GCC vs. Clang/LLVM: An In-Depth Comparison of C/C++ Compilers

Introducing GNU Compiler Collection (GCC) and Clang/Low Level Virtual Machine (LLVM); comparing the performance of both C/C++ compilers

Alibaba Tech Follow
Aug 29, 2019 · 17 min read

Background

Visual C++, GNU Compiler Collection (GCC), and Clang/Low Level Virtual Machine (LLVM) are three mainstream C/C++ compilers in the industry. Visual C++ provides graphical user interfaces (GUIs) and is easy to debug, but it is not suitable for Linux platforms. Therefore, this document mainly compares GCC with Clang/LLVM.

GCC is a program language compiler developed by GNU. It is a set of free software released under the GNU General Public License (GPL) and GNU Lesser General Public License (LGPL). It is an official compiler for the GNU and Linux systems, and a main compiler for compiling and creating other UNIX operating systems.

LLVM contains a series of modularized compiler components and tool chains. It can optimize program languages and links during compilation, runtime, and idle time and generate code. LLVM can serve as a background for compilers in multiple languages. Clang is a C, C++, Objective-C, or Objective-C++ compiler that is compiled in C++ based on LLVM and released under the Apache 2.0 license. Clang is mainly used to provide performance superior to that of GCC.

Through long-term development and iteration, GCC, Clang, and LLVM have become mature compilers in the industry. Then, which compiler is better? Which one should we use to compile and build programs and systems?

Significance of a Good Compiler

Modern processors all have superscalar and long pipelines, and complex internal structures, and they support vector extension units in the Complex Instruction Set Computer (CISC) or Reduced Instruction Set Computer (RISC) architecture. For many programs that contain general computing-intensive kernels, programmers can use vector extension commands to greatly improve program execution performance. For example, in matrix and vector operations, the combined multiplication and addition commands are used to improve performance and accuracy. Bit mask commands are used for branch processing in vector operations. However, to achieve the highest performance, programmers and compilers still need to expend a lot of effort to handle tasks with complex memory access modes and non-standard kernels.

Additionally, standards of modern advanced languages constantly abstract the details of underlying hardware and data structures to generate general code that is more logical and mathematical, instead of specific operation instructions and memory access paths. C++ standards are increasingly expressive and abstract. Python is popular because it is more readable and expressive, even at the cost of a lower running speed. Higher expressiveness increases the burden of the compiler to generate good assembly code from the complex structures compiled by programmers. The compiler must be smarter and work harder to maximize performance by using the code. Not all compilers can do this. When selecting a

compiler, we must first consider whether the same code segment can generate more efficient assembly commands.

In addition to generating high-performance executable programs, modern compilers must also have high performance themselves. A large-sized software project in C++ may contain hundreds to thousands of individual translation units. Each translation unit may contain thousands of lines of code. C++ code can also use a large number of template-based programming technologies. These technologies require the compiler to transfer relevant information multiple times to generate a target file. The compilation of large-sized C++ projects may take several hours, and multiple mutually dependent changes must be submitted concurrently during development. Each submission requires developers to recompile most of the code libraries. Therefore, faster compilers (build tools) are critical for achieving high productivity for large teams.

In terms of language extension, modern computing systems with multiple kernels, vector processing capabilities, and accelerators provide capabilities superior to the natural capabilities of common programming languages. Therefore, specific high-performance computing (HPC) frameworks, such as OpenMP and OpenACC, can fill this gap. These frameworks provide application program interfaces (APIs) that programmers can use to express parallelism in code. The compiler and the corresponding runtime library must map the parallel code to the processor architecture. Many HPC projects depend on OpenMP and OpenACC standards, which are being extended by developers and hardware manufacturers. Therefore, the compilers must keep up with the development of language extension standards.

In conclusion, a good compiler allows us to focus on the process of programming, rather than fighting against its shortcomings. It can support the latest language standards, generate optimized commands from the most abstract code, and compile the source code in less time.

GCC Development History

Before learning GCC, you need to first understand the GNU Project. Richard Stallman launched the GNU Project in 1984 to build a UNIX-like open source software system. The GNU operating system has not evolved extensively over time. However, it has incubated many excellent and useful open source software tools, such as Make, Sed, Emacs, Glibc, GDB, and GCC as well. These GNU open source software and Linux kernels together constitute the GNU/Linux system. In the

beginning, GCC provided stable and reliable compilers, based on the C programming language, for the GNU system. Its full name is GNU C Compiler. Later, more languages (such as Fortran, Obj-C, and Ada) were supported, and the full name of GCC changed to GNU Compiler Collection.

GCC-1.0 was released by Richard Stallman in 1987, more than thirty years ago. From the software perspective it is very old. Someone collected the GCC development records from between 1989 and 2012, and produced a thirty-minute animated video (GNU Compiler Collection dev history 1989–2012), intuitively demonstrating the development process of GCC. We can learn about the development history of GCC from its versions:

- GCC-1.0: released by Richard Stallman in 1987.
- GCC-2.0: released in 1992 and supported C++. Later, the GCC community was split because Richard Stallman defined GCC as a reliable C compiler of the GNU system and thought that GCC at that time was sufficient for the GNU system and the development focus should be shifted from GCC to the GNU system itself. Other major developers hoped to continue improving GCC and make more radical developments and improvements in various aspects. These active developers left the GCC community in 1997 and developed the EGCS fork.
- GCC-3.0: Obviously, developers generally had a strong desire for good compilers. The EGCS fork developed smoothly and became recognized by more and more developers. Eventually, EGCS was used as the new GCC backbone and GCC-3.0 was released in 2001. The split community was re-merged again, but Richard Stallman's influence had been weakened to a certain extent. Additionally, the GCC Industrial Committee had begun to decide the development direction of GCC.
- GCC-4.0: released in 2005. This version was integrated into Tree Serial Storage Architecture (SSA), and GCC evolved to be a modern compiler.
- GCC-5.0: released in 2015. Later, the GCC version policy was adjusted and a major version was released each year. An unexpected benefit is that the version number corresponds with the year. For example, GCC-7 was released in 2017, and GCC-9 was released in 2019.

Now, GCC development has entered the "modern chronicle". Facing the competitive pressure of LLVM, the GCC community has actively made many adjustments, such

as accelerating compilation and improving the compilation warning information. Over the past 30 years, GCC has evolved from a challenger in the compiler industry to a mainstream compiler for Linux systems, and now faces the challenge of LLVM. Fortunately, the GCC community is making adjustments to accelerate the development of GCC. We can expect that the competition between the two compilation technologies will continue to provide software developers with better compilers.

Development History of Clang and LLVM

LLVM

LLVM was originated from the research by Chris Lattner on UUIC in 2000. Chris Lattner wanted to create a dynamic compilation technology for all static and dynamic languages. LLVM is a type of open source software developed under the BSD License. The initial version 1.0 was released in 2003. In 2005, Apple Inc. hired Chris Lattner and his team to develop programming languages and compilers for Apple computers, after which the development of LLVM entered the fast lane. Starting from LLVM 2.5, two minor LLVM versions were released every year (generally in March and September). In November 2011, LLVM 3.0 was released to become the default XCode compiler. XCode 5 started to use Clang and LLVM 5.0 by default. The version policy was adjusted for LLVM 5.0 and later versions, and two major versions are released every year. The current stable version is 8.0.

The name of LLVM was first abbreviated from Low Level Virtual Machine. As this project is not limited to the creation of a virtual machine, the abbreviation LLVM is often questioned. After LLVM was developed, it became a collective term for many compilation tools and low-level tool technologies, making the name less appropriate. Developers decided to abandon the meaning behind this abbreviation. Now LLVM has become the official brand name, applicable to all projects under LLVM, including LLVM Intermediate Representation (LLVM IR), LLVM debugging tools, and LLVM C++ standard libraries. LLVM can be used as a traditional compiler, JIT compiler, assembler, debugger, static analysis tool, and for other functions related to programming languages.

In 2012, LLVM won the software system award of Association for Computing Machinery (ACM), along with traditional systems such as UNIX, WWW, TCP/IP, TeX, and Java. LLVM greatly simplifies the implementation of new programming

language tool chains. In recent years, many new programming languages such as Swift, Rust, and Julia have used LLVM as their compilation framework. Additionally, LLVM has become the default compiler for Mac OS X, iOS, FreeBSD and Android systems.

Clang

Clang is designed to provide a frontend compiler that can replace GCC. Apple Inc. (including NeXT later) has been using GCC as the official compiler. GCC has always performed well as a standard compiler in the open source community. However, Apple Inc. has its own requirements for compilation tools. On the one hand, Apple Inc. added many new features for the Objective-C language (or even, later, the C language). However, GCC developers did not accept these features and assigned low priority to support for these features. Later, they were simply divided into two branches for separate development, and consequently the GCC version released by Apple Inc. is far earlier than the official version. On the other hand, the GCC code is highly coupled and hard to be developed separately. Additionally, in later versions, the code quality continues to decrease. However, many functions required by Apple Inc. (such as improved Integrated Development Environment (IDE) support) must call GCC as a module, but GCC never provides such support. Moreover, the GCC Runtime Library Exemption fundamentally limits the development of LLVM GCC. Also limited by the license, Apple Inc. cannot use LLVM to further improve the code generation quality based on GCC. Therefore, Apple Inc. decided to write the frontend Clang of C, C++, and Objective-C languages from scratch to completely replace GCC.

As the name suggests, Clang only supports C, C++, and Objective-C. Development started in 2007 and the C compiler was first completed. Clang for Objective-C cloud be fully used for the production environment in 2009. Support for C++ also progressed quickly. Clang 3.3 fully supported C++ 11, Clang 3.4 fully supported C++ 14, and Clang 5 fully supported C++ 17, and all were significantly ahead of GCC at that time.

Clang/LLVM and GCC Community

GCC Community

Like other open source software communities, the GCC community is dominated by free software enthusiasts and hackers. In the process of development, the GCC

community management and participation mechanisms are gradually formed today. Currently, the GCC community is a relatively stable and well-defined acquaintance society in which each person has clear roles and duties:

- Richard Stallman and Free Software Foundation (FSF): Although seldom involved in GCC community management, Richard Stallman and FSF are still detached in license and legal affairs.
- GCC Industrial Committee: It manages the GCC community affairs, technology-independent GCC development topics, and the appointment and announcement of reviewers and maintainers. It currently has 13 members.
- Global maintainers: They dominate GCC development activities. To some extent, they determine the development trend of GCC. Currently, there are 13 global maintainers, who do not all hold office in GCC Industrial Committee.
- Frontend, middle-end, and backend maintainers: They are the maintainers of frontend, backend, and other modules. They are responsible for the code of the corresponding GCC module, and many of them are the main contributors to the module code. It is worth noting that reviewers are generally classified into this group. The difference is that reviewers cannot approve their own patch, while maintainers can submit their own modifications within their scope of responsibility without approval from reviewers.
- Contributors: They are the most extensive developer groups in the GCC community. After signing the copyright agreement, any developers can apply for the Write after Approval permission from the community, and then submit the code by themselves.

Like other open source communities, the mature GCC community is no longer dominated by hackers. Commercial companies began to play important roles in the community, such as recruiting developers and sponsoring development meetings. Currently, the GCC community is dominated by the following types of commercial companies:

- System vendors, mainly including RedHat and SUSE.
- Chip vendors, mainly including Intel, ARM, AMD, and IBM (PowerPC).
- Specialized vendors, such as CodeSourcery and tool chain service providers like AdaCore based on the Ada language. CodeSourcery had a brilliant history and

recruited many famous developers, but declined after it was acquired by Mentor.

In the current GCC community, chip vendors dominate backend development, while system vendors guide other development areas. In terms of community development, the GCC code is currently hosted on its own SVN server. A Git API is provided to facilitate development and submission. Patch review is similar to that in the Linux Kernel community and uses the Mailing List form. As mentioned above, the GCC community is a relatively stable (or closed) acquaintance society. The community basically has 150 to 200 active contributors each year and holds a developer conference in September every year. In September 2019, the developer conference will be held in Montreal, Canada.

LLVM Community

The LLVM community is a noob-friendly compiler community. It quickly responds to the questions of new users and patch reviews. This is also the basis and source for subsequent LLVM Foundation discussions and the adoption of the LLVM Community Code of Conduct, and causes a series of politically correct discussions.

All LLVM projects and problems are discussed through the DevExpress email list, and code submission is notified through the commits email list. All bugs and feature modifications are tracked through the bugs list. The submitted patches are recommended for the master branches. The style complies with LLVM Coding Standards and code review is performed through Phabricator. Currently, the LLVM code repository has been migrated to GitHub.

Unlike the GCC community, the LLVM community only has the LLVM Foundation. The LLVM Foundation has eight members. In addition to managing LLVM community affairs, each member of the LLVM Foundation has to guide LLVM development issues related to technology. Currently, the president is Tanya Lattner, the wife of Chris Lattner. Chris Lattner himself is also a member of the foundation and has strong control over the LLVM community and the development direction of LLVM.

The code review policy in the LLVM community is basically the same as that in the GCC community. The difference is that, due to the fast development of LLVM, many contributors do not have commit access permission, and have to submit their code through the maintainers. Currently, the Clang and LLVM communities have more than 1,000 contributors each year. Generally, developer conferences are held in

April and October yearly. The developer conference in October 2019, will be held in San Jose, USA.

The LLVM license is changed from UIUC License to Apache 2.0 License with LLVM exceptions. It is mainly used to solve the problem that the LLVM runtime library is based on MIT License and the patent authorization required for the project is too extensive. Under this license, LLVM allows anyone to derive commercial products from LLVM without any restrictions, and does not require that any derivatives provide open source code, thus promoting the extensive use of LLVM, including:

1. The downloading or use of LLVM in whole or in part for personal, internal, or commercial purposes. The ability to modify LLVM code without contributing it back to the project.
2. The creation of a package or release version containing LLVM. The association of LLVM with code authorized by all other major open source licenses (including BSD, MIT, GPLv2, and GPLv3).
3. When distributing LLVM again, you must retain the copyright notice. You cannot delete or replace the copyright header. The binary file containing LLVM must contain the copyright notice.

Performance Comparison between GCC and LLVM

Test Server

Architecture: x86_64
Processor: Intel (R) Xeon (R) Platinum 8163 CPU @ 2.50 GHz
L1 data cache: 32 KB
L2 cache: 1,024 KB
L3 cache: 33,792 KB
Memory: 800 GB
Operating system: Alibaba Group Enterprise Linux Server release 7.2 (Paladin)
Kernel: 4.9.151–015.ali3000.alios7.x86_64
Compiler: Clang/LLVM 8.0 GCC8.3.1

Benchmark

SPEC CPU 2017 is a set of CPU subsystem test tools for testing the CPU, cache, memory, and compiler. It contains 43 tests of four categories, including SPECspeed

2017 INT and FP that test the integer speed and floating point operation speed and SPECrate 2017 INT and FP that test the integer concurrency rate and floating point concurrency rate. Clang does not support the Fortran language. Therefore, in this example, the C/C ++ programs in the SPEC Speed test set are used to test the single-core performance difference between the binary programs generated by Clang and GCC. The following table lists the SPEC CPU2017 C and C++ sets:

CINT2017 SpeedCFP2017
Speed600.perlbench_s619.lbm_s602.gcc_s644.nab_s605.mcf_s620.omnetpp_s623.xalancbmk_s625.x264_s631.deepsjeng_s641.leela_s657.xz_s

Test Methods

The LLVM-lnt automation framework is used to perform the test and compare the performance. It runs in the same way as runcpu of SPEC CPU. Before LLVM-lnt runs, cache (echo 3 > /proc/sys/vm/drop_caches) is cleared and then the test dataset runs. Next, the ref dataset runs three times. The average value of the three ref test run results is used as the final result. To reduce performance fluctuations caused by CPU migration or context switch, processes running on the test dataset and ref dataset are bound to a CPU core by using the CPU affinity tool. For the compile time test, this method uses thread 1 to build the test program and compare the test items that have been compiled for a long time. The compile time does not include the linker execution time. It only includes the time when all source files in all test programs are generated.

Compilation Performance Comparison

The GCC compilation process is as follows: read the source file, preprocess the source file, convert it into an IR, optimize and generate an assembly file. Then the assembler generates an object file. Clang and LLVM do not rely on independent compilers, but integrate self-implemented compilers at the backend. The process of generating assembly files is omitted in the process of generating object files. The object file is generated directly from the IR. Besides, compared with the GCC IR, the data structure of LLVM IR is more concise. It occupies less memory during compilation and supports faster traversal. Therefore, Clang and LLVM are advantageous in terms of the compilation time, which is proven by the data obtained from SPEC compilation, as shown in the figure below. Clang reduces the single-thread compilation time by 5% to 10% compared with GCC. Therefore, Clang offers more advantages for the construction of large projects.

Comparison of SPEC compilation time

Comparison of Execution Performance

Most cloud workloads require that the applications can run in different clusters. When creating these applications, do not specify machine-related parameters. To adapt to the fast iteration caused by demand changes, off-premises workloads must also be debuggable. Therefore, apart from some stable and common libraries that enable high compilation optimization levels, the workload itself has a low compilation and optimization level (O2 or below). To meet this requirement, this document compares the performance of different compilers at the O2 and O3 optimization levels for INT Speed programs, as shown in the following figure:

Performance comparison of the SPEC CPU2017 INT Speed

GCC has a 1% to 4% performance advantage over Clang and LLVM for most programs at the O2 and O3 levels, and on average has an approximately 3% performance advantage for SPEC CPU2017 INT Speed. In terms of 600.perlbench_s and 602.gcc_s/O2, GCC has a great performance advantage (more than 10%). These two test items have no outstanding hotspots and can reflect the comprehensive optimization effect of the compiler. The test results show that GCC is always advantageous in performance optimization. However, for two AI-related programs, including 631.deepsjeng_s and 641.leela_s, which are newly added to the SPEC test, Clang and LLVM improve the performance by more than 3% in comparison to GCC. This also reflects the rapid progress of LLVM in terms of optimization. For the 625. x264_s O2 optimization, LLVM improves the performance by 40% because the hotspot of the case complies with the vectorized rules. But Clang and LLVM optimize the vectors at the O2 level, while GCC optimizes the vectors at the O3 level. Except for vectorized programs, GCC does not greatly improve the performance at the O3 level compared with that at the O2 level. In other words, the programs are not sensitive to GCC O3 optimization. In contrast, Clang and LLVM significantly improve the performance of some programs (such as 600. perlbench_s and 602. gcc_s) at the O3 level.

HPC programs, such as FP Speed, generally run on high-end servers. They have stable core algorithms, high requirements for performance-related vectorization and parallelism, and enable high levels of optimization (O3 or above). Therefore, this document compares the performance at the O3 + march = native (skylake-avx512) optimization level, as shown below:

Performance comparison of SPEC CPU2017 FP Speed

For the two FP programs, GCC also can improve the performance by about 3%. Clang and LLVM are conservative in loop optimization and thus not advantageous in performance. However, the Polly sub-project of Clang and LLVM provides a high-level loop and data locality optimizer that has been widely applied in machine learning, high performance computing, and heterogeneous computing optimization. I believe that Polly can greatly improve the performance of programs that contain hotspot loops complying with vectorization and parallelism rules. I will also analyze the performance of Polly in a series of benchmarks and workloads.

Concluding Remarks

From the benchmarking tests above, we can see that Clang offers more advantages for the construction of large projects while GCC is always advantageous in performance optimization. The bla depends on your specific application

In addition to performance comparison, I would like to share the advantages and disadvantages of GCC and Clang and LLVM:

Advantages of GCC

- GCC supports more traditional languages than Clang and LLVM, such as Ada, Fortran, and Go.
- GCC supports more less-popular architectures, and supported RISC-V earlier than Clang and LLVM.
- GCC supports more language extensions and more assembly language features than Clang and LLVM. GCC is still the only option for compiling the Linux kernel. Although research on kernel compilation by using Clang and LLVM is also reported in the industry, the kernel cannot be compiled without modifying the source code and compilation parameters.

Advantages of Clang and LLVM

- Emerging languages are using the LLVM frameworks, such as Swift, Rust, Julia, and Ruby.
- Clang and LLVM comply with the C and C ++ standards more strictly than GCC. GNU Inline and other problems during GCC upgrade do not occur.
- Clang also supports some extensions, such as attributes for thread security check.
- Clang provides additional useful tools, such as scan-build and clang static analyzer for static analysis, clang-format and clang-tidy for syntax analysis, as well as the editor plug-in Clangd.
- Clang provides more accurate and friendly diagnostic information, and highlights error messages, error lines, error line prompts, and repair suggestions. Clang regards the diagnostic information as a feature. The diagnostic information began to be improved only from GCC 5.0, and became mature in GCC 8.

(Original article by Ma Jun 马骏)

. . .

Alibaba Tech

First hand and in-depth information about Alibaba's latest technology → Facebook: **"Alibaba Tech"**. Twitter: **"AlibabaTech"**.

Programming C App Development Coding Linux

444 claps

WRITTEN BY

Alibaba Tech

First-hand & in-depth information about Alibaba's tech innovation in Artificial Intelligence, Big Data & Computer Engineering. Follow us on Facebook!

Follow

See responses (3)

More From Medium

More from Alibaba Tech

BindingX: Going Native, Without Going Native

Alibaba Tech in Noteworthy - The...
Jun 7, 2018 · 9 min read

289

Related reads

Building a Website with C++

Robert W. Oliver II in Sourcerer Blog
Jan 20, 2018 · 7 min read

4.96K

Also tagged App Development

5 DevOps Monitoring Strategies for Your Application

Steve Kosten in Towards Data Science
May 2 · 7 min read ★

4