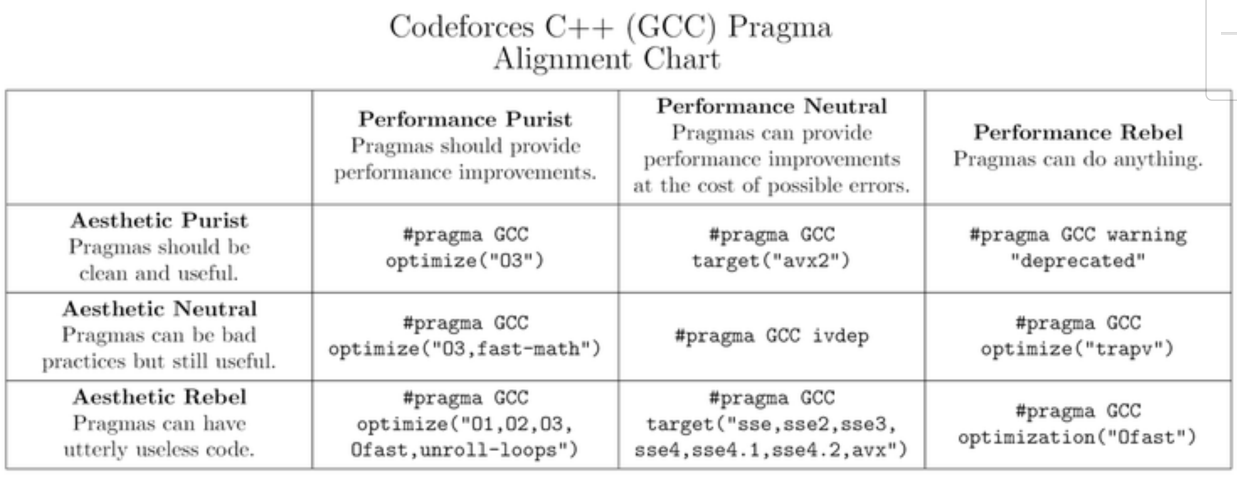
nor's blog

[Tutorial] GCC Optimization Pragmas

By nor, 15 months ago,

Introduction

A while ago ToxicPie9 and I made and posted this meme:



Codeforces C++ (GCC) Pragma
Alignment Chart

	Performance Purist Pragmas should provide performance improvements.	**Performance Neutral** Pragmas can provide performance improvements at the cost of possible errors.	**Performance Rebel** Pragmas can do anything.
Aesthetic Purist Pragmas should be clean and useful.	`#pragma GCC optimize("O3")`	`#pragma GCC target("avx2")`	`#pragma GCC warning "deprecated"`
Aesthetic Neutral Pragmas can be bad practices but still useful.	`#pragma GCC optimize("O3,fast-math")`	`#pragma GCC ivdep`	`#pragma GCC optimize("trapv")`
Aesthetic Rebel Pragmas can have utterly useless code.	`#pragma GCC optimize("O1,O2,O3,Ofast,unroll-loops")`	`#pragma GCC target("sse,sse2,sse3,sse4,sse4.1,sse4.2,avx")`	`#pragma GCC optimization("Ofast")`

To my surprise, many people are quite confused about it. In fact, I realized there are a lot of misconceptions about pragmas. Most C++ users on Codeforces put a few lines of pragmas at the start of every submission. However, I believe many of them don't fully understand what they're doing or how pragmas work; the only thing people seem to think is that "add pragmas -> code go brr".

Pragmas are a form of the dark arts that are feared and used, both correctly and incorrectly, by lots of competitive programmers. They are widely believed to make your code much faster, but sometimes may lead to slowdowns and even runtime errors on certain platforms. In this blog, I will explain the effects of `#pragma GCC optimize` and `#pragma GCC target`, how they work, and how you should and shouldn't use them.

Feel free to skip to the end of the blog for a TL;DR.

Warning

All of this discussion is for GCC, and the code you will write might not be portable across different compilers or platforms (in terms of turning on the same optimizations). However, you may assume that most of them work on Codeforces and other x86 platforms that are not too old.

Some Non-examples

The following does nothing.

```
#pragma GCC optimize(" unroll-loops")
#pragma gcc optimize("Ofast")
#pragma GCC optimization("Ofast")
#pragma optimize(Ofast)
```

Yes, these are real-life examples we have seen being used by many competitive programmers, including quite a few LGMs. Perhaps the third one among these came from this blog which seems to have popularized the idea of using pragmas, but with a small mistake. Many people are misled to believe that some of the above work, when they actually don't -- stop using them.

If ever in doubt about whether your pragmas are correct, turn on most compiler warnings with the command-line option `-Wall` (or the more specific `-Wunknown-pragmas`). For example, if you compile the code above with `-Wall` , you'll get the following output, which tells you that the pragmas are invalid and useless:

```
foo.cpp:2: warning: ignoring ‘#pragma gcc optimize’ [-Wunknown-pragmas]
    2 | #pragma gcc optimize("Ofast")
      |
foo.cpp:3: warning: ignoring ‘#pragma GCC optimization’ [-Wunknown-pragmas]
    3 | #pragma GCC optimization("Ofast")
      |
foo.cpp:4: warning: ignoring ‘#pragma optimize ’ [-Wunknown-pragmas]
    4 | #pragma optimize(Ofast)
      |
foo.cpp:1:37: warning: bad option ‘-f unroll-loops’ to pragma ‘optimize’
[-Wpragmas]
    1 | #pragma GCC optimize(" unroll-loops")
      |                                     ^
```

Try to check if your submissions have similar problems. If yes, then you have probably been using pragmas wrong the entire time -- it's time to change your default code.

What Is "#pragma GCC optimize"?

It turns on certain optimization flags for GCC. The syntax is

```
#pragma GCC optimize (option, ...)
```

From the official source on GCC pragmas, this pragma allows you to set global optimization flags (specified by `option`) for functions that come after it. Let's have a look at a few of them:

- `O0` , `O1` : These are pretty useless for competitive programming purposes, so we won't discuss these here.
- `O2` : This is the default optimization option on Codeforces, so using this might not give any tangible benefit.
- `O3` : This is the first non-trivial optimization option. It can make your code slower sometimes (due to the large size of generated code), but it is not very frequent in competitive programming. Some of the things it does are:
    - Auto-vectorize the code if the mentioned architectures allow it. This can make your code much faster by using SIMD (single instruction, multiple data) which kinda parallelizes your code on an instruction level. More info below.
    - Function inlining — inlines functions aggressively if possible (and no, marking functions as inline doesn't inline functions, nor does it give hints to the compiler)
    - Unrolls loops more aggressively than `O2` (this might lead to instruction cache misses if generated code size is too large)
- `Ofast` : This is one of the more controversial flags. It turns on all optimizations that `O3` offers, along with some other optimizations, some of which might not be standards compliant. For instance, it turns on the `fast-math` optimization, which assumes floating-point arithmetic is associative (among other things), and under this assumption, it is not unexpected to see your floating-point error analysis go to waste. `Ofast` may or may not make your code faster; only use this if you're sure it does the right things.

You can also use some other options, like:

- `unroll-loops` -- Enables aggressive loop unrolling, which reduces the number of branches and optimizes parallel computation, but might increase code size too much and lead to instruction cache misses.
- `unroll-all-loops` -- Usually makes the program slower.
- `strict-overflow` -- Enables some optimizations that take advantage of the fact that signed integer overflow is undefined behavior.
- `trapv` -- This one is quite special, as it is not usually considered an "optimization": enabling this will make your code run much slower, but causes signed integer overflows

to generate runtime errors. Useful for debugging. (Editor's note: if your system supports, it's recommended to use a sanitizer instead. You can find tutorials on the internet, so we won't discuss it here.)

A full list of supported flags in GCC can be found here.

An example: let's say you want to use `O3` and `unroll-loops`. Then you could do something like

```
#pragma GCC optimize("O3")
#pragma GCC optimize("unroll-loops")
```

or

```
#pragma GCC optimize("O3,unroll-loops")
```

or

```
#pragma GCC optimize("O3","unroll-loops")
```

Note that the options are strings inside quotation marks, and there's no space after or before the comma in the second way of writing. However, you can have a space before or after the comma in the last way of writing without any issues.

What Is "#pragma GCC target"?

Many modern computers and almost all competitive programming judges run on the x86 architecture. It has received a lot of additions over the years -- particularly, extra features (instructions) that enable different calculations or make existing ones much faster.

Compilers allow you to take advantage of these instruction sets extensions. For example, CPUs that support the `popcnt` instruction can theoretically compile `__builtin_popcount` into one instruction, which is much faster than usual implementations of this function. Similarly, if you want to auto-vectorize code, you'd need some instruction sets that actually support the vector instructions. Some of these instruction sets are `sse4.2`, `avx`, `avx2`. Here's a list of instruction sets that are usually supported on Codeforces:

- `avx` and `avx2`: These are instruction sets that provide 8, 16 and 32 byte vector instructions (i.e., you can do some kinds of operations on pairs of 8 aligned integers at the same time). Prefer using avx2 since it's newer.
- `sse`, `sse2`, `sse3`, `sse4`, `sse4.1`, `sse4.2`: These are instruction sets that are also for vectorization, but they're older and not as good as avx and avx2. These are useful for competitions on websites such as Yandex, where avx2 is not supported and gives a runtime error due to unrecognized instruction (it corresponds to a SIGILL signal —

ill-formed instruction).
- `popcnt` , `lzcnt` — These optimize the popcount (`__builtin_popcount` family) and count leading zeros (`__builtin_clz` family) operations respectively.
- `abm` , `bmi` , `bmi2` : These are bit manipulation instruction sets (note that `bmi` is not a subset of `bmi2`). They provide even more bitwise operations like `ctz` , `blsi` , and `pdep` .
- `fma` : This is not so widely used, since avx and sse make up for most of it already.
- `mmx` : This is even older than the sse* family of instruction sets, hence is generally useless.

You should be quite familiar with them if you have written SIMD code before. If you're interested in a certain instruction set, there are plenty of resources online.

An example: if you want to use, say, `avx2` , `bmi` , `bmi2` , `popcnt` and `lzcnt` , you can write

```
#pragma GCC target("avx2,bmi,bmi2,popcnt,lzcnt")
```

Again, note that we don't have spaces in the string.

There are two main ways to use `#pragma GCC target` .

- You can use it with the optimization pragmas. It allows the compiler to automatically generate efficient SIMD instructions from parts of your code (based on the optimization flags), often boosting their performance by roughly 2, 4 or even 8 times.
- It enables you to use the intrinsics of supported instruction sets. For example,

```
#include <immintrin.h>
// returns the odd bits of x: 0b01010101 -> 0b1111
uint32_t odd_bits(uint32_t x) {
    return _pext_u32(x, 0x55555555u);
}
```

This code works very efficiently, but won't compile unless you specify a valid target. Normally this is done by adding the `-mbmi2` compilation flag, but this is impossible on Codeforces, so you can only enable it by other ways like `#pragma GCC target("bmi2")` .

- An extreme example of this is this super-fast convolution code that makes heavy use of SIMD intrinsics to squeeze the running time of an $\mathcal{O}(N \log N)$ algorithm with a traditionally bad constant factor with $N \approx 5 \times 10^5$ to only 49ms.

Note that using vectorization targets won't work unless you have `O3` or `Ofast` on (more specifically, `tree-vectorize` , with at least `O2` -- note that `O2` is turned on by default on Codeforces).

Function Attributes

Just in case you don't want to apply these optimizations globally, you can use function attributes. For instance,

```
__attribute__((target("avx2"), optimize("O3", "unroll-loops"))) void work() {
    // do something
}
```

This enables these attributes for this function only, and the rest of the code will be compiled with the global options.

Bonus

`#pragma GCC ivdep` : This pragma forces the compiler to vectorize a loop if the loop was not vectorized due to possible aliasing. This means that the compiler wasn't able to prove that vectorizing is safe due to data dependency, but you can, due to knowledge of the problem at hand, show that this case will never happen. This results in undefined behaviour if you're not careful enough about there not being a dependency across loop iterations. Think of this having a similar idea behind it as pointer restrict in C99.

`#pragma GCC push_options` , `#pragma GCC pop_options` : These maintain a stack of target and optimization pragmas, enabling you to temporarily switch to another set of options. `#pragma GCC reset_options` resets all the options.

`#pragma GCC optimize "Ofast"` and `#pragma GCC optimize "-Ofast"` also surprisingly work. The same holds for stuff like `#pragma GCC optimize "-funroll-loops"` and `#pragma GCC optimize "unroll-loops"` . However, `#pragma GCC target "avx2"` works but `#pragma GCC target "-mavx2"` doesn't.

Some Caveats

As we have pointed out already, there might be some caveats associated with using the aforementioned pragmas.

- Some of these optimize/target pragmas can potentially make your code slower too, due to code size and other associated stuff.
- You might get runtime errors if you're careless about your choice of instruction sets. In general, on any platform, you should try to use each instruction set (and also write code that benefits from that instruction set!) and see if you're getting weird behaviour or not. There are also some ways to check if the machine your code is going to be run on has certain instruction sets at compile time, but inconsistencies with the actual set of instruction sets are possible in subtle ways, so you shouldn't in general depend on them.

A good way, however, is to run custom tests with stuff like `assert(__builtin_cpu_supports("avx2"))` .

- In general, constant-factor optimization at such a low level is somewhat unpredictable, and you should rely on measuring the performance rather than guessing from the instruction counts/latencies. This also means taking the contents of this blog with a grain of salt and not blindly believing that using these pragmas will somehow help you cheese every $\mathcal{O}(n^2)$ solution into the time limit for $n = 10^5$. A good way to look at generated assembly is by using godbolt.org.

Some Examples

There are multiple incidents in codeforces folklore that point to how fast pragmas can make your code. Some instances are:

- https://codeforces.com/blog/entry/89640
- https://codeforces.com/blog/entry/66279
- https://codeforces.com/blog/entry/81467 (unroll-loops isn't actually used)

Some more helpful discussion can be seen in the thread for this comment: https://codeforces.com/blog/entry/94609?#comment-836718

Conclusion/TL;DR

If you skipped the entire blog to read this part, it's recommended that you also read the Some Non-examples section and check if you're using an incorrect pragma.

No pragma is absolutely safe to use. Don't blindly believe that one line of pragma can suddenly, magically boost the speed of all your submissions. Often, you will need some testing or researching to decide how to improve your constant factor.

When restricted to competitive programming on Codeforces or a fixed platform, however, some results might be more predictable. Specifically,

- The default `O2` already has a lot of optimizations compared to `O0` . Increasing it to `O3` or even `Ofast` might increase or decrease the performance, depending on your code and input. Most online judges provide a "custom test" function which helps you figure out which one is more appropriate. You can also test other flags if you know what they do.
- Usually, the `avx` and `avx2` targets are supported by newer online judges. When they are supported, these can usually make your code faster. They often work best when your code is easily vectorizable (e.g., tight loops) -- which requires some experience to write. Again, custom tests are always useful (also, make sure they don't cause errors due to architecture issues). You can try the `sse` family if they don't work.
- Other targets like `popcnt` and `bmi` don't usually matter a lot when the compiler is optimizing, however operations like `__lg` , `popcount` , etc. require them to be fast.

This is important when you use bitsets or do heavy bit manipulations.

Some pragmas I personally use on Codeforces are:

```
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
```

If on a platform with no avx2 support, I switch out the avx2 with either avx or one of the sse's. Some (ancient) judges might not have support for the other targets either, so it's usually a decision you need to make as mentioned in the blog.

Acknowledgments

Thanks to ToxicPie9 for helping me write this blog and suggesting both stylistic changes as well as things to include! Please give him upvotes.

optimization bad, optimize good

+854 nor 15 months ago 40

1

Comments (40)

Write comment?

ToxicPie9

15 months ago, # | +332

as a co-author, i helped making the meme and the post.

please upvote if you find the post helpful. we might post more about technical stuff in the future.

→ Reply

hbarp

15 months ago, # | ← Rev. 2 +45

Finally a blog that explains behind the scenes for pragma spells black magic. Thanks ! :pkinglove:

→ Reply

sslotin

15 months ago, # | +57

Very nice post. Most people in the competitive programming community are using pragmas very imprudently like if it was some magic, so demystifying them is extra important.

One thing to note is that most SIMD extensions are backwards compatible, so you only need to include the latest one instead of the `sse,sse2,sse3...` mess that people often use (except for the situational ones like `popcnt`). So just leaving the following two cover your needs in 95% of the cases:

```
#pragma GCC optimize("O3")
#pragma GCC target("avx2")
```

Also, it may or may not be beneficial to ask your compiler to tune your code for a specific architecture. GCC isn't very good at that, but it almost certainly won't make things worse:

```
#pragma GCC target("avx2,tune=native")
```

All in all, I guess for competitive programming this setup covers almost all and has a very low chance of backfiring:

```
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("avx2,tune=native")
```

Also, I wrote a broader article on the topic, and it has some other compiler optimizations if someone is interested.
→ Reply

15 months ago, # ^ | ☆ ← Rev. 2 +25

nor

Thanks for the comments, the backwards compatibility in most people's code is quite a mess (hence it warranted its own place in the meme :)).

I tend to avoid `tune=native` since it sometimes breaks some stuff locally/on the judge (for instance, compilation error here: https://godbolt.org/z/53c877nbh). I haven't tried that yet on CF though. Also I tend to not use Ofast since I've heard about people's code breaking on geometry problems, and a couple of times it led to much slower code than O3 (though it's quite rare, so it should be fine anyway).

Your book that you linked is pretty nice by the way, hoping to see more stuff like that!
→ Reply

15 months ago, # ^ | ☆ ← Rev. 2 +24

Kyou_mo_kawaii

You explained why sse, mmx etc are redundant because of avx2. But why omit the bit manipulation stuff like popcnt, lzcnt, abm, bmi, bmi2? Can you give an example of when these can "backfire"? Also it seems like fma is not always covered by avx2 either right?

I am tempted to use this as my default now:

```
// change to O3 to disable fast-math for geometry problems
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC
target("avx2,popcnt,lzcnt,abm,bmi,bmi2,fma,tune=native")
```

(and @nor, it would super useful if you had a *correct* copy-and-pastable snippet in your tldr. If I googled your blog post during the contest I would probably end up copying the first block of code I saw, which is the one that does nothing lol)

EDIT: sslotin I just started looking at your book and it's fucking amazing! I highly recommend anyone who was interested enough to click on this blog post to read it too.

→ Reply

15 months ago, # ^ | +21

nor

I think tune=native should be used cautiously; you should check if it compiles.

Also regarding the copy-and-pastable snippet, I mentioned in the blog that it depends quite a bit on the specific judge machines, so it's not exactly copy-and-pastable. And if anything, it should already be in your template so you don't need to copy paste from this blog post :)

However I added the pragmas that I tend to use on Codeforces just for easy reference. Hopefully that helps.

→ Reply

15 months ago, # ^ | ← Rev. 2 +5

sslotin

By *backfire* I mean to cause the program to either crash or slow down significantly, or in general to behave differently on your laptop and on the testing system (without necessarily causing runtime errors). The more hardware-specific options and optimizations are enabled, the higher the chances of that happening.

To be safe you need to find out the exact microarchitecture the server is running, look up the list of its extensions and intersect it with what you have locally. I did this for CodeForces a while back when I was curious, and found out it was running a Skylake or something very similar, which has these extensions listed in the docs:

> *"skylake": Intel Skylake CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AVX2, AES, PCLMUL, FSGSBASE,*

RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, CLFLUSHOPT, XSAVEC and XSAVES instruction set support.

There are a lot of them, and I don't really know what a third of them does or which are included in which. The reason I recommend only leaving `avx2` is because it is the only thing that is almost guaranteed to also be enabled on your workstation (unless you are an Apple M1 user) and also probably the only extension that really matters — I doubt anyone has ever been bottlenecked by the throughput of popcount, to be honest.

Actually, I think for CodeForces in particular this is what you should to do *when submitting*:

```
#pragma GCC target("arch=skylake")
```

That's it, nothing else except optimization flags. This not only enables all available extensions and only them, but also tells the compiler additional info about the exact microarchitecture such as instruction latencies (this option implies `tune=skylake`). But be aware that if you leave this pragma uncommented when testing locally, there is a small chance it will crash or behave differently unless you are also running a Skylake.

On some onsite contests like the ICPC WF it is guaranteed that the testing servers will be identical to the workstations, so you can query the platform with `g++ -march=native -Q --help=target | grep march` and replace `skylake` with that.

(I have not tested it though. Please someone try it and tell if I am wrong.)

→ Reply

nor

15 months ago, # ^ | ☆ ← Rev. 2 +9

There's a quick way to check if the targets you're interested in are supported on an unfamiliar online judge: `assert(__builtin_cpu_supports("<target-name>"));` will throw a runtime error if the CPU doesn't support that target.

For an onsite contest, since you would have access to machines with the same configuration as the judging machines, it's nicer to run `g++ -march=native -E -v - </dev/null 2>&1 |`

`grep cc1` and see which targets work (`-mno-avx` means avx is not supported, `-mavx` means avx is supported).

Specifying architectures inside pragmas is kind of hopeless on GCC, for instance, `#pragma GCC target("arch=skylake")` (or anything trying to pass arch/tune parameters to the target) gives a compilation error: https://godbolt.org/z/K5xTeG3Pv

→ Reply

sslotin

15 months ago, # ^ Rev. 2 0

> *Specifying architectures is kind of hopeless on GCC*

Look at the errors: `attribute value 'arch=skylake' was already specified in 'target' attribute`. Some concurrency-related parts of the GCC C++ library are using exactly this method for performance, but I guess they are conflicting with ours because you can't specify architecture twice or something.

If we put that pragma after the library import this seems to work: https://godbolt.org/z/dWYPjfnWP. Not sure which parts of the standard library will be optimized in this case though.

UPD: it also works without changing anything in older (≤8) GCC versions.

→ Reply

nor

15 months ago, # ^ | 0

Yeah, as far as I can see, it suffices to keep only iostream before the import to avoid the compilation error. Too bad that `bits/stdc++.h` doesn't work here.

I tried making sure that some STL functions get optimized, but unfortunately couldn't manage to do that: https://godbolt.org/z/rh59TGbPE (so as of now I

can't find a way to use `arch=skylake` fruitfully).
→ Reply

sslotin

15 months ago, # ^ | 0

I think this is a bug in GCC 9. When defining hardware-specific library implementations, you are supposed to use `#pragma GCC push_options` with target, implement a function or a module, and then pop it back (which is a way to do target-inside-target), but in that one file that causes the error it is not done like that.

It is easy to fix, but for already installed compilers there is nothing we can do unfortunately. Still works on the C++17 CF compiler and most other platforms though.
→ Reply

ExplodingFreeze

15 months ago, # ^ | 0

Polygon claims to judge on the i3-8100, so targeting Coffee Lake might be ideal. Though as far as I know Coffee Lake has no drastic architectural changes from Kaby Lake or Skylake, so it may not lead to a noticeable benefit over targeting Skylake.
→ Reply

sslotin

15 months ago, # ← ^ Rev. 2 +3

I did some investigation, and yes, you are right: it is in fact Coffee Lake. And it shows the same model number running the gist I linked above.

![](https://i.ibb.co/rQn889h/Screenshot-from-2021-10-27-21-40-59.png)

However, as you noted, there aren't many architectural changes from Kaby Lake or Skylake. In fact, in terms of optimization-relevant features, there aren't any at all, so GCC does not distinguish between them and simply groups them all as "skylake".
→ Reply

Snow-Flower

15 months ago, # | ☆ +62

Norosity
→ Reply

SPyofgame

15 months ago, # | ☆ +15

As if there is a second upvote button, I would smash it to this blog.
→ Reply

Duy_e

15 months ago, # ^ | ☆ +31

smash yourself instead :>
→ Reply

SPyofgame

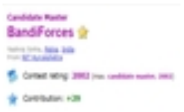
15 months ago, # ^ | ☆ 0

No thank you. I already smashed my rating down to pupil few days ago :>
→ Reply

Priyaanshut

15 months ago, # | ☆ +20

norz
→ Reply


KingRayuga

15 months ago, # | ☆ **0**

norz

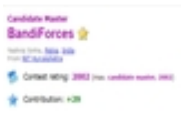
→ Reply

hbarp

15 months ago, # ^ | ☆ **+11**

Lets not forget ToxicPie9 also co-authored the blog !

→ Reply


KingRayuga

15 months ago, # ^ | ☆ **+18**

ToxicPie9 orz

→ Reply

hsekharsahoo

15 months ago, # | ☆ **+11**

norzosity

→ Reply

sslotin

15 months ago, # | ☆ ← Rev. 3 **+36**

Let's petition **MikeMirzayanov** to simply add `-march=native` (or `-march=skylake` or whichever architecture invokers are running on) to the compilation flags the next time he adds a new C++ compiler (preferably Clang). Then all the `target` pragmas can be omitted completely, only leaving optimization options which are situational.

This ensures that the compiler knows everything about the hardware while not increasing compilation time — in fact, it will probably save a lot of CPU resources (on average by 10-20% on all submissions would be my guess). I have no idea why it is not done so on all online judges.

→ Reply

prabhav_

15 months ago, # | ☆ **+9**

Heartfelt thanks from the bottom of my heart.

→ Reply

codemastercpp

15 months ago, # | ☆ **+9**

nor orz :prayge:

→ Reply

ssvb

15 months ago, # | +24

About AVX2. These instructions are also notorious for downclocking the CPU, depending on the number of CPU cores that are using AVX2 simultaneously. The exact effect of this varies depending on the CPU model. See https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/ (it's mainly about AVX-512, but AVX2 also has similar properties to some extent).

So on the Codeforces platform, the AVX2 pragmas may be potentially not very neighbour-friendly among other things. Imagine a quadcore judge machine, which processes 4 submissions in parallel. If 3 of them are actively using AVX2, then it's possible that the CPU may downclock and affect the performance of the unsuspecting 4th non-AVX2 submission too. Again, this may be a non-issue depending on the CPU model, but it would be great if Codeforces could try to do some experiments and benchmarks in this scenario.

The best solution may be just to disable AVX2 instructions on the judge machines.

→ Reply

ExplodingFreeze

15 months ago, # ^ | ← Rev. 3 0

As far as I'm aware AVX2 is nowhere near the power hog that AVX512 is.

See this analysis for some rough numbers in some CPU limited workloads. It is for rocket lake, so its applicability to most judges that are still using the skylake-like architectures isn't 1:1, but it still paints a rough picture of the effect of AVX/AVX2 on power draw.

Also I would highly expect that any decent judge would completely disable core turbos and/or take other measures to mitigate noisy neighbor issues even in the case of full loading of across all cores. And I do believe no judge right now supports AVX-512 for this exact reason.

→ Reply

priyansh_34

15 months ago, # | ← Rev. 5 0

Hi nor and ToxicPie9,

In the problem CF Edu. Round Problem B. I faced an issue, first I wrote the code without using any pragmas and it was accepted. 133543512

Then I tried with some pragmas

```
#pragma GCC optimize("Ofast,unroll-loops") #pragma GCC
target("avx2,popcnt,lzcnt,abm,bmi,bmi2,fma,tune=native")
```

On using them it displayed wrong in testcase 6.133542940.

Can you or anyone please explain, why this was happened. Any help will be very helpful. Thanks

PS: The error was caused by 2nd pragma, but I don't know why that was happening.
→ Reply

ToxicPie9

15 months ago, # ^ | ← Rev. 3 +26

~~I think the issue is unrelated with pragmas;~~ the solution should be incorrect even without them. **Update**: it is indeed related to pragmas in a weird way, see the comment below

On line 88, you calculated the answer of `ceil(rem / k)` by dividing two doubles. However, the `double` type can only store 52 bits of precision. Since both numbers can be close to 1e18 in magnitude, when you convert them to doubles, you might lose some precision and their values might change. (For example, if you convert the value `10000000000000001` $(10^{16}+1)$ into `double`, it becomes `10000000000000000` (10^{16}).) If one of the values `rem` or `k` changes during conversion, you are likely to get a wrong answer.

The more precise way to calculate the answer is to do integer division instead -- for **positive** integers a, b, the exact result of $\lceil \frac{a}{b} \rceil$ can be calculated in C++ with:

- `a % b == 0 ? a / b : a / b + 1`
- `a / b + (a % b != 0)`
- `1 + (a - 1) / b`

~~In fact, I do not understand why one of your solutions got accepted... Maybe the compiler did something suspicious? There's no issue on my end :thonk:~~
→ Reply

priyansh_34

15 months ago, # ^ | +16

Thanks ! For your nice and to the point explanation <3.
→ Reply

ToxicPie9

15 months ago, # ^ | +16

Update: Seems like it is caused by an "excess precision" issue that happens on specific compilers on Codeforces. See https://codeforces.com/blog/entry/78161 for more details.

After some testing, only "GNU G++17 (7.3.0)" on Codeforces seem to make the code use extra precision and output the correct answer. In this case pragmas is actually related to the problem: targets like AVX2 support 64-bit floating points, which fixes the excess precision issue and causes the submission to get WA correctly.

→ Reply

15 months ago, # | ☆ +8

Kyou_mo_kawaii

Are there any useful pragmas for controlling stack frame sizes in recursive functions for optimizing MLE? (today's div3's 1607F - Robot on the Board 2 was a good example, and also this past discussion)

It doesn't seem like `#pragma GCC optimize("Os")` does much. Looking at the list of gcc optimization flags, it does seem like this is something you can control but I don't know how to specify them as pragmas.

From: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html `-fconserve-stack Attempt to minimize stack usage. The compiler attempts to use less stack space, even if that makes the program slower. This option implies setting the large-stack-frame parameter to 100 and the large-stack-frame-growth`

→ Reply

15 months ago, # ^ | ☆ +3

nor

You can use `#pragma GCC optimize("conserve-stack")` for that. To check if this pragma works, you can use the compiler flags mentioned in the blog. In general, if you have an option like `-fsomething`, it makes sense to try `#pragma GCC optimize("something")` and see if it works.

→ Reply

15 months ago, # ^ | ☆ 0

Kyou_mo_kawaii

Thanks! Unfortunately it doesn't seem like `conserve-stack` did anything for my problem. I want to try the other flags on that page too but I still can't figure out the pragma syntax for stuff like `--param=large-stack-frame-growth=100`.

In particular I am trying to get a recursive solution for 1607F to pass. I did get it working for C++17 32 bit, but not for 64 bit (neither C++20 nor C++17): https://codeforces.com/contest/1607/submission/134144039

Do you have any tips?

→ Reply

Honey_Badger

15 months ago, # | ☆ 0

Why didn't you include tzcnt target in your pragmas list?

→ Reply

nor

15 months ago, # ^ | ☆ +24

tzcnt is an instruction that is supported in the bmi instruction set, so it isn't needed to be mentioned separately. Also, the last I checked, it gave me an unrecognized target error.

→ Reply

mukulrawat_03

4 months ago, # | ☆ 0

Can anyone explain which pragma function causes error and how ?

Solution 1-> Pragma on https://codeforces.com/contest/1737/submission/175313327

Solution 2-> Pragma off https://codeforces.com/contest/1737/submission/175313262

→ Reply

nor

4 months ago, # ^ | ☆ +9

It is probably due to some pragma changing how excess precision (in this case) is handled by the C++ compiler on Codeforces you are using (which is 32-bit, and there are a lot of weird things that happen with precision with the Codeforces 32-bit C++ compiler), but I might be wrong. It is probably because the compiler could be performing some optimizations in the code without pragmas but not the one which has the pragmas (the pragmas might be telling the optimizer to not optimize that code but do some other optimization instead), that it is able to keep extra precision in the code without the pragmas.

Ideally, your code should have gotten WA both times, since you are using `std::sqrt` on a `long long` which causes loss of precision. So there is nothing wrong with the pragmas (you were just relying on a compiler specific behavior). Submitting on a 64-bit compiler validates the above theory, the relevant submission can be found here: 175318712.

→ Reply

4 months ago, # ^ | ☆ 0

Thanks

→ Reply

mukulrawat_03

ScorpioDagger

new, 2 months ago, # | +5

Very informative elaborating what is happening underneath the hood. I reached this blog from google. It helped me get this 552D - Ваня и треугольники from 1600 ms to 1270 ms using a brute-force approach.

→ Reply

↑
1
↓