上机作业 A4：GCC 与 Clang/LLVM 的优化比较

布置周：11 月 20 日
提交周：12 月 4 日 23：59

目标：通过本次作业，希望同学们掌握 GCC 与 Clang/LLVM 的安装方法，了解 gcc 及 Clang/LLVM 的常用编译优化选项，并掌握编译后代码的性能比较方法。

作业要求：在 Ubuntu/X64 上安装 GCC/G++ 11.0+ 和 Clang/Clang++ 14.0+两个编译器（“*版本号+*”表示大于或者等于这个版本号就行），然后用一个改造后的性能测试基准程序（CppPerformanceBenchmarks）去测量两个编译器在不同优化选项下所生成代码的代码大小和运行时间。把搜集的数据可视化呈现并做比较分析，总结分析结果。

1. 安装 GCC(建议安装版本：11.0+) 和 Clang/LLVM (建议安装版本：14.0+)

（一）在 A1 练习中，同学们已经在 Ubuntu 系统上安装好了 gcc/g++开发工具包，请确保 gcc 和 g++在可执行文件的搜索路径上，并且可以正确运行，e.g.

```
# sjr @ DESKTOP-MOBM97T in /usr/bin [14:23:44]
$ which clang clang++
/usr/bin/clang
/usr/bin/clang++
```

（二）在 A1 练习中，同学们已经在 Ubuntu 系统上安装好了 clang/clang++开发工具包，请确保 clang 和 clang++在可执行文件的搜索路径上，并且可以正确运行，e.g.

```
# sjr @ DESKTOP-MOBM97T in /usr/bin [14:26:04]
$ which gcc g++
/usr/bin/gcc
/usr/bin/g++
```

在/usr/bin 目录下面使用 file clang clang++ 和 file gcc g++会发现他们都只是符号链接。

```
# sjr @ DESKTOP-MOBM97T in /usr/bin [14:17:28]
$ file clang clang++
clang:   symbolic link to ../lib/llvm-14/bin/clang
clang++: symbolic link to ../lib/llvm-14/bin/clang++

# sjr @ DESKTOP-MOBM97T in /usr/bin [14:18:41]
$ file gcc g++
gcc: symbolic link to gcc-11
g++: symbolic link to g++-11
```

Write-up 1：找出 gcc/g++和 clang/clang++的对应可执行文件分别在哪并且解释有何不同？为什么要使用符号链接？

（请找到各自对应的可执行文件并截图，回答上述问题）

2. 安装性能测试集

我们用的测试集从 https://gitlab.com/chriscox/CppPerformanceBenchmarks 上下载，并经过一定的修改。为了缩短测试时间，对一些测试例我们特意减少了测试的迭代次数。

直接使用 git clone 拉取仓库或者下载 CppPerformanceBenchmarks.tar 并解包到自己的测试目录下即可。查看 makefile，可以发现里面包含了许多不同的测试。

3. 编译器及优化选项的组合测试

- 选取某一个具体的测试(e.g. lookup_table, loop_fusion etc.)，修改 makefile，使得 makefile **仅编译及运行你所关注的测试例** （**可以把处理其余测试例所对应的行删掉，也可以添加当前 makefile 内没有包含的测试例并相应修改 makefile**），提示：
  主要修改 makefile 中的
  ```
  BINARIES = machine \
             ....
  ```
  及
  ```
  report:   $(BINARIES)
            echo "##STARTING Version 1.0" > $(REPORT_FILE)
            date >> $(REPORT_FILE)
            echo "##Compiler: $(CC) $(CXX)" >> $(REPORT_FILE)
            echo "##CFlags: $(CFLAGS)" >> $(REPORT_FILE)
            echo "##CPPFlags: $(CPPFLAGS)" >> $(REPORT_FILE)
            echo "System Information collected by program: " >> $(REPORT_FILE)
            ./machine >> $(REPORT_FILE)
            ...
  ```
  两部分的相关行。
- 如果你选取的测试例的运行时间太短(导致不同编译优化选项间的数据无法进行合理比较)，可以修改此测试对应 cpp 文件中 iterations 变量的值，一旦确定了 iterations 变量的值，在后续的测试中不能再修改此值。相应地，如果你选取的测试例的运行时间太长，则可以减少此测试对应 cpp 文件中 iterations 变量的值。

```
// this constant may need to be adjusted to give reasonable minimum times
// For best results, times should be about 1.0 seconds for the minimum test run
// on 3Ghz desktop CPUs, 4000k iterations is about 1.0 seconds
int iterations = 4000;
```

- 对于一个特定的测试例，按照下面的测试组合进行测试并进行数据收集，由于每一次测试的结果都存在 report.txt 文件内，需要在每次测试运行完把 report.txt 进行重命名以保留此次测试的数据，比如我们可以用以下命令保留用 gcc 在-O0 编译优化选项下收集的测试数据：
  ```
  $mv report.txt report.txt.gcc-O0
  ```
- 编译器用 CC 及 CXX 两个宏来控制，编译优化选项用 OPTLEVEL 宏来控制。CC 和 CXX 需要保持一致，比如说 CC 选 clang，则 CXX 必须为 clang++。运行一个具体测试的命令如下（以 clang 编译器，-O2 编译优化选项为例）：

$make clean
$make report CC=clang CXX=clang++ OPTLEVEL=-O3
$mv report.txt report.txt.clang-O2

同时记下此组合下编译出来测试例的代码大小(文件大小)。比如选择了 loop_interchange 测试例，则记下在“clang”+“-O1”的组合下 loop_interchange 测试例的代码大小为 179648 字节（注意：即使用同一编译器的不同版本编译同一个测试例，代码大小也可能会有差别）。

```
$ ls -la | grep stepanov_abstraction
-rwxr-xr-x  1 sjr sjr 179648 Nov  2 15:44 stepanov_abstraction
```

（每次测试对 report.txt 修改名字，当然这种方式有点呆，那么怎么自动化实现该功能呢）

测试组合：

编译器：{ gcc/g++，clang/clang++}
编译器优化选项：{-O0, -O1, -O2}
共有 2*3=6 种组合

Write-up 2：请说明一下你选择了哪个测试例，以及你选择这个测试例的原因。

Write-up 3：你对 makefile 及所选测试例对应的源文件做了哪些修改，为什么？

（请解释修改的原因，谈谈 makefile 中各部分作用）

Write-up 4：自动化脚本实现对测试组合进行测试并获得结果

（通过修改 makefile 以及编写 shell 脚本来实现，在报告中写明脚本思路）

4. 测试数据的分析

Write-up 5：请针对选定的测试例及收集到的 6 个测试数据结果文件（以及相应测试例编译后的代码大小），进行数据分析，总结分析洞见。要求：

- 需要对不同组合下测试例的代码大小及运行性能进行分析
- 尽量用可视化的形式来呈现分析结果
- 分析洞见的总结尽量简明扼要，如有可能请加上对所总结洞见的合理解释