G++ optimization beyond -O3/-Ofast

Asked 11 years, 11 months ago Modified 3 years, 6 months ago

Viewed 67k times

100

The Problem

We have a mid-sized program for a simulation task, that we need to optimize. We have already done our best optimizing the source to the limit of our programming skills, including profiling with Gprof and Valgrind.

When finally finished, we want to run the program on several systems probably for some months. Therefore we are really interested in pushing the optimization to the limits.

All systems will run Debian/Linux on relatively new hardware (Intel i5 or i7).

The Question

What are possible optimization options using a recent version of g++, that go beyond -O3/-Ofast?

We are also interested in costly minor optimization, that will payout in the long run.

What we use right now

Right now we use the following g++ optimization options:

- `-Ofast`: Highest "standard" optimization level. The included `-ffast-math` did not cause any problems in our calculations, so we decided to go for it, despite of the non standard-compliance.
- `-march=native`: Enabling the use of all CPU specific instructions.
- `-flto` to allow link time optimization, across different compilation units.

c++ **g++** **compiler-optimization**

Share

Improve this question

Follow

edited Aug 14, 2016 at 23:21

asked Jan 24, 2013 at 1:27

Haatschii

9,269 ● 13 ● 61 ● 98

7 Have you tried profile-driven optimisation - although that will depend on having "representative" data for the profiling. Beyond that, I think it's identifying hot-spots and looking at what code the processor generates and see if you can organise the data/code better or come up with a different algorithm. – Mats Petersson Jan 24, 2013 at 1:36

9 Note that starting your program one day later and getting 1% performance increase form spending that day optimizing will only break even after a runtime of 100 days. In other words, starting your program run a few days earlier will likely outweigh small optimizations. – sth Jan 24, 2013 at 1:40

3 @sth: This is of curse true. However I hope to find some hints/tricks that could also be reused in later projects, so I don't have to spend the day I win from the optimization... – Haatschii Jan 24, 2013 at 1:48

2 @OliCharlesworth: Your probably right, so I took that explicit example out. Howerver, I hope there might be flags/tricks that yield maybe even more than minor speedups. – Haatschii Jan 24, 2013 at 2:04

1 I didn't check myself, but `-ffast-math` not always makes the code faster according to this blog. – tr3w Jan 24, 2013 at 9:50

8 Answers

Sorted by: Highest score (default)

120

Most of the answers suggest alternative solutions, such as different compilers or external libraries, which would most likely bring a lot of rewriting or integration work. I will try to stick to what the question is asking, and focus on what can be done with GCC alone, by activating compiler flags or doing minimal changes to the code, as requested by the OP. This is not a "you must do this" answer, but more a collection of GCC tweaks that have worked out well for me and that you can give a try if they are relevant in your specific context.

Warnings regarding original question

Before going into the details, a few warning regarding the question, typically for people who will come along, read

the question and say "the OP is optimising beyond O3, I should use the same flags than he does!".

- `-march=native` enables usage of **instructions specific to a given CPU architecture**, and that are not necessarily available on a different architecture. The program may not work at all if run on a system with a different CPU, or be significantly slower (as this also enables `mtune=native`), so be aware of this if you decide to use it. More information here.
- `-Ofast`, as you stated, enables some **non standard compliant** optimisations, so it should used with caution as well. More information here.

Other GCC flags to try out

The details for the different flags are listed here.

- `-Ofast` enables `-ffast-math`, which in turn enables `-fno-math-errno`, `-funsafe-math-optimizations`, `-ffinite-math-only`, `-fno-rounding-math`, `-fno-signaling-nans` and `-fcx-limited-range`. You can go even further on **floating point calculation optimisations** by selectively adding some **extra flags** such as `-fno-signed-zeros`, `-fno-trapping-math` and others. These are not included in `-Ofast` and can give some additional performance increases on calculations, but you must check whether they actually benefit you and don't break any calculations.
- GCC also features a large amount of **other optimisation flags** which aren't enabled by any "-O"

options. They are listed as "experimental options that may produce broken code", so again, they should be used with caution, and their effects checked both by testing for correctness and benchmarking. Nevertheless, I do often use `-frename-registers`, this option has never produced unwanted results for me and tends to give a noticeable performance increase (ie. can be measured when benchmarking). This is the type of flag that is very dependant on your processor though. `-funroll-loops` also sometimes gives good results (and also implies `-frename-registers`), but it is dependent on your actual code.

PGO

GCC has **Profile-Guided Optimisations** features. There isn't a lot of precise GCC documentation about it, but nevertheless getting it to run is quite straightforward.

- first compile your program with `-fprofile-generate`.
- let the program run (the execution time will be significantly slower as the code is also generating profile information into .gcda files).
- recompile the program with `-fprofile-use`. If your application is multi-threaded also add the `-fprofile-correction` flag.

PGO with GCC can give amazing results and really significantly boost performance (I've seen a 15-20% speed increase on one of the projects I was recently working on). Obviously the issue here is to have some

data that is sufficiently representative of your application's execution, which is not always available or easy to obtain.

GCC's Parallel Mode

GCC features a **Parallel Mode**, which was first released around the time where the GCC 4.2 compiler was out.

Basically, it provides you with **parallel implementations of many of the algorithms in the C++ Standard Library**. To enable them globally, you just have to add the `-fopenmp` and the `-D_GLIBCXX_PARALLEL` flags to the compiler. You can also selectively enable each algorithm when needed, but this will require some minor code changes.

All the information about this parallel mode can be found here.

If you frequently use these algorithms on large data structures, and have many hardware thread contexts available, these parallel implementations can give a huge performance boost. I have only made use of the parallel implementation of `sort` so far, but to give a rough idea I managed to reduce the time for sorting from 14 to 4 seconds in one of my applications (testing environment: vector of 100 millions objects with custom comparator function and 8 cores machine).

Extra tricks

Unlike the previous points sections, this part does **require some small changes in the code**. They are also GCC specific (some of them work on Clang as well), so compile time macros should be used to keep the code portable on other compilers. This section contains some more advanced techniques, and should not be used if you don't have some assembly level understanding of what's going on. Also note that processors and compilers are pretty smart nowadays, so it may be tricky to get any noticeable benefit from the functions described here.

- GCC builtins, which are listed here. Constructs such as `__builtin_expect` can help the compiler do better optimisations by providing it with **branch prediction** information. Other constructs such as `__builtin_prefetch` brings data into a cache before it is accessed and can help reducing **cache misses**.
- function attributes, which are listed here. In particular, you should look into the `hot` and `cold` attributes; the former will indicate to the compiler that the function is a **hotspot** of the program and optimise the function more aggressively and place it in a special subsection of the text section, for better locality; the later will optimise the function for size and place it in another special subsection of the text section.

I hope this answer will prove useful for some developers, and I will be glad to consider any edits or suggestions.

Share Improve this answer

Follow

edited May 30, 2021 at 4:49

Gabriel Staples

51.5k ● 30 ● 273 ● 354

answered Jul 21, 2016 at 18:33

Pyves

6,463 ● 8 ● 43 ● 60

5 Thanks, this answer describes pretty much what we ended up doing, especially PGO proved to be quite useful. Additionally I also liked the ACOVEA project suggested by @zaufi, although it didn't work out for this project. – Haatschii Aug 14, 2016 at 23:13

4 Wow, didn't know about the PGO options! About 30% improvement in my case. – fhucho Dec 31, 2016 at 22:19

2 "These are not included in -Ofast" I am pretty sure that is wrong. If you look at the GCC docs for -ffast-math (turned on by -Ofast), it also turns on -funsafe-math-optimizations, which turns on -fassociative-math. (among others) There is a phrase in the docs "This option is not turned on by any -O option", that I consider a documentation error, since -Ofast does turn them on. Also, PGO turn on -funroll-loops, which turns on -frename-registers. – uLoop Jan 2, 2018 at 22:57

@uLoop: the GCC documentation is indeed not always clear. I have checked those flags using the compiler's `-Q` flag and have tweaked the answer accordingly. – Pyves Jan 8, 2018 at 22:21

1 @Pyves I also came across another method to compliment along with yours: Feedback directed optimization with GCC and Perf: blog.wnohang.net/index.php/2015/04/29/… However, having snags with this one, the article isn't up-to-date, some commands are depracated and gcov_create is having problems with reading perf's perf.data file. Maybe you

can investigate and provide some guidance. – Demon Jul 7, 2020 at 3:26

18

> relatively new hardware (Intel i5 or i7)

Why not invest in a copy of the Intel compiler and high performance libraries? It can outperform GCC on optimizations by a significant margin, typically from 10% to 30% or even more, and even more so for heavy number-crunching programs. And Intel also provide a number of extensions and libraries for high-performance number-crunching (parallel) applications, if that's something you can afford to integrate into your code. It might payoff big if it ends up saving you months of running time.

> We have already done our best optimizing the source to the limit of our programming skills

In my experience, the kind of micro- and nano-optimizations that you typically do with the help of a profiler tend to have a poor return on time-investments compared to macro-optimizations (streamlining the structure of the code) and, most importantly and often overlooked, memory access optimizations (e.g., locality of reference, in-order traversal, minimizing indirection, wielding out cache-misses, etc.). The latter usually involves designing the memory structures to better reflect

the way the memory is used (traversed). Sometimes it can be as simple as switching a container type and getting a huge performance boost from that. Often, with profilers, you get lost in the details of the instruction-by-instruction optimizations, and memory layout issues don't show up and are usually missed when forgetting to look at the bigger picture. It's a much better way to invest your time, and the payoffs can be huge (e.g., many O(logN) algorithms end up performing almost as slow as O(N) just because of poor memory layouts (e.g., using a linked-list or linked-tree is a typical culprit of huge performance problems compared to a contiguous storage strategy)).

Share Improve this answer

Follow

answered Jan 24, 2013 at 2:16

Mikael Persson

18.5k ● 6 ● 40 ● 53

The reasons we don't (yet) use the intel compiler is that it does not support certain C++11 features we are using. If this changes soon enough we will try the ICC as well. I mostly agree with you second part. But apart from letting further people have a look at the code, I don't see how we can further improve it. Therefore my question was if there are more things we can make the compiler do. – Haatschii Jan 24, 2013 at 2:30

2 @Haatschii Yeah, I'm sorry I can't directly answer your question (i.e., how to squeeze the most out of GCC), cause I don't think you can. I just thought it would be worth putting those few points out there (using ICC and doing memory optimizations) as better avenues to actually achieve your goal. – Mikael Persson Jan 24, 2013 at 2:44

2 I'm very skeptical of the claim 'typically from 10% to 30% or even more'. At the very least, these margins are well outside

what I've measured in my own work. I'd love to see a published collection of benchmarks demonstrating that, provided equivalent compiler flags are used and the flags used are published, if only to see if I've missed optimization opportunities on the intel compilers. – apmccartney Mar 26, 2017 at 19:19

8

If you can afford it, try VTune. It provides MUCH more info than simple sampling (provided by gprof, as far as I know). You might give the Code Analyst a try. Latter is a decent, free software but it might not work correctly (or at all) with Intel CPUs.

Being equipped with such tool, it allows you to check various measure such as cache utilization (and basically memory layout), which - if used to its full extend - provides a huge boost to efficiency.

When you are sure that you algorithms and structures are optimal, then you should definitely use the multiple cores on i5 and i7. In other words, play around with different parallel programming algorithms/patterns and see if you can get a speed up.

When you have truly parallel data (array-like structures on which you perform similar/same operations) you should give OpenCL and SIMD instructions(easier to set up) a try.

Share Improve this answer

Follow

answered Jan 24, 2013 at 10:17

Red XIII

5,987 ● 4 ● 29 ● 29

8

huh, then final thing you may try: ACOVEA project: Analysis of Compiler Optimizations via an Evolutionary Algorithm -- as obvious from the description, it tries a genetic algorithm to pick the best compiler options for your project (doing compilation maaany times and check for timing, giving a feedback to the algorithm :) -- but results could be impressive! :)

Share Improve this answer Follow

edited Jun 28, 2016 at 11:33

answered Jan 24, 2013 at 1:43

zaufi

7,089 ● 28 ● 35

5

Some notes about the currently chosen answer (I do not have enough reputation points yet to post this as a comment):

The answer says:

> `-fassociative-math`, `-freciprocal-math`, `-fno-signed-zeros`, and `-fno-trapping-math`. These are not included in `-Ofast` and can give some additional performance increases on calculations

Perhaps this was true when the answer was posted, but the GCC documentation says that all of these are

enabled by `-funsafe-math-optimizations`, which is enabled by `-ffast-math`, which is enabled by `-Ofast`. This can be checked with the command `gcc -c -Q -Ofast --help=optimizer`, which shows which optimizations are enabled by `-Ofast`, and confirms that all of these are enabled.

The answer also says:

> other optimisation flags which aren't enabled by any "-O" options... `-frename-registers`

Again, the above command shows that, at least with my GCC 5.4.0, `-frename-registers` is enabled by default with `-Ofast`.

Share Improve this answer

Follow

answered Jul 29, 2017 at 10:48

user3708067

613 ● 7 ● 13

"Again, the above command shows that, at least with my GCC 5.4.0, -frename-registers is enabled by default with -Ofast." At May 2024, gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html mentions that `-frename-registers` is enabled with `-funroll-loops`, yet `-funroll-loops` is *not* enabled with `-Ofast`. – Bart May 28 at 12:46

1

It is difficult to answer without further detail:

- what type of number crunching?

- what libraries are you using?
- what degree of paralelization?

Can you write down the part of your code which takes the longest? (Typically a tight loop)

If you are CPU bound the answer will be different than if you are IO bound.

Again, please provide further detail.

Share Improve this answer

Follow

answered Jan 24, 2013 at 3:25

Escualo

42k ● 26 ● 91 ● 137

1

I would recommend taking a look at the type of operations that costitute the heavy lifting, and look for an optimized library. There are quite a lot of fast, assembly optimized, SIMD vectorized libraries out there for common problems (mostly math). Reinventing the wheel is often tempting, but it is usually not worth the effort if an existing soltuion can cover your needs.Since you have not stated what sort of simulation it is I can only provide some examples.

http://www.yeppp.info/

http://eigen.tuxfamily.org/index.php?title=Main_Page

https://github.com/xianyi/OpenBLAS

Share Improve this answer

Follow

answered Dec 16, 2015 at 22:01

uLoop

225 ● 1 ● 9

-3

with gcc intel turn of / implement -fno-gcse (works well on gfortran) and -fno-guess-branch-prbability (default in gfortran)

Share Improve this answer

Follow

answered Jul 11, 2015 at 10:07

xTrameshmen

71 ● 1 ● 1