中国科学院大学

University of Chinese Academy of Sciences

高性能计算编程

编译、调试和构建工具

授课教师：李会元 黎雷生

2025/03/05

中国科学院大学
University of Chinese Academy of Sciences

目录
Contents

中国科学院大学
University of Chinese Academy of Sciences

01 编译链工具

GCC与编译基本过程

- GCC(GNU Compiler Collection)编译工具链包含了GCC编译器在内的一整套工具，主要包含了GCC编译器、Binutils工具集、glibc标准函数库。一般情况下，我们说的GCC编译工具链就是指GCC编译器。
- GCC原名为GNU C语言编译器（GNU C Compiler），只能对C语言进行编译等处理。后来随着其功能的扩展，可以支持更多编程语言，如C++、Fortran、Go以及各类处理器架构上的汇编语言等。

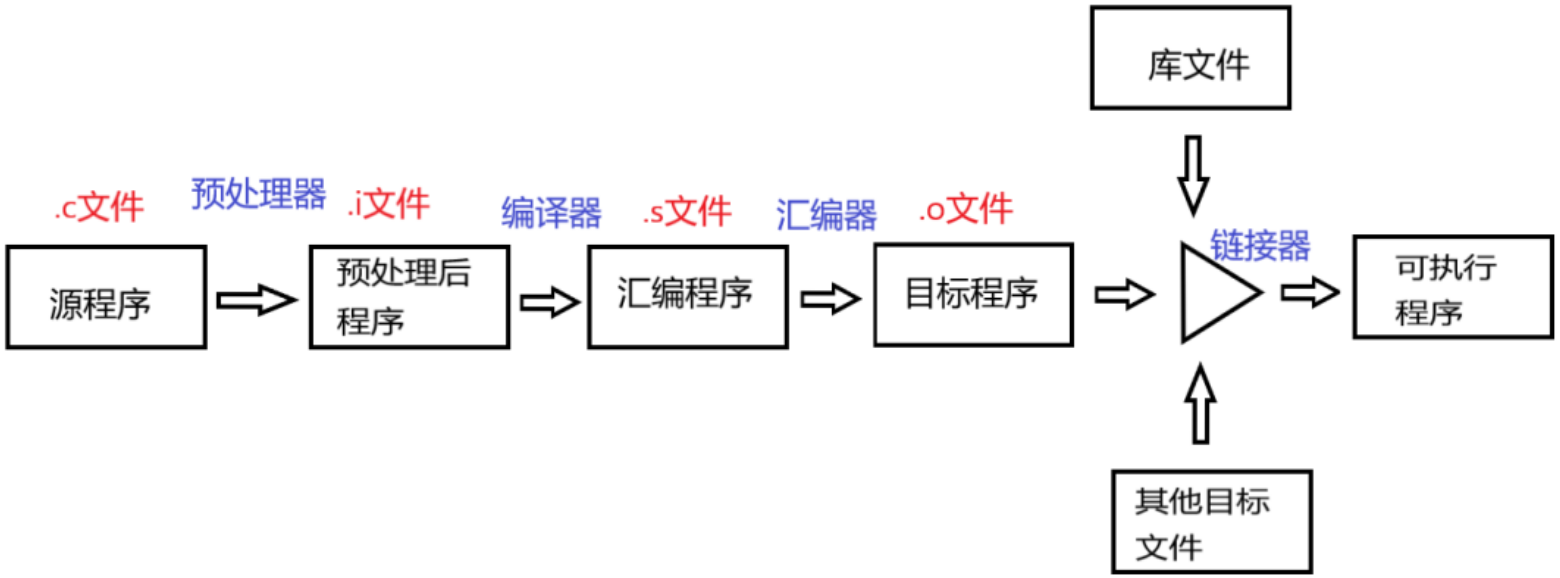
◘GCC常用编译选项

选项	作用解析
-E	预处理生成 .i 文件
-S	编译生成 .s 汇编文件
-c	汇编生成 .o 目标文件
-o	指定目标文件
-O	优化选项，有1-3级
-D	指定宏
-I (大写i)	指定包含头文件的路径（绝对、相对路径都可）
-l (小写L)	指定库名，libxxx.a或libxxx.so
-L	包含的库路径
-g	生成调试信息，用于gdb调试。
-Wall	显示更多警告信息。-W{warning}来打开指定的warning。-Wno-{warning}关闭指定的warning。
-Werror	warning信息当作error，强制修改warning问题。

◘实验环境

服务器	项目	配置
Intel-V100	CPU	Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
	操作系统	CentOS Linux release 7.9.2009; Linux v100 3.10.0-1062.4.1.el7.x86_64
	GCC/gdb	gcc version 11.2.0 (GCC)
	binutils	2.27.x86_64.44.base.el7_9.1
	Intel Compiler	icc version 19.0.0.117
	clang/clang++/flang	AMD clang version 17.0.0
	make	GNU make 4.4
	cmake	cmake version 3.24.0-rc3

- 编译一般流要经过预处理、编译、汇编这么一系列步骤才生成目标文件。
  - 是因为在每一阶段都有相应的优化技术，只有在每个阶段分别优化并生成最为高效的机器指令才能达到最大的优化效果。
  - 更容易支持多种语言和多种指令架构。



- 需要注意的是，后续过程默认包含前置过程，编译器一般不会明确提示编译的流程。

- 预处理（Preprocess，调用cpp）：由预处理器完成，对源程序中的伪指令（以#开头的指令）和特殊符号进行处理，伪指令包括宏定义指令、条件编译指令和头文件中包含的指令。主要功能：
  - 将所有的#define删除，并将宏定义进行宏展开；
  - 处理所有条件编译指令，如#if、#ifdef、#ifndef、#else、#elif、#endif等；
  - 处理其他宏处理 #include预编译指令，将被包含的头文件内容插入该预编译指令的位置，如果是多重包含的话会递归执行；
  - 指令，包括#error、#warning、#line、#pragma；
  - 处理所有注释（C++的//，C语言的/**/），一般会用一个空格来代替连续的注释；
  - 添加行号和文件标识，以便于编译时编译器产生调试用的行号信息及编译时产生编译错误和警告时可以把行号打印出来；
  - 保留所有的#pragma编译器指令；
  - 处理预定义的宏：如__DATE__、__FILE__等；

预处理

```
$ g++ -E preproc.cpp
```

```
#define VEC_LENGTH 128
 double a[VEC_LENGTH];
 double b[VEC_LENGTH];
 double c[VEC_LENGTH];
#ifdef TIMING
 size_t start = GetUsec();
#endif
 vecadd(a, b, c, VEC_LENGTH);
#ifdef TIMING
 size_t finish = GetUsec();
 printf("Timing = %ldus\n",finish - start);
#endif
```

```
double a[128];
double b[128];
double c[128];


vecadd(a, b, c, 128);
```

宏定义

```
$ g++ -E preproc.cpp -DTIMING
```

```
#define VEC_LENGTH 128
 double a[VEC_LENGTH];
 double b[VEC_LENGTH];
 double c[VEC_LENGTH];
#ifdef TIMING
 size_t start = GetUsec();
#endif
 vecadd(a, b, c, VEC_LENGTH);
#ifdef TIMING
 size_t finish = GetUsec();
 printf("Timing = %ldus\n",finish - start);
#endif
```

```
double a[128];
double b[128];
double c[128];

size_t start = GetUsec();

vecadd(a, b, c, 128);

size_t finish = GetUsec();
printf("Timing = %ldus\n",finish - start);
```

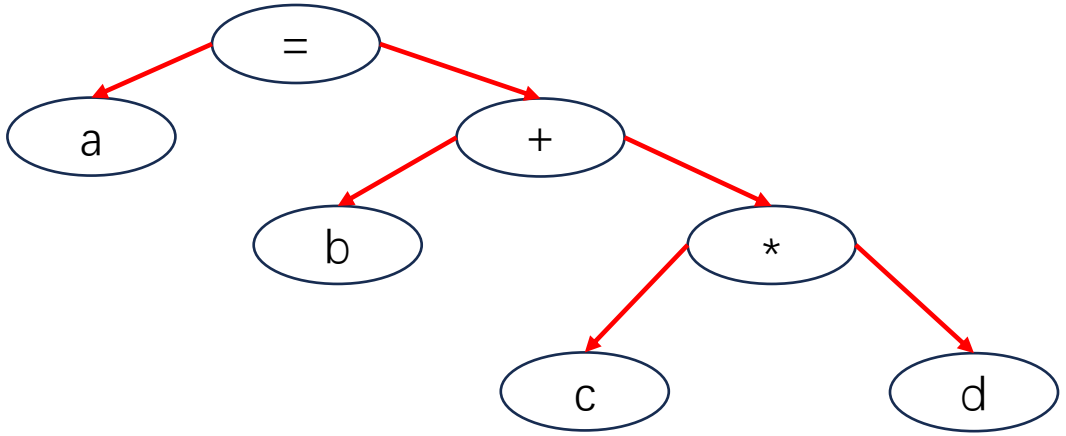
编译

- 编译（Compilation，调用cc）：由编译器完成，对预处理后的文件进行词法分析、语法分析、语义分析以及优化后生成相应的汇编代码文件。
- 词法分析：源代码的字符序列分割成一系列的记号(Token)。

```
a = b + c * d;
```

```
“a” “=“ ”b” ”+” ”c” ”*” ”d” ”;”
```

- 语法分析：根据语法规则，确定程序的语法结构，把词法分析阶段得到的token序列组合成表达式(expression)、语句(statement), 函数(function)等各种语法单元。



- 语义分析：在语法分析阶段构造出来的抽象语法树的基础上，对程序进行静态语义的分析，确保程序不仅符合语法规范，还应该能够正常执行。

编译

- 中间代码生成：中间代码是一种在语法结构和数据类型更接近目标语言的表现形式。它处理起来要比源语言更加简单，便于进行逻辑优化，更加容易生成简洁、高效的目标代码。

- 代码优化：为了最终生成的程序能够更高效的执行和存储，现代编译器往往会花很大力气去对代码进行优化，而且会不止一次地对代码进行扫描和优化。

- 目标代码生成：以优化后的中间代码作为输入，产生等价的目标代码作为输出。代码生成的过程中往往也伴随着代码优化，要充分考虑到对硬件资源的最优化使用，如寄存器、cache、流水线等。

编译

```
$ g++ -S -masm=intel vecmatrix.cpp -I./include
```

```
double a[128];
double b[128];
double c[128];


vecadd(a, b, c, 128);
```

```
mov   rbp, rsp
sub   rsp, 101408
mov   DWORD PTR [rbp-101396], edi
mov   QWORD PTR [rbp-101408], rsi
lea   rdx, [rbp-3088]
lea   rsi, [rbp-2064]
lea   rax, [rbp-1040]
mov   ecx, 128
mov   rdi, rax
call  _Z6vecaddPdS_S_m
```

汇编

- 汇编（Assembly，调用as）:将汇编代码转变成机器可执行的二进制代码（机器码），并生成目标文件。

从汇编文件

```
$ g++ -c vecmatrix.s
```

```
$ ls -l vecmatrix.o
-rw-rw-r-- 1 iscashpc iscashpc 2792 Mar  1 09:40 vecmatrix.o
```

从源代码文件

```
$ g++ -c vecmatrix.cpp -I./include
$ g++ -c ./src/matrix.cpp -I./include
```

链接

- 链接(Link，调用ld)：链接是将多个可重定位文件进行空间和地址重分配，符号解析和重定位最终组成一个可执行目标文件的过程。

```
$ g++ -o vecmatrix  ./vecmatrix.o
vecmatrix.o: In function `main':
vecmatrix.cpp:(.text+0xe1): undefined reference to `SimpleMultiply(double const*, double const*, double*, unsigned long, unsigned long, unsigned long)'
collect2: error: ld returned 1 exit status
```

```
$ g++ -o vecmatrix  ./vecmatrix.o ./matrix.o
```

```
$ ls -l vecmatrix vecmatrix.o matrix.o
-rw-rw-r-- 1 iscashpc iscashpc 1496 Mar  1 10:13 matrix.o
-rwxrwxr-x 1 iscashpc iscashpc 8520 Mar  1 10:14 vecmatrix
-rw-rw-r-- 1 iscashpc iscashpc 2568 Mar  1 10:12 vecmatrix.o
```

链接

对于标准库等常用库，编译器默认自动添加。

```
$ ldd vecmatrix
```

```
linux-vdso.so.1 =>  (0x00007ffc4c9ab000)
libstdc++.so.6 => /usr/local/lib64/libstdc++.so.6 (0x00007f576cbb9000)
libm.so.6 => /lib64/libm.so.6 (0x00007f576c8b7000)
libgcc_s.so.1 => /usr/local/lib64/libgcc_s.so.1 (0x00007f576c69f000)
libc.so.6 => /lib64/libc.so.6 (0x00007f576c2d1000)
/lib64/ld-linux-x86-64.so.2 (0x00007f576cfbf000)
```

- 静态链接库
  - 可重定位目标文件以一种特定的方式打包成一个单独的文件，并且在链接生成可执行文件时，从这个单独的文件中“拷贝” 需要的内容到最终的可执行文件中。
- 静态链接库的编译和链接

```
$ g++ -c ./src/matrix.cpp -I./include -DTIMING
$ ar rc ./lib/libmatrix.a matrix.o
$ g++ -o vecmatrix.static vecmatrix.o -lmatrix -L./lib/
$ g++ -o vecmatrix matrix.o vecmatrix.o
```

- 对比静态库链接和.o文件链接

```
$ ls -l vecmatrix vecmatrix.static
-rwxrwxr-x 1 iscashpc iscashpc 8672 Mar  2 12:20 vecmatrix
-rwxrwxr-x 1 iscashpc iscashpc 8672 Mar  2 12:18 vecmatrix.static
```

- 优点：静态库被打包到应用程序中加载速度快； 发布程序无需提供静态库。
- 缺点：相同的库文件数据可能在内存中被加载多份，消耗系统资源，浪费内存； 库文件更新需要重新编译项目文件。

- 动态链接库
  - 在链接时不将所有二进制代码都“拷贝”到可执行文件中，而是仅仅“拷贝”一些重定位和符号表信息，这些信息可以在程序运行时完成真正的链接过程。
- 动态链接库的编译和链接

```
$ g++ -c ./src/matrix.cpp -fPIC -shared -o matrix.shared.o -I./include/
$ g++ -shared -o ./lib/libmatrix.so matrix.shared.o
$ g++ -o vecmatrix.dyn vecmatrix.o -lmatrix -L./lib/
```

- 对比动态链接和静态链接

```
$ ls -l vecmatrix.static vecmatrix.dyn
-rwxrwxr-x 1 iscashpc iscashpc 8656 Mar  2 12:27 vecmatrix.dyn
-rwxrwxr-x 1 iscashpc iscashpc 8672 Mar  2 12:18 vecmatrix.static
```

- 优点： 可实现不同进程间的资源共享； 动态库升级简单，只需要替换库文件，无需重新编译应用程序；可以控制何时加载动态库，不调用库函数动态库不会被加载。
- 缺点：加载速度比静态库慢； 发布程序需要提供依赖的动态库。

运行时使用动态链接库

•动态链接库的目录需要配置在LD_LIBRARY_PATH环境变量。

```
[iscashpc@v100 g++code]$ ./vecmatrix.dyn
./vecmatrix.dyn: error while loading shared libraries:
 libmatrix.so: cannot open shared object file: No such
 file or directory
[iscashpc@v100 g++code]$ export LD_LIBRARY_PATH=/home/
iscashpc/g++code/lib:$LD_LIBRARY_PATH
[iscashpc@v100 g++code]$ ./vecmatrix.dyn
Timing = 1us
aaa = 0
```

•扩展学习：使用dlopen dlsym dlerror dlclose等系统调用使用动态库。

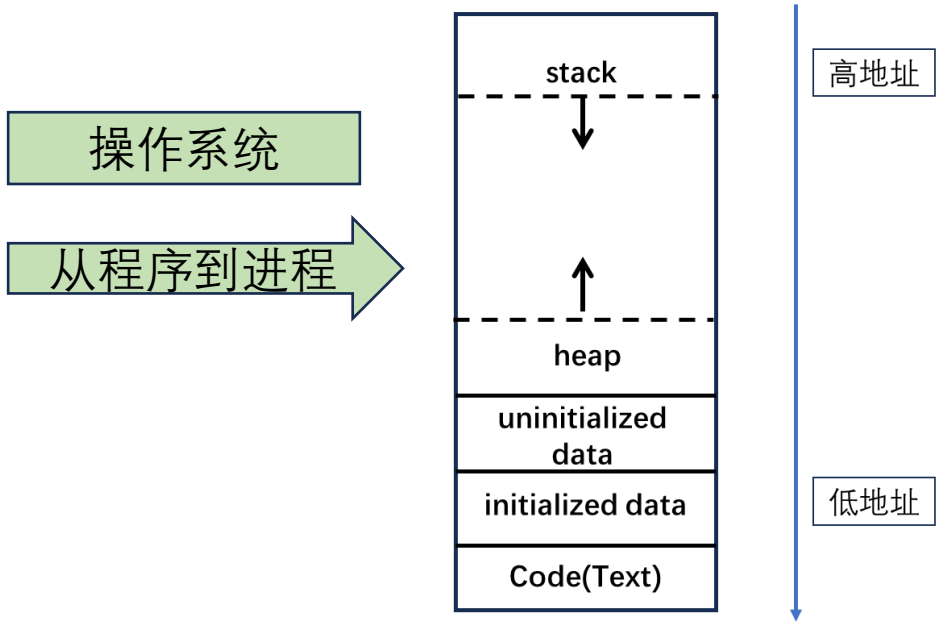
linux动态加载动态链接库—— dlopen()_linux dlopen-CSDN博客

可执行程序与进程

ELF可执行文件格式

- ELF的全程是Executable Linkable Format，是一种二进制文件格式。形式化的规范允许操作系统正确解释机器指令。组织可执行程序的代码、数据和库等。
- 对于可执行文件，通常包含 .text, .data, .rodata, .bss等部分。

.text	包含可执行代码。
.data	初始化数据，可读写。
.rodata	初始化数据，只读。
.bss	未初始化数据，可读写。



- Binutils工具集
  - size： 列出文件每个部分的内存大小，如代码段、数据段、总大小等。
  - readelf： 显示有关ELF（Executable and Linkable Format）格式文件内容的信息。ELF格式是UNIX系统实验室作为应用程序二进制接口开发的。
  - nm： 查看目标文件中出现的符号(函数、全局变量等)。
  - objcopy： 将目标文件的一部分或者全部内容拷贝到另外一个目标文件中，或者实现目标文件的格式转换。可用于目标文件格式转换，如.bin转换成.elf、.elf转换成.bin等。
  - objdump： 显示程序文件相关信息，最主要的作用是反汇编。
  - addr2line： 将程序地址翻译成文件名和行号；给定地址和可执行文件名称，它使用其中的调试信息判断与此地址有关联的源文件和行号，通常搭配 nm 使用。
  - strings： 显示程序文件中的可显示字符串。
  - ar：创建和管理静态库。
  - ranlib：命令用于 更新库的符号索引表，相当于ar –S。

GCC优化编译选项

- 编译warning和error

```
$ g++ -c warningerror.cpp -Wall
vecmatrix.cpp: In function 'int main(int, char**)':
vecmatrix.cpp:34:7: warning: 'aaa' may be used uninitialized [-Wmaybe-
uninitialized]
   34 |   c[0]=aaa;
      |   ~~~~^~~~
```

```
$ g++ -c warningerror.cpp -Wall -Werror
vecmatrix.cpp: In function 'int main(int, char**)':
vecmatrix.cpp:34:7: error: 'aaa' may be used uninitialized [-Werror=maybe-
uninitialized]
   34 |   c[0]=aaa;
      |   ~~~~^~~~
cc1plus: all warnings being treated as errors
```

- -W{warning}来打开指定的warning。-Wno-{warning}关闭指定的warning。

调试选项

```
$ g++ -o vecmatrixopt.O0.s –O0 -S vecmatrixopt.cpp
```

```
$ g++ -o vecmatrixopt.g.s -g -S vecmatrixopt.cpp
```

- vimdiff对比两条命令产生的汇编文件。观察到-g产生辅助调试的代码。

优化选项

```
$ g++ -o vecmatrixopt.O0.s -S vecmatrixopt.cpp //默认-O0
```

```
$ g++ -o vecmatrixopt.O1.s -O1 -S vecmatrixopt.cpp
```

```
$ g++ -o vecmatrixopt.O2.s –O2 -S vecmatrixopt.cpp
```

```
$ g++ -o vecmatrixopt.O3.s -O3 -S vecmatrixopt.cpp
```

- vimdiff对比不同级别优化选项命令产生的汇编文件。可以观察的部分优化效果。

- -Ofast：激进优化可能违反语言标准。包括-O3 -ffast-math。
- -funroll-loops：使能循环展开(loop unrolling)

◘处理器架构标识，面向架构的优化(没有被包含在-O3)。

• -march=<arch>： 探索特定处理器硬件特征，为特定处理器产生指令。<arch> 表示二进制支持的最低要求 (不可移植)。
• -mtune=<tune arch>：指定目标微架构。为一类处理器产生优化代码不探索特定的有硬件特征。二进制仍然兼容其他处理器，如同一架构系列的CPUs（可能比-march稍慢）。

•注意:
• <tune arch> 应该总是大于<arch>。
•如果没有指定，-mtune通常设置为“generic”。
• -march=native , -mtune=native , -mcpu=native， 允许编译器决定架构(不总是准确)。

浮点运算标识

•通常来说，使能以下的标识会减低浮点精度，破坏IEEE754标准，并且时实现相关的（没有被包括在O3）。

- -fno-signaling-nans：不发送NaN信号。
- -fno-trapping-math：关闭浮点异常。
- -mfma -ffp-contract=fast ： 强制浮点表达式收缩，例如形成融合乘加运算。
- -ffinite-math-only：关闭特定情况下处理inf和NaN。
- -funsafe-math-optimizations：允许破环结合律和使能倒数优化。
- -ffast-math：使能激进浮点优化。所有前面的标识，加上flush-to-zero非规格化数。

探索优化选项

```
$ g++  -Q  --help=optimizer -O2
```

•列出优化级别使用的优化方法。

```
[iscashpc@v100 g++code]$ g++  -Q  --help=optimizer -O1 |more
The following options control optimizations:
  -O<number>
  -Ofast
  -Og
  -Os
  -faggressive-loop-optimizations       [enabled]
  -falign-functions                     [disabled]
  -falign-functions=
  -falign-jumps                         [disabled]
  -falign-jumps=
  -falign-labels                        [disabled]
  -falign-labels=
  -falign-loops                         [disabled]
  -falign-loops=
  -fallocation-dce                      [enabled]
```

```
[iscashpc@v100 g++code]$ g++  -Q  --help=optimizer -O2
The following options control optimizations:
  -O<number>
  -Ofast
  -Og
  -Os
  -faggressive-loop-optimizations       [enabled]
  -falign-functions                     [enabled]
  -falign-functions=                    16
  -falign-jumps                         [enabled]
  -falign-jumps=                        16:11:8
  -falign-labels                        [enabled]
  -falign-labels=                       0:0:8
  -falign-loops                         [enabled]
  -falign-loops=                        16:11:8
  -fallocation-dce                      [enabled]
```

截图(Alt + A)

•-f控制独立的选项。如-O1 –falign-functions 会启用align-functions， -O2 -fno-align-functions 禁用align-functions。

输出编译过程优化信息

```
$ g++ -O3 -S vecmatrixopt.cpp -fopt-info
```

```
vecmatrixopt.cpp:39:26: optimized:  Inlining int64_t GetUsec()/556 into int
main(int, char**)/559.
vecmatrixopt.cpp:38:9: optimized:  Inlining int vecadd(double*, double*,
double*, size_t)/557 into int main(int, char**)/559.
vecmatrixopt.cpp:37:25: optimized:  Inlining int64_t GetUsec()/556 into int
main(int, char**)/559.
vecmatrixopt.cpp:51:1: optimized:  Inlining void
__static_initialization_and_destruction_0(int, int)/1067 into (static
initializers for vecmatrixopt.cpp)/1186.
vecmatrixopt.cpp:15:23: optimized: loop vectorized using 16 byte vectors
vecmatrixopt.cpp:15:23: optimized: loop turned into non-loop; it never loops
vecmatrixopt.cpp:15:23: optimized: loop vectorized using 16 byte vectors
```

其他编译器、Fortran语言

LLVM和Intel Compiler

- clang和clang++是基于LLVM(Low Level Virtual Machine)的C/C++编译器。它与GNU C语言规范几乎完全兼容，并在此基础上增加了额外的语法特性。
- icc icpc是Intel开发的c c++编译器，对Intel CPU提供良好的支持。

•编译器较好支持C/C++语言标准，使用方法比较一致。对于编译器之间的差异，查询帮助文档比较解决。

•OpenBLAS是应用广泛的高性能BLAS(Basic Linear Algebra Subprograms)开源库。以编译OpenBLAS-0.3.26选项为例比较不同编译器使用的选项。

- `gcc -c -O2 -Wall -m64 -fPIC -msse3 -mssse3 -msse4.1 -mavx -mavx2 -march=skylake-avx512`
- `icc -c -O2 -fPIC -msse3 -mssse3 -msse4.1 -mavx -mavx2 -march=skylake-avx512`
- `clang -O2 -Wall -m64 -fPIC -msse3 -mssse3 -msse4.1 -mavx -mavx2 -march=skylake-avx512`

Fortran语言

- FORTRAN是英文“FORmulaTRANslator”的缩写。Fortran语言的最大特性是接近数学公式的自然描述，可以直接对矩阵和复数进行运算。自诞生以来广泛地应用于数值计算领域，积累了大量高效而可靠的源程序。很多专用的大型数值运算计算机针对Fortran做了优化。Fortran90，Fortran95，Fortran2003的相继推出使Fortran语言具备了现代高级编程语言的一些特性。

Fortran编译环境

•gfortran ifort flang分别是GCC LLVM Intel编译系统的Fortran编译器。

- OpenBLAS-0.3.26 Fortran代码编译选项。
  - `gfortran -O2 -Wall -frecursive -fno-optimize-sibling-calls -m64 -fPIC -msse3 -mssse3 -msse4.1 -mavx -mavx2 -march=skylake-avx512 -mavx2 -fno-tree-vectorize`
  - `ifort -O2 -recursive -fp-model strict -assume protect-parens -fPIC -msse3 -mssse3 -msse4.1 -mavx -mavx2 -march=skylake-avx512`
  - `flang -O2 -Mrecursive -Kieee -fno-unroll-loops -Wall -fPIC -msse3 -mssse3 -msse4.1 -mavx -mavx2`

•链接库-lgfortran –lifort -lflang

- 扩展学习
  - Quickstart tutorial — Fortran Programming Language (fortran-lang.org)
  - Compiler options comparison — The Flang Compiler (llvm.org)

02

调试工具

软件缺陷和处理方式

- 错误、缺陷和失效
  - 错误，人类的失误导致产生与期望不同的后果。错误导致软件缺陷。
  - 缺陷，不期望的软件行为（正确性、性能等）。缺陷潜在导致软件失效。
  - 失效，可观察到的软件不正确行为。

- 处理软件缺陷的方式
  - 动态分析
    - 技术：print, run-time debugging, sanitizers, fuzzing, unit test support, performance regression tests
    - 限制：不能覆盖所有程序状态。
  - 静态分析：主动检查源代码潜在的错误。
    - 技术：warnings, static analysis tool, compile-time checks
    - 限制：Turing’s undecidability theorem, exponential code paths

Assertion（断言）

- 断言是发现违反前提的语句。断言代表代码中的不变量。
- 可以是运行时(assert)和编译时(static_assert)。
- 运行时断言失效不应该在正常运行的程序中暴露出来。
- 断言可能拖慢执行，通过定义NDEBUG(#define NDEBUG)宏禁用断言。

```
# include <cassert> // <-- needed for "assert"
# include <cmath> // std::is_finite
# include <type_traits> // std::is_arithmetic_v
template<typename T>
T sqrt(T value) {
static_assert(std::is_arithmetic_v<T>, // precondition
"T must be an arithmetic type");
assert(std::is_finite(value) && value >= 0); // precondition
int ret = ... // sqrt computation
assert(std::is_finite(value) && ret >= 0 && // postcondition
(ret == 0 || ret == 1 || ret < value));
return ret;
}
```

执行时调试(gdb)

```
$ g++ -O0 -g [-g3] <program.cpp> -o program
$ gdb [--args] ./program <args...>
```

- -O0:禁用任何代码优化。
- -g :启用调试。
  - 在可执行程序存储符号表（映射汇编和源代码行）。
  - 拖慢编译和执行阶段
- -g3:产生增强调试信息。

gdb – Breakpoints

Command	Abbr.	Description
breakpoint *<file>:<line>*	b	insert a breakpoint in a specific line
breakpoint *<function name>*	b	insert a breakpoint in a specific function
breakpoint *<ref >* **if** *<condition>*	b	insert a breakpoint with a conditional statement
delete	d	delete all breakpoints or watchpoints
delete *<breakpoint number>*	d	delete a specific breakpoint
clear [*function name/line number*]		delete a specific breakpoint
enable/disable *<breakpoint number>*		enable/disable a specific breakpoint
info breakpoints	info b	list all active breakpoints

gdb – Watchpoints/Catchpoints

Command	Abbr.	Description
watch *<expression>*		stop execution when the value of expression changes(variable, comparison, etc.)
rwatch *<variable/location>*		stop execution when variable/location is read
delete *<watchpoint number>*	d	delete a specific watchpoint
info watchpoints		list all active watchpoints
catch throw		stop execution when an *exception* is thrown

gdb – Control Flow

Command	Abbr.	Description
run [args]	r	run the program
continue	c	continue the execution
finish	f	continue until the end of the current function
step	s	execute next line of code (follow function calls)
next	n	execute next line of code
until *<program point>*		continue until reach line number, function name, address, etc.
CTRL+C		stop the execution (not quit)
quit	q	exit
help [*<command>*]	h	

gdb – Stack and Info

Command	Abbr.	Description
list	l	print code
list *<function or #start,#end>*	l	print function/range code
up	u	move up in the call stack
down	d	move down in the call stack
backtrace [full]	bt	prints stack backtrace (call stack) [local vars]
info args **info locals** **info variables**		print current function arguments print local variables print all variables
info *<breakpoints/watchpoints/registers>*		show information about program breakpoints/watchpoints/registers

gdb – Print and set

Command	Abbr.	Description
print *<variable>*	p	print variable
print/h *<variable>*	p/h	print variable in hex
print/nb *<variable>*	p/nb	print variable in binary (n bytes)
print/w *<address>*	p/w	print address in binary
p /s *<char array/address>*		print char array
p *array var@n		print n array elements
p (int[4])*<address>*		print four elements of type int
p *(char)&***<std::string>*		print std::string

Command	Abbr.	Description
set variable*<variable=EXP>*		To change the value of variable in gdb and continue execution with changed value. with $), a register (a few standard names starting with $), or an actual variable in the program being debugged. EXP is any valid expression.
set subcommand		With a subcommand, this command modifies parts of the gdb environment.

gdb – Disassemble

Command	Abbr.	Description
disasseble *<function name>*		disassemble a specified function
disasseble <0xStart,0xEnd addr>		disassemble function range
nexti *<variable>*		execute next line of code (follow function calls)
stepi *<variable>*		execute next line of code
p /s *<char array/address>*		print char array
x/nfu *<address>*		examine address n number of elements, f format (**d**: int, **f**: float, etc.), u data size (**b**: byte, **w**: word, etc.)

Debugging using gdb

```
[iscashpc@v100 gdbcode]$ g++ -g debugcode.cpp -o debugcode.g
[iscashpc@v100 gdbcode]$ gdb ./debugcode.g
```

```
(gdb) b main
Breakpoint 1 at 0x400855: file debugcode.cpp, line 37.
(gdb) b SimpleMultiply(double const*, double const*, double*
ned long, unsigned long, unsigned long)
Breakpoint 2 at 0x40074b: file debugcode.cpp, line 22.
(gdb) list
23          for (int col = 0; col < K; ++col) {
24            double sum = 0.0;
25            for (int k = 0; k < N; ++k) {
26              sum += a[row * K + k] * b[k * N + col];
27            }
28            c[row * N + col] = sum;
29          }
30        }
31      }
32
(gdb) b debugcode.cpp:26 if k=6
Breakpoint 3 at 0x400775: file debugcode.cpp, line 26.
(gdb)
```

```
(gdb) info reg
rax            0x40083d         4196413
rbx            0x0              0
rcx            0x100            256
rdx            0x7fffffffcc18   140737488342040
rsi            0x7fffffffcc08   140737488342024
rdi            0x1              1
```

```
(gdb)  set disassembly-flavor intel
(gdb) disassemble vecadd
Dump of assembler code for function _Z6vecaddPdS_S_m:
   0x00000000004006b2 <+0>:     push   rbp
   0x00000000004006b3 <+1>:     mov    rbp,rsp
   0x00000000004006b6 <+4>:     mov    QWORD PTR [rbp-0x18],rdi
   0x00000000004006ba <+8>:     mov    QWORD PTR [rbp-0x20],rsi
   0x00000000004006be <+12>:    mov    QWORD PTR [rbp-0x28],rdx
   0x00000000004006c2 <+16>:    mov    QWORD PTR [rbp-0x30],rcx
   0x00000000004006c6 <+20>:    mov    QWORD PTR [rbp-0x8],0x0
   0x00000000004006ce <+28>:    jmp    0x40071e <_Z6vecaddPdS_S_m+108>
```

Debugging using checkpoint

- A GDB checkpoint is a separate process that is created by copying the state of the debugged process using the fork() function. Once created the checkpoint process will remain suspended until it is selected using the restart command. You can switch back to the main process by executing the restart command with a checkpoint number of 0

```
Breakpoint 2, vecadd (a=0x7fffffffc760, b=0x7fffffffc360, c=
0x7fffffffbf60, length=128) at ./debugcode.cpp:15
15          for(size_t i = 0; i < length; ++i){
(gdb) checkpoint
checkpoint 1: fork returned pid 98426.
(gdb) n
16            c[i] = a[i] + b[i];
(gdb) n
15          for(size_t i = 0; i < length; ++i){
(gdb) n
16            c[i] = a[i] + b[i];
(gdb) n
15          for(size_t i = 0; i < length; ++i){
(gdb) c
Continuing.
Timing = 13535117us

Breakpoint 3, SimpleMultiply (a=0x7fffffff3f60, b=0x7ffffffe
bf60, c=0x7fffffffe3f60, M=64, N=64, K=64) at ./debugcode.cpp
:22
22          for (int row = 0; row < M; ++row) {
(gdb) n
23            for (int col = 0; col < K; ++col) {
(gdb) n
24              double sum = 0.0;
(gdb) restart 1
Switching to process 98426
#0  vecadd (a=0x7fffffffc760, b=0x7fffffffc360,
    c=0x7fffffffbf60, length=128) at ./debugcode.cpp:15
15          for(size_t i = 0; i < length; ++i){
```

Checkpoint

restart

Reverse Debugging

- The “record” command begins recording the execution of your application, making notes of things like memory and register values. When you arrive at a point in your application at which you’d like to go backwards.
- As you move backwards through code, gdb reverts the state of memory and registers, effectively unexecuting lines of code.

```
Breakpoint 1, vecadd (a=0x7fffffffc760, b=0x7fffffff
c360, c=0x7fffffffbf60, length=128) at debugcode.cpp
:15
15          for(size_t i = 0; i < length; ++i){
(gdb) record
(gdb) n
16            c[i] = a[i] + b[i];
(gdb) n
15          for(size_t i = 0; i < length; ++i){
(gdb) n
16            c[i] = a[i] + b[i];
(gdb) n
15          for(size_t i = 0; i < length; ++i){
(gdb) n
16            c[i] = a[i] + b[i];
(gdb) n
15          for(size_t i = 0; i < length; ++i){
(gdb) p i
$1 = 2
(gdb) reverse-next
16            c[i] = a[i] + b[i];
(gdb) reverse-next
15          for(size_t i = 0; i < length; ++i){
(gdb) p i
$2 = 1
(gdb) s
16            c[i] = a[i] + b[i];
(gdb) record stop
Process record is stopped and all execution logs are
 deleted.
(gdb)
```

Debugging Core Dump

```
int main(int argc, char **argv) {
int *array = new int[100];
array[0] = 0;
int res = array[argc + 100];
for(int i=0; i< 100000;++i){
   array[i+1]=array[i]+1;
}
delete [] array;
return res;
}
```

```
$ ulimit -c 10000000
$ g++ -g heap_buffer_overflow.cpp -o ./heap_overflow
$ ./heap_overflow
Segmentation fault (core dumped)
$ gdb ./heap_overflow ./core.9907
```

```
Reading symbols from ./heap_overflow...
[New LWP 8863]
Core was generated by `./heap_overflow'.
Program terminated with signal SIGSEGV, Segmentation
fault.
#0  0x0000000000400620 in main (argc=1,
    argv=0x7ffdd98cd208)
    at heap_buffer_overflow.cpp:8
8           array[i+1]=array[i]+1;
warning: File "/usr/lib/libstdc++.so.6.0.29-gdb.py" a
uto-loading has been declined by your `auto-load safe
-path' set to "$debugdir:$datadir/auto-load".
```

Debugging optimized code(—)

```
$ g++ -O3 -o debugcode.O3 debugcode.cpp
```

```
(gdb) b vecadd
Breakpoint 3 at 0x4007c0
(gdb) b GetUsec
Function "GetUsec" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
```

```
(gdb) x/8i 0x4005e0
   0x4005e0 <main>:     push   rbx
   0x4005e1 <main+1>:   xor    esi,esi
   0x4005e3 <main+3>:   sub    rsp,0x18c10
   0x4005ea <main+10>:  lea    rdi,[rsp+0x10c10]
   0x4005f2 <main+18>:  call   0x4005a0 <gettimeofday@plt>
   0x4005f7 <main+23>:  xor    eax,eax
   0x4005f9 <main+25>:  imul   rbx,QWORD PTR [rsp+0x10c10],0xf4240
   0x400605 <main+37>:  add    rbx,QWORD PTR [rsp+0x10c18]
(gdb) b *0x4005ea
Breakpoint 2 at 0x4005ea
```

```
Breakpoint 1, 0x00000000004005e0 in main ()
(gdb) p $eax
$1 = 4195808
(gdb) set $eax=0x6666
(gdb) p $eax
$2 = 26214
(gdb) p/x $eax
$3 = 0x6666
(gdb)
```

Debugging optimized code（二）

- Using **-fno-omit-frame-pointer** might assist debugging optimized code; however, this option does not guarantee that the frame pointer will always be used.

```
$ g++ -S -O3 -o debugcode.O3.s  debugcode.c
$ g++ -S -O3 -fno-omit-frame-pointer -o debugcode.O3.frame.s  debugcode.cpp
```

Stripping and loading debugging symbols

- Many projects strip their final executables before deploying them. Stripping removes the debug symbols from the executable, but it also removes more than that.

```
$g++ -O3 -g debugcode.cpp -o debug
$strip debug -o debug.stripped
$strip  --only-keep-debug debug -o debug.sym
```

```
-rwxrwxr-x 1 iscashpc iscashpc 32840 2月  25 14:30 debug
-rw-rw-r-- 1 iscashpc iscashpc  1293 8月  29 13:44 debugcode.cpp
-rwxrwxr-x 1 iscashpc iscashpc 14320 2月  25 14:32 debug.stripped
-rwxrwxr-x 1 iscashpc iscashpc 21232 2月  25 14:33 debug.sym
```

- Loading debugging symbols

```
Reading symbols from ./debug.stripped...
(No debugging symbols found in ./debug.stripped)
(gdb) b main
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) symbol-file ./debug.sym
Reading symbols from ./debug.sym...
(gdb) b main
Breakpoint 1 at 0x401050: file debugcode.cpp, line 8.
```

Debugging Multiple Threads

```
$  g++ -g -fopenmp gdbthread.cpp
```

```
Thread 2 "a.out" hit Breakpoint 2, main._omp_fn.0(void) () at gdbthread
.cpp:74
74          sum = sum + 4.0 / (1.0 + x*x);
(gdb) info threads
  Id   Target Id                                      Frame
  1    Thread 0x7ffff7fca7c0 (LWP 78720) "a.out" main._omp_fn.0(void)
    () at gdbthread.cpp:74
* 2    Thread 0x7ffff6a8b700 (LWP 78732) "a.out" main._omp_fn.0(void)
    () at gdbthread.cpp:74
(gdb) info thread 2
  Id   Target Id                                      Frame
* 2    Thread 0x7ffff6a8b700 (LWP 78732) "a.out" main._omp_fn.0(void)
    () at gdbthread.cpp:74
(gdb) f 2
#2  0x00007ffff7065ea5 in start_thread () from /lib64/libpthread.so.0
```

Advanced GDB Features

- Search for commands matching a REGEXP

```
(gdb) apropos
```

- Attaching GDB to a Running Process

```
$ps –ef |grep a.out
$gdb <executable> <pid>
or
$gdb –p <pid>
```

- Following a Process on a Fork

```
(gdb) set follow-fork-mode child    # Set gdb to follow child on fork
(gdb) set follow-fork-mode parent   # Set gdb to follow parent on fork
(gdb) show follow-fork-mode         # Display gdb's follow mode
```

- Signal Control

```
(gdb) signal SIGCONT
(gdb) signal SIGALARM
```

内存调试工具-Sanitizer(一)

Sanitizer

- Sanitizers是谷歌发起的开源工具集，包括了AddressSanitizer, MemorySanitizer, ThreadSanitizer, LeakSanitizer，Sanitizers项目本是LLVM项目的一部分，GNU也将该系列工具加入到了GCC编译器中。
- AddressSanitizer检查地址相关问题，包括释放后使用、重复释放、堆溢出、栈溢出等等问题。
- LeakSanitizer检查内存泄漏问题。
- ThreadSanitizer检查线程数据竞争和死锁问题。
- MemorySanitizer检查使用未初始化内存问题。
- 其他还包括HWSAN和UBSan。
- 内核Sanitizer包括KASAN和KMSAN，分别提供内核中动态内存错误检查和未初始化内存使用问题检查。

内存调试工具-Sanitizer(二)

检查堆溢出

```
//g++ heap_buffer_overflow.cpp  -o heap_buffer_overflow  -ggdb  -fsanitize=address
int main(int argc, char **argv) {
 int *array = new int[100];
 array[0] = 0;
 int res = array[argc + 100];
 delete [] array;
 return res;
}
```

```
[iscashpc@v100 sanitizer]$ ./heap_buffer_overflow
=================================================================
==1882==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x614
0000001d4 at pc 0x000000400850 bp 0x7ffd3a2ae730 sp 0x7ffd3a2ae728
READ of size 4 at 0x6140000001d4 thread T0
    #0 0x40084f in main /home/iscashpc/g++code/sanitizer/heap_buffer_o
verflow.cpp:6
    #1 0x7f327e062554 in __libc_start_main (/lib64/libc.so.6+0x22554)
    #2 0x4006f8  (/home/iscashpc/debugging/sanitizer/heap_buffer_overf
low+0x4006f8)

0x6140000001d4 is located 4 bytes to the right of 400-byte region [0x6
14000000040,0x6140000001d0)
```

内存调试工具-栈(stack)保护

- 编译时栈尺寸检查
  - -Wstack-usage=<byte-size>：当函数栈使用量可能超过byte-size时产生warning。栈的使用量计算使用保守方法(无VLA, no variable-length array)。
  - -fstack-usage：编译器输出每个函数栈使用量信息。
  - -Wvla：函数使用VLA时产生warning。
  - -Wvla-larger-than=<byte-size>：VLA是无界的或由可能超过byte-size的参数确定数组界限时产生warning。

- 运行时检查
  - 提供_FORTIFY_SOURCE，运行时mem*等函数提供缓冲区溢出检查。

内存调试工具-valgrind

- valgrind是一组自动检查内存管理和线程bugs的工具组件。

```
#include <stdlib.h>
void k(void) {
  int *x = (int*)malloc(8 * sizeof(int));
  x[9] = 0;
}
int main(void) {
  k();
  return 0;
}
```

```
$ g++ -g leak.cpp
$ valgrind --leak-check=full ./a.out
```

```
HEAP SUMMARY:
    in use at exit: 32 bytes in 1 blocks
  total heap usage: 2 allocs, 1 frees, 72,736 bytes allocated

32 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C29F73: malloc (vg_replace_malloc.c:309)
   by 0x400533: k() (in /home/iscashpc/debugging/valgrind/a.out)
   by 0x400551: main (in /home/iscashpc/debugging/valgrind/a.out)

LEAK SUMMARY:
   definitely lost: 32 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

内存调试工具

- 内存工具帮助高效管理内存，检查越界访问等。
  - ptmalloc
  - tcmalloc
  - jemalloc
  - dmalloc
  - Electric Fence

静态分析

- GCC Static Analyzer　-fanalyzer
- Clang Static Analyzer　--analyze
- cppcheck
- PVS-Studio
- FBInfer
- deepCode
- SonarSource

中国科学院大学
University of Chinese Academy of Sciences

03 构建工具

configure

- 在Linux系统中安装开源软件，”./configure-->make-->make install”仍然是比较常见的方法。
- ./configure配置脚本功能是对系统做很多的测试，以用来检测出你的安装平台的目标特征，比如它会检测系统是否有CC或GCC。
- ./configure脚本有大量的命令行选项，对不同的软件包来说，这些选项可能会有变化，但是许多基本的选项是不会改变的。使用”./configure --help”命令就可以看到可用的所有选项。

configure常用选项

选项	作用
--prefix=PREFIX	安装目录前缀。程序安装相应文件在PREFIX/bin PREFIX/lib等目录。
--disable-FEATURE	禁用某个FEATURE。
-enable-FEATURE[=ARGUMENT]	启用某个FEATURE。
--with-PACKAGE[=ARGUMENT]	使用某个软件包。
--without-PACKAGE	不使用某个软件包。

configure是由Autotools系列工具产生，Autotools包括aclocal autoconf automake。

make(一)

- make(GNU make)是一个项目构建工具，用于方便地编译、链接多个源代码文件，自动决定哪些源文件需要重新编译。
- make的执行规则由Makefile文件确定。Makefile主要由若干个target顺序排列组成。一个target的prerequisites只能由0个或几个target组成，所有的target根据依赖关系构成一个有向图或树结构。
- make的执行逻辑
  - make执行一个target时，会遍历以当前节点为根节点的整个子树结构。然后从叶子往上一层层地检查每一个节点，如果节点的子树结构中包含修改时间戳更大的target，则执行相应命令，否则不执行。
  - make的默认执行的target是 Makefile 中的第一个target，也可以make完成指定的target。
- 参考资料
  - Cpp学习笔记 ——4.make与Makefile - 知乎 (zhihu.com)
  - Makefile Tutorial By Example

make(二)

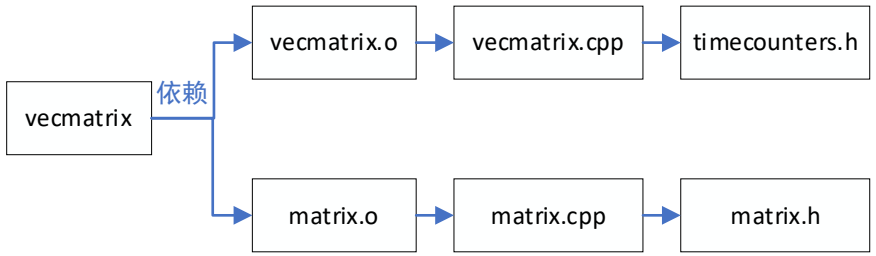
- target的语法结构

```
<target> : <prerequisites>
[tab] <commands>
```

- 简单的Makefile

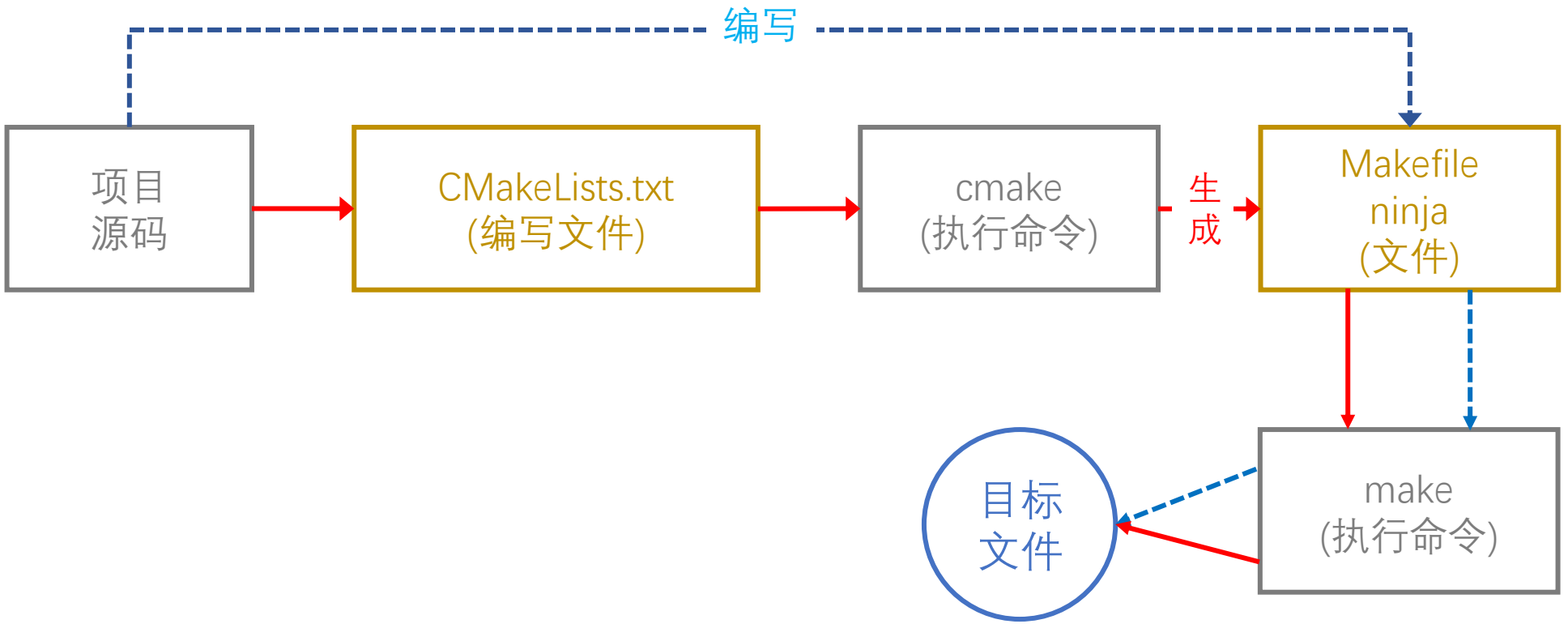
```
all:vecmatrix
vecmatrix:vecmatrix.o matrix.o
        g++ -o vecmatrix $^
vecmatrix.o:vecmatrix.cpp
./include/timecounters.h
        g++ -c vecmatrix.cpp -I./include/
matrix.o:./src/matrix.cpp ./include/matrix.h
        g++ -c ./src/matrix.cpp -I./include/
install:vecmatrix
        cp vecmatrix  ./bin/
clean:
        rm vecmatrix *.o
```

- 依赖关系



CMake简介

- CMake是一个跨平台的项目构建工具，它允许开发者编写一种与平台无关的CMakeLists.txt文件来指定整个工程的编译流程，然后再根据编译平台，自动生成本地化的Makefile和工程文件。Cmake是图灵完备的语言。



最简单的例子

CMakeLists.txt

```
project(my_project) # project name
add_executable(program program.cpp) # compile command
```

```
$ mkdir build
$ cd build/
$ cmake ../
……
$ make
[ 50%] Building CXX object CMakeFiles/program.dir/program.cpp.o
[100%] Linking CXX executable program
[100%] Built target program
```

Parameters and Message

CMakeLists.txt

```
project(my_project)
add_executable(program program.cpp)
if (VAR)
message("VAR is set, NUM is ${NUM}")
else()
message(FATAL_ERROR "VAR is not set")
endif()
```

```
$ cmake ../
CMake Error at CMakeLists.txt:8 (message):
  VAR is not set
$ cmake -DVAR=ON  -DNUM=4 ../
VAR is set, NUM is 4
```

Language Properties

```
project(my_project
LANGUAGES CXX)
cmake_minimum_required(VERSION 3.24)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
add_executable(vecadd ${PROJECT_SOURCE_DIR}/program.cpp)
# PROJECT_SOURCE_DIR is the root directory of the project
```

Target Commands

```
project(my_project)
cmake_minimum_required(VERSION 3.24)
add_executable(vecmatrix) # also add_library(vecmatrix)
target_include_directories(vecmatrix
PUBLIC include/
PRIVATE src/)
# target_include_directories(vecadd SYSTEM ...) for system headers
target_sources(vecmatrix # best way for specifying
PRIVATE src/vecadd.cpp # program sources and headers
PRIVATE src/matrix.cpp
PUBLIC include/timecounters.h
PUBLIC include/matrix.h)
target_compile_definitions(vecmatrix PRIVATE TIMING)
target_compile_options(vecmatrix PRIVATE -g)
target_link_libraries(vecmatrix PRIVATE stdc++)
target_link_options(vecmatrix PRIVATE -s)
```

Build Type

```
if (CMAKE_BUILD_TYPE STREQUAL "Debug") # "Debug" mode
# cmake already adds "-g -O0"
message("DEBUG mode")
if (CMAKE_COMPILER_IS_GNUCXX) # if compiler is gcc
target_compile_options(program "-g3")
endif()
elseif (CMAKE_BUILD_TYPE STREQUAL "Release") # "Release" mode
message("RELEASE mode") # cmake already adds "-O3 -DNDEBUG"
endif()
```

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake -DCMAKE_BUILD_TYPE=Release ..
```

Custom Targets and File Managing

```
add_custom_target(echo_target # makefile target name
COMMAND echo "Hello" # real command
COMMENT "Echo target")
# find all .cpp file in src/ directory
file(GLOB_RECURSE SRCS ${PROJECT_SOURCE_DIR}/src/*.cpp)
# compile all *.cpp file
target_sources(vecmatrix PRIVATE ${SRCS}) # prefer the explicit file list instead
```

```
cmake ..
 make echo_target
```

Local and Cached Variables

```
project(my_project)
set(VAR1 "var1") # local variable
set(VAR2 "var2" CACHE STRING "Description1") # cached variable
set(VAR3 "var3" CACHE STRING "Description2" FORCE) # cached variable
option(OPT "This is an option" ON) # boolean cached variable
# same of var2
message(STATUS "${VAR1}, ${VAR2}, ${VAR3}, ${OPT}")
```

```
$ cmake .. # var1, var2, var3, ON
$ cmake -DVAR1=a -DVAR2=b -DVAR3=c -DOPT=d .. # var1, b, var3, d
```

Manage Cached Variables

```
$ ccmake ../    #cmake gui
```

```
                                                        Page 1 of 1

CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX                /usr/local
OPT                                 ON
VAR2                                var2
VAR3                                var3




Keys: [enter] Edit an entry [d] Delete an entry
      [l] Show log output   [c] Configure
      [h] Help              [q] Quit without generating
      [t] Toggle advanced mode (currently off)
```

Find Packages

```
find_package(Boost 1.36.0 REQUIRED) # compile only if Boost library is found
if (Boost_FOUND)
target_include_directories("${PROJECT_SOURCE_DIR}/include" PUBLIC
${Boost_INCLUDE_DIRS})
else()
message(FATAL_ERROR "Boost Lib not found")
endif()
```

第一次作业

- 作业
  - 1、任选硬件和软件平台，分别使用编译器-g –O0 –O1 –O2 –O3编译homework1.cpp，记录运行时间；
  - 2、根据g++的-fopt-info（或其他编译器类似选项）输出的信息分析不同编译选项使用的优化方法。建议结合汇编代码分析。
- 提交内容
  - 1、硬件和软件环境，cpu型号、操作系统（是否虚拟机）和编译器版本等信息。
  - 2、代码运行时间。可以使用图表。
  - 3、优化方法分析。
- 评分
  - 提交内容1，满分2分；提交内容2，满分3分；提交内容3，满分5分。
  - 自己可以给出一个评分，并说明理由。

感谢！