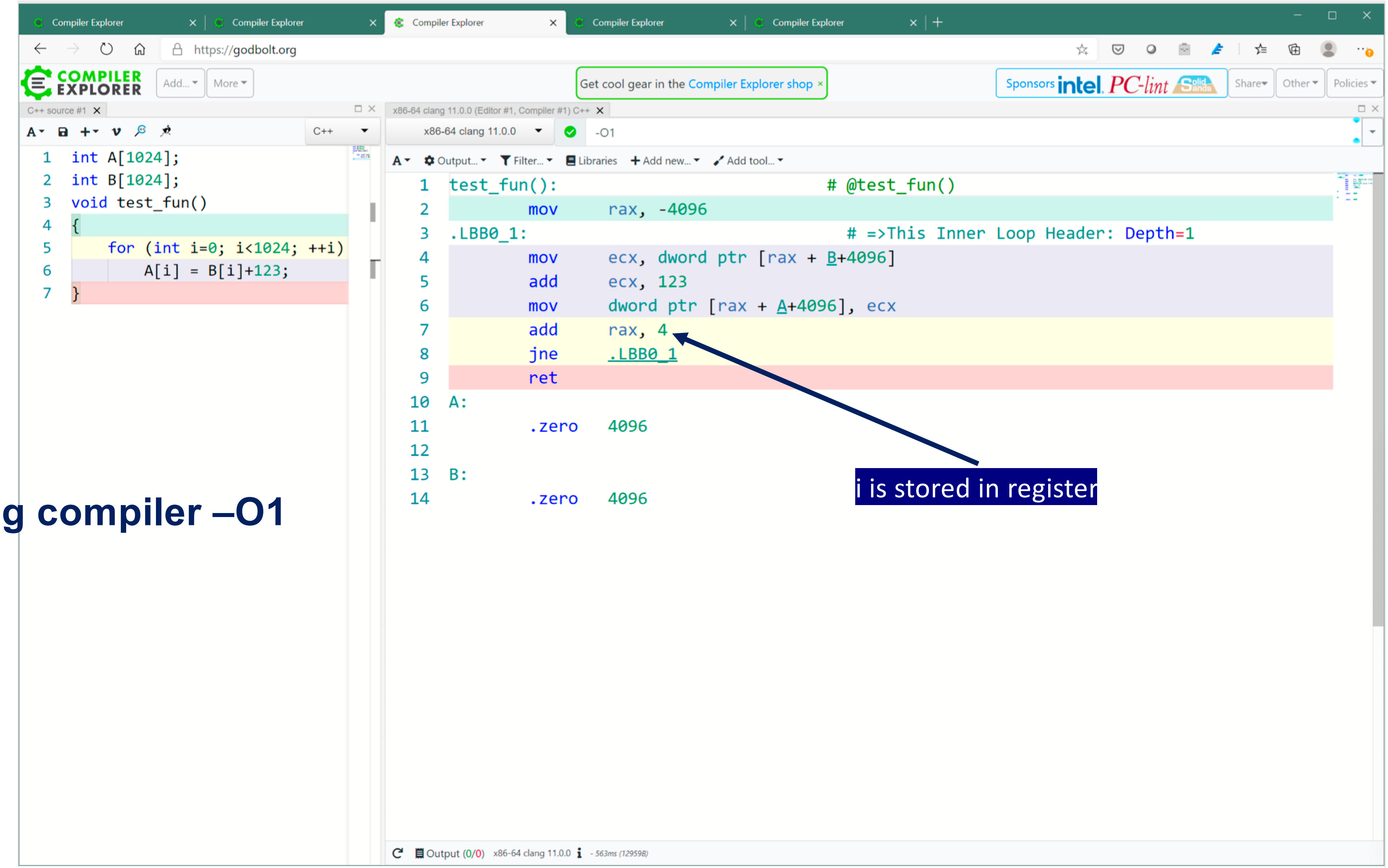


http://gcc.godbolt.org

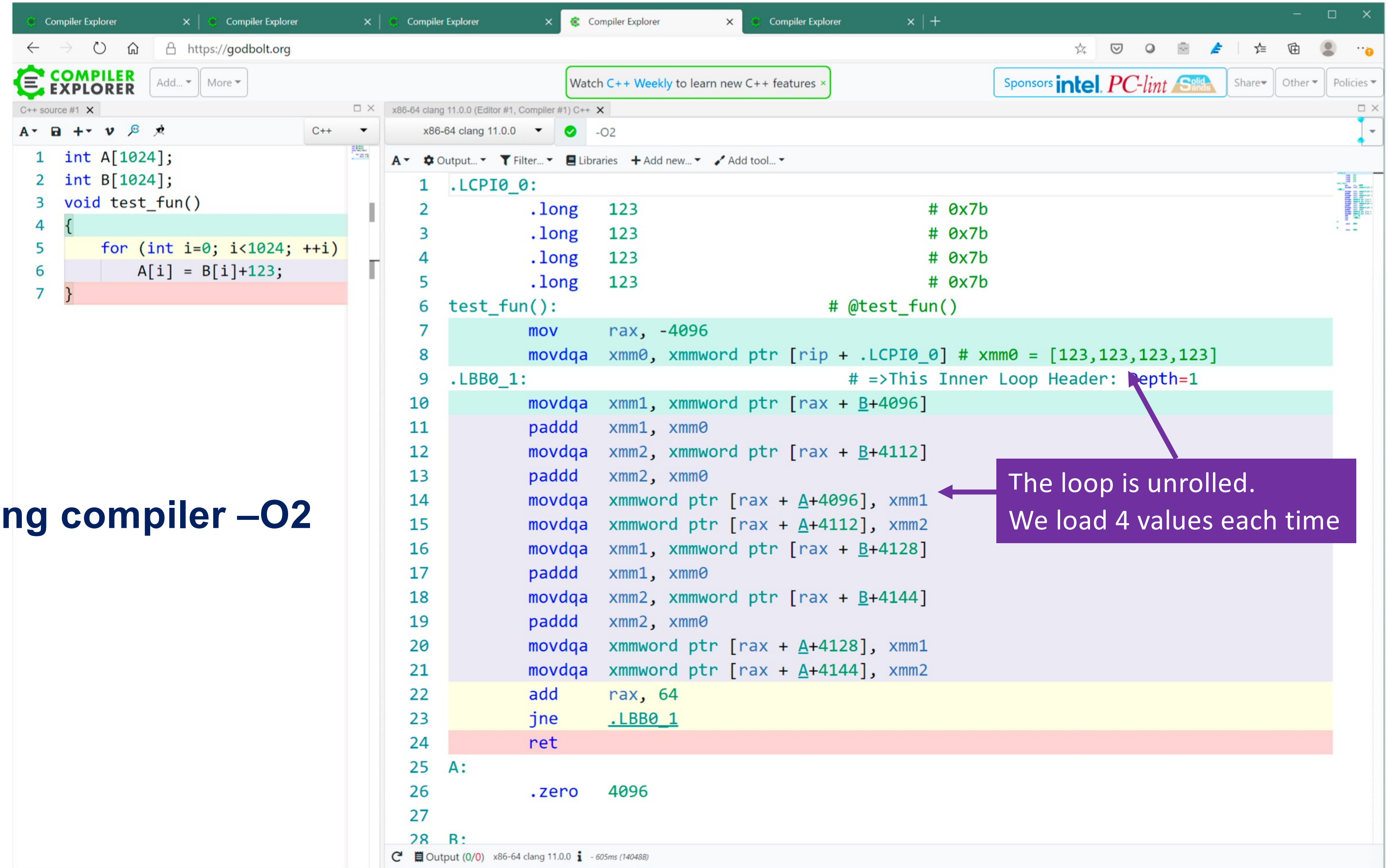
Clang compiler –O0



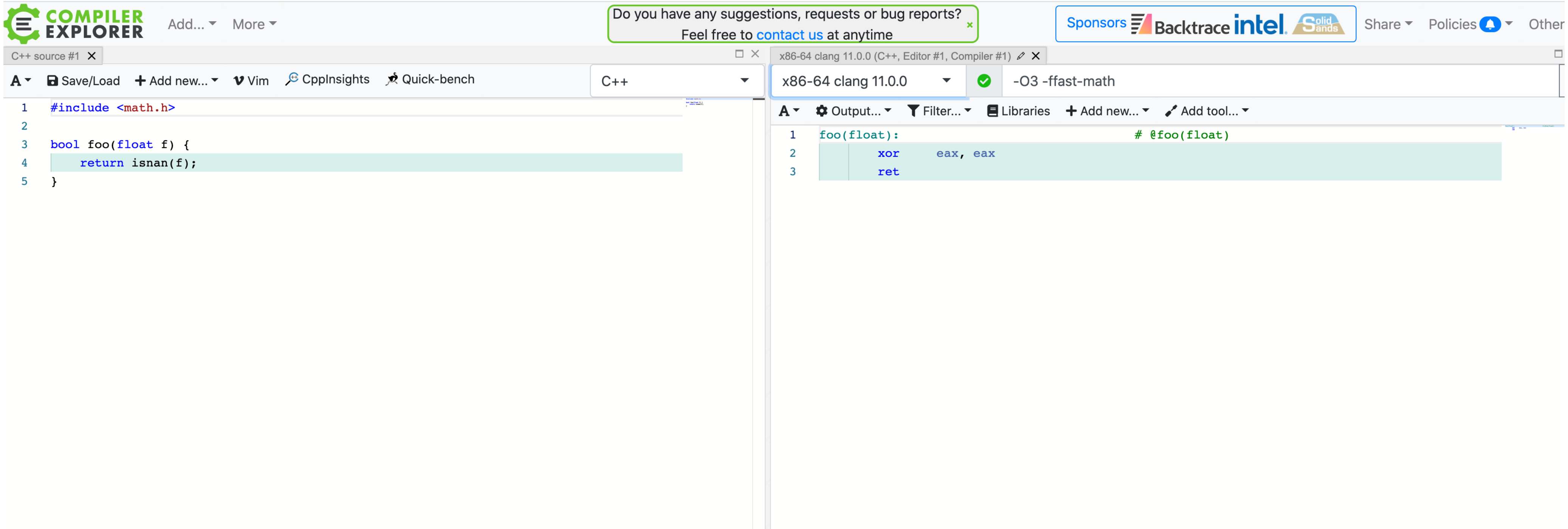
Clang compiler –O1



Clang compiler –O2



Compile optimization



Platform-specific options

- Compilers may provide platform-specific options for different types of CPUs.
    - Control features for hardware floating-point modes
    - use of special instructions for different CPUs
    - start with –m in the command line (gcc) or with –x (intel)

- The features of Intel and AMD x86 families can be controlled to produce compatible code for all x86 processors, it is possible to compile for optimizing for specific processors.

```
gcc –Wall –march=pentium4 hello.c
```

Platform-specific options

- By default, compiler assumes general architectures.
- If you know that the generated binary will run on newer machines with instruction set extensions, specify it on the command line.
- Use –march=native to auto-detect the target CPU model in gcc.
- Intel compiler
  - -x<simd_instr_set> to ensure latest vectorization hardware/instruction set is used. Default is SSE2 instruction set.
  - -xHOST optimize code based on the native node used for compiling. Make sure the login node is identical to your compute node!

Vectorization

- One critical optimization is **vectorization** based on the **register size**.
- For gcc, vectorization occurs at -03 while for intel compilers vectorization occurs at –O2.
- -O3/-O2 is often insufficient as compilers produce binaries that work for general platforms. Currently the SSE instruction is by default.
- For most of our computers, SSE is less than optimal.
- Developer must give additional options to generate binaries suitable for a more recent target architecture.

Vectorization

- On Tai-Yi, the processor supports AVX-512 instruction set, which assumes a 512-bit vector width in the register. The default SSE instructions would use only a fraction of the vector processing capability.

Vectorization with Intel Compiler Flags

- Intel compilers
  - -xCore-AVX2 to compile for AVX2, 256-bit vector width
  - -xCommon-AVX512 to compile with AVX-512
  - -xCommon-AVX512 –qopt-zmm-usage=high to be more aggresive
  - -xSKYLAKE-AVX512 on TaiYi

- GCC compilers
  - Use –mavx2 to compile for AVX2
  - GCC 5.3 + has -march=skylake-avx512
  - GCC 6.1 + has -march=knl
  - GCC 9.1 + has -march=cascadelake

Vectorization with Intel Compiler Flags

- Optimize across objects (e.g. to inline functions)
  - -ip in icc enables inter-procedural optimization within files, while keeping the original line number for debugging
  - -ipo in icc produces optimizations which combine code in different files
  - -fwhole-program in gcc

- To disable vectorization (for rough estimation of vectorization)
  - -no-vec in icc
  - -fno-tree-vectorize (after –O3) in gcc

Vectorization with Intel Compiler Flags

- Use optimization report options for info on vectorization

  icc –O3 –qopt-report=2 –qopt-report-phase=vec source.c

  =n controls the amount of detail in source.optrpt

  n = 0 : No vector report

  n = 1 : List the loops that were vectorized

  n = 2 : adds the loops that were not vectorized with a reason

  n = 3 : adds summary information from the vectorizer about all loops

  n = 4 : adds verbose information from the vectorizer about all loops

  n = 5 : adds details about any data dependencies encountered.

- GCC: -fopt-info-vec and –fopt-info-vec-missed

  -ftree-vectorizer-verbose=n (before gcc-4.9)

More on Intel Compiler Flags

- -qopt-prefetch=n enables various levels of data prefetching. n ranges from 0 to 4. n=3 is included in –O2
- -no-prec-div enables optimizations that give slightly less precise results than full IEEE division
- -fp-model fast=1|2 requests more aggressive (value unsafe)optimization for floating point math
- -fast = -O3 –xHOST –ipo –static –no-prec-div –fp-model fast=2 Notice that the –static flag may cause a linking error as most libraries are dynamic now.
- -qopenmp enables parallelizer to generate multithreaded code based on the OpenMP directives
- -mp1 improves floating point precision and consistency at a small cost to speed

More on Intel Compiler Flags

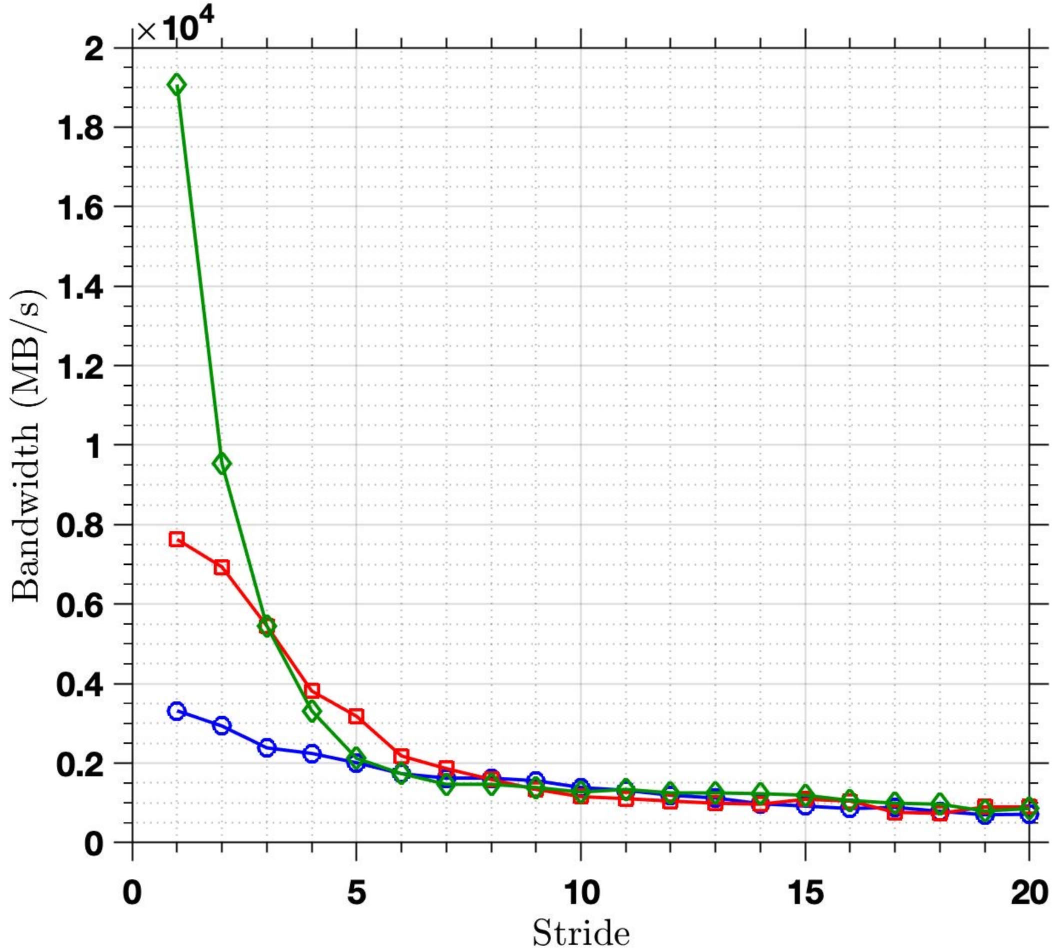
Using fast math options will affect the floating point arithmetic. This may cause inconsistent results or results that are not reproductible across architectures. To avoid the inconsistency, you may consider the options below. (search onlien for gcc counterparts.)

- -fp-model precise disable optimizations that are not value safe on floating point operations
- -fitconsistency enables improved floating point consistency. This may slightly reduce execution speed
- -fp-speculation=strict tells the compiler to disable speculation on floating point operations.

Compiler optimization

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define PTS 10000000
#define MXSTRIDE 20
void main() {
  unsigned long i, istride, msec;
  double *data, cnorm, mean, rate;
  struct timeval tm1, tm2;
  data = malloc(PTS*MXSTRIDE*sizeof(double));
  cnorm = 1.0 / (RAND_MAX + 1.0);
  for (i = 0; i <= PTS*MXSTRIDE; i++) {
    data[i] = (rand() + 1) * cnorm;
  }
  for (istride = 1; istride <= MXSTRIDE; istride++) {
    mean = 0.;
    gettimeofday(&tm1, NULL);
    for (i = 0; i < PTS*istride; i += istride) {
      mean = mean + data[i];
    }
    gettimeofday(&tm2, NULL);
    mean = mean / PTS;
    msec = 1000 * (tm2.tv_sec - tm1.tv_sec) + \
        (tm2.tv_usec - tm1.tv_usec) / 1000;
    rate = sizeof(double)*PTS*(1000.0/msec) / (1024*1024);
    printf("stride %2lu, mean %f, ", istride, mean);
    printf("time %3lu msec, rate %4.0f MB/s\n", msec, rate);
  }
}
```

Compiler optimization



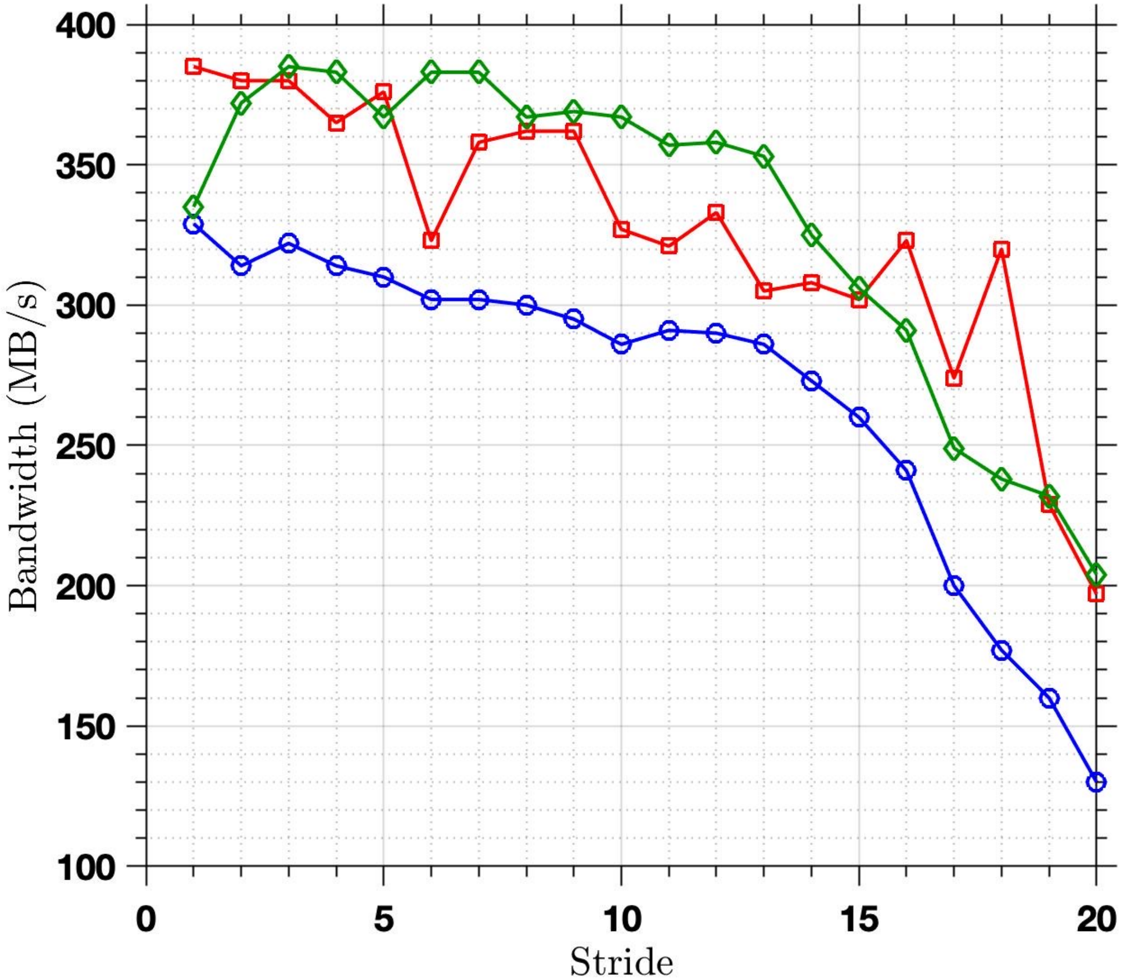
gcc –O0

gcc –O2

gcc –O3 –march=native –ffast-math

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#define PTS 10000000
#define MXSTRIDE 20
void main() {
  unsigned long i, k, istride, msec;
  double *data, cnorm, mean, stddev, u1, u2, xk, deltmean, recipk, rate;
  double two_pi = 2.0 * M_PI;
  struct timeval tm1, tm2;
  data = malloc(PTS*MXSTRIDE*sizeof(double));
  cnorm = 1.0 / (RAND_MAX + 1.0);
  for (i = 0; i <= PTS*MXSTRIDE; i++) {
    data[i] = (rand() + 1) * cnorm;
  }
  for (istride = 1; istride <= MXSTRIDE; istride++) {
    k = 1;
    stddev = 0;
    mean = 0.;
    u1 = (rand() + 1) * cnorm;  // need one extra data point
    gettimeofday(&tm1, NULL);
    for (i = 0; i < PTS*istride; i += istride) {
      // convert data to normal distribution using Box-Muller
      u2 = u1;
      u1 = data[i];
      xk = sqrt(-2.0 * log(u1)) * cos(two_pi * u2);
      // compute the cumulateive mean and standard deviation*
      deltmean = xk - mean;
      recipk = 1.0/k;
      mean = mean + deltmean*recipk;
      stddev = stddev + (k - 1)*(deltmean*deltmean)*recipk;
      k += 1;
    }
    gettimeofday(&tm2, NULL);
    stddev = stddev / k;
    msec = 1000 * (tm2.tv_sec - tm1.tv_sec) + \
        (tm2.tv_usec - tm1.tv_usec) / 1000;
    rate = sizeof(double)*PTS*(1000.0/msec) / (1024*1024);
    printf("stride %2lu, mean %9.6f, stddev %8.6f ", istride, mean, stddev);
    printf("time %3lu msec, rate %4.0f MB/s\n", msec, rate);
  }
// *http://www.cs.berkeley.edu/~mhoemmen/cs194/Tutorials/variance.pdf
}
```

Compiler optimization



gcc –O0

gcc –O2

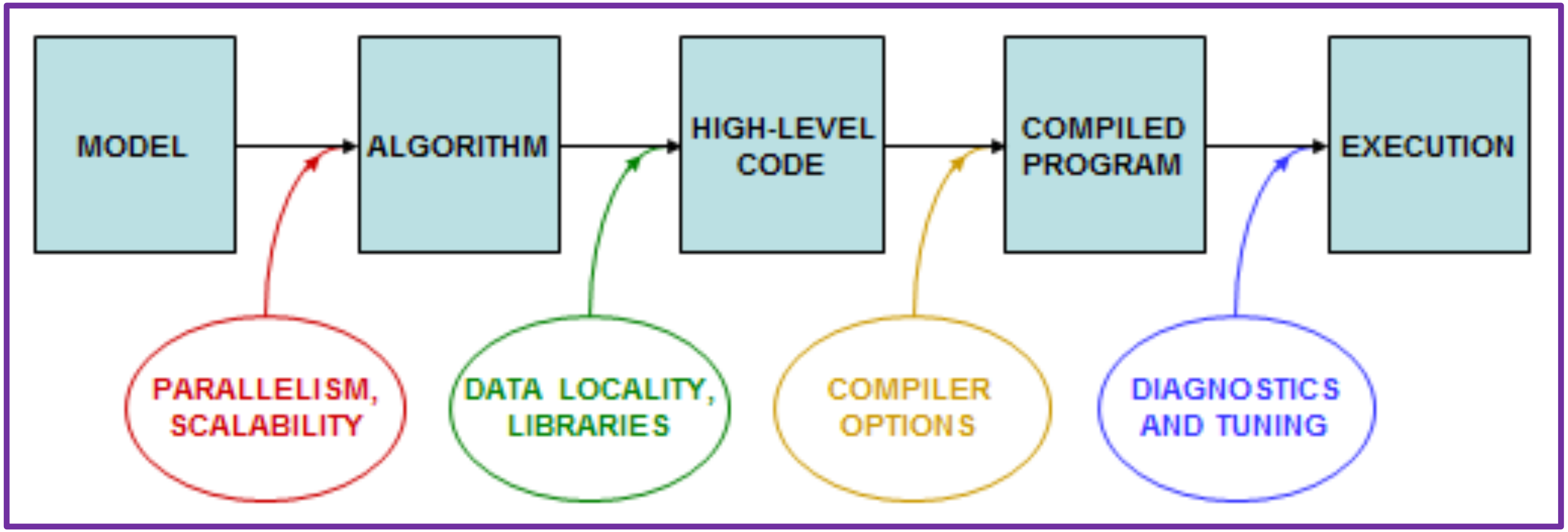
gcc –O3 –march=native –ffast-math

Summary

Practical Tips

- Check the system details thoroughly
- Choose a compiler to build your application and be aware the differences among compilers
- Start with some basic flags and try additional flags one at a time (incremental optimization)
- Use the built-in libraries and tools to save time and improve performance (MKL)
- You may influence and hint the compiler in your code style
- The only way to know if it works is to run experiments and find out yourself.
- You can generate an assembly code to see if desired optimiation is turned on.

A big picture



Improve single-core performance
1. write a code with good locality
2. Employ optimized HPC libraries wherever possible
3. Using appropriate compiler flags when building your code

Practical Tips

- `cat /proc/cpuinfo`

  provide processor details

- `cat /proc/meminfo`

  provide the memory details

- `/usr/sbin/ilstat`

  provide the interconnect IB fabric details

- `/usr/bin/lscpu`

  show cpu details including cache sizes

References

- An Introduction to GCC for the GNU compilers gcc and g++ by B. Gough
- Using the GNU Compiler Collection
- Introduction to High Performance Scientific Computing by V. Eijkhout
- Manuals of the Intel Compiler