Proprietary and Free Fortran Compiler Optimizations: or GCC is better than Supposed, Part 2

Dave Love
dave.love@manchester.ac.uk

2021-10-20

Measurements of run times of a Fortran benchmark set on Intel SKX and IBM POWER9 demonstrate that proprietary compilers — particularly Intel's — don't have the commonly-supposed general superiority over free ones. Suitable optimizations make the mean performance for `gfortran` essentially the same as for Intel `ifort` on this set (but not IBM XL). Some outliers are analysed to understand missed optimizations, resulting in one GCC bug fix so far.

1 Perspective

To first order, 'everyone' seems to 'know' that proprietary tools and libraries provide much better performance than free software — specifically the Intel compiler(s)[1] and Intel MKL. One sees (unqualified) statements that `ifort` produces code that's N times faster than `gfortran`, for various values of N. Similarly for MKL and free BLAS, which should be the subject of another note.

Those are myths, presumably spread by people who either haven't made measurements, or have misinterpreted measurements.[2] Part 1 of this series showed the effectiveness of GCC on MPI internals compared with using hand-coded intrinsics. This note presents measurements on a benchmark set. There's also some explanation from (limited, so far) investigation; benchmarks are of rather limited use unless they're directed at understanding the system and how they're run.

The Intel myth probably arises at least partly from compiler defaults. Intel's are actually non-standard-conforming, at least by including a rough equivalent of GCC's `-ffast-math`. That, for instance, allows vectorizing more widely (e.g. reductions), and restructuring loop nests.[3] Also, vectorization isn't currently included in the most commonly used GCC optimization level, `-O2`, and binaries are often built with a minimum level of hardware support — only SSE2 for x86_64 — especially if they are from distribution packages.

Most times I've been told GCC won't vectorize something that Intel will, and I've had the code to try, I have been able to vectorize it. There's a notable, but understood, counterexample in the `mp_prop_design` code below, however.

Even if the code generated by a proprietary compiler is somewhat faster on average, as you might expect if it comes from a hardware company with deep architectural knowledge, you may consider how much that matters. It will have relatively little effect in HPC applications limited by communication, filesystem i/o, or tuned library performance, and performance is likely not reproducible to better than ∼10% anyway. Also reliability is important, and Intel `ifort` has proved surprisingly problematic compared with `gfortran` in substantial research computing support experience.[4] It's also infinitely slow on POWER or ARM! We should compare proprietary compilers

[1] I don't know whether C and Fortran are actually separate compilers, unlike with GCC.

[2] A hard-bitten background in experimental physics leads me to take *understanding* measurements, and potential problems with them, more seriously than many.

[3] Typically the `-funsafe-math-optimizations` component of `-ffast-math` is actually enough for that.

[4] At least it's a lot better than the infamous 'Pentium GCC' which was once the bane of people fielding GCC bug reports, but surprising, given its heritage from DEC Fortran, which was highly-regarded.

on those platforms, but I can only readily do so on POWER9, with results below. (Arm's proprietary compiler only advertises partial Fortran 2008 support, and only OpenMP 3.1. I don't have much information on Fujitsu's and Cray's.)

It is worth noting that all(?) the major proprietary compilers seem to be moving to being based on LLVM, so at least their backends are likely to have much in common. However, despite all the attention on LLVM, GCC remains competitive.

The Polyhedron benchmark set for Fortran has been investigated. It happens to be there, is used to compare proprietary compilers, and shows GCC (`gfortran`) somewhat poorly in the published results. (They use GCC released in 2015 and `ifort` 17, presumably is from 2017.) Even so, `gfortran` is $< 20\%$ slower in the bottom line, discounting auto-parallelization, which you're not so likely to use. The programs in the set may not be very representative of most HPC workloads, for which you wouldn't normally consider a serial set anyway, so it would be worth also using, say, the OpenMP version of NPB.

The benchmark set has also been used to compare things like error-reporting and standards conformance, but this note is confined to performance of the generated code.

2 x86_64 measurements

Results of an attempt at a fairer comparison are shown in table 1 with what appear to be a priori reasonable 'best' flags. It is really unrealistic always to compile with the same flags — perhaps even for different routines in the same program — but that's how the framework is defined to run. Particularly the *fast-math*-type options used may break some code.

The geometric mean for `gfortran` in the best (profile-directed) mode is essentially equal to `ifort`'s. Profile direction made no significant difference to the `ifort` results, surprisingly, so is omitted. The GCC and `ifort` versions were the latest available at the time, but they appear mostly quite insensitive overall to different recent versions; compare the results for the system compiler, GCC 8, versus GCC 11, which is currently the latest. Significant differences do occur from version to version in individual tests, most notably for `fatigue2`, where a critical optimization seems to be chosen unstably. Such differences may be due to performance bugs or instabilities in the generally intractable problem of optimal optimization. Results are also included for the NAG compiler and the unrealistically old system version of Flang, which didn't have a newer backport.

The measurements were made on a quiet workstation running a Debian 10 desktop, with a Skylake-X (AVX-512) processor. SKX might be expected to favour the Intel compiler, since it appears relatively hard to optimize for compared with Haswell (AVX2), say. However, I haven't tried to assess how important vectorization, in particular, is for each case.[5]

The tests were bound to one core, and the `tuned(1)` profile `latency-performance` was in effect. The 'standard' parameters from the benchmark set were used, so a typical invocation would be

```
    hwloc-bind core:3 ../pbharness gfor-11 standard
```

Flags for GCC were chosen to be as similar as possible to `ifort -fast`, although `-fexternal-blas` was omitted for simplicity as it wasn't effective; BLAS was only used in one case with default `-fblas-matmul-limit`, where `matmul` wasn't a bottleneck. GCC defaults to `-mprefer-vector-width=256` on SKX, i.e. half-width AVX512 because of the well-known clocking down caused by full-width AVX512. `ifort` doesn't appear to offer the choice. However, changing the width to 512 made little difference to the mean; only `doduc` and `test_fpu2` saw any significant benefit, and surprisingly only $\sim 10\%$ for `test_fpu2`, while others were worse. It is also surprising that `linpk`'s linear algebra doesn't benefit.

Debug options were added, in line with the usual instructions to users to provide some chance of diagnosing failures (except for Flang, which failed some cases with `-g`). I don't know whether other compilers have the same promise as GCC that debugging information doesn't affect the generated code, which makes `-g` always worthwhile unless small binary size is crucial.

Input/output was to a local magnetic disk. It was quite large in at least one case, but running on `/dev/shm` didn't help. The timing error estimate produced by the framework isn't shown for simplicity, but was mostly below 0.1%, with exceptions starred (which are still much better than usually achieved in realistic cases on a compute cluster).

[5]It might be worthwhile to run MAQAO's `oneview` profile analysis for all the cases, rather than just the ones below.

The test system ran back-ported Linux 5.10, not the 4.19 default. That was found generally to improve performance, perhaps due to better side-channel security mitigations somehow, even for compute-intensive programs.[6] (The microcode package was current, but I failed to note which microcode version was actually running.)

2.1 Discussion

The results may be affected by bad code. `gfortran` warns about invalid code in two cases. It was actually necessary to fix an inconsistent interface declaration/use in `gas_dyn2` to compile it initially with `gfortran -fwhole-program`; however, that wasn't necessary with the `-flto` that was used ultimately.[7] There are also suspicious floating point exceptions reported by three cases marked in table 1, but running with the `undefined` and `address` sanitizers and using `-finit-real=snan` didn't immediately locate the problems. Then, `rnflow` actually fails validation of the output with `gfortran` and `nagfor`.

`doduc` has array bound errors at compile time, but it's not immediately clear whether just upping bounds of the arrays in common is correct. The poor convergence of the error bound in other cases that don't use random numbers might suggest uninitialized variables, despite not being picked up as warnings or by instrumentation other than one 'referenced but never set' from `nagfor` in `air`.

In view of those problems, table 1 shows also the geometric mean with the potentially problematic results excluded. That favours `gfortran` over `ifort`, which probably shouldn't be taken too seriously, especially with a factor of two difference for one excluded case.

2.2 Profiling

Only understanding the results makes benchmarks worthwhile. We can use `perf`, for instance, to provide some information about cases of interest, such as where `gfortran` loses to `ifort`.

`ac` wastes a lot of time in `DMOD`, although the intrinsic is inlined, unlike with POWER. `gfortran` gets considerably closer to `ifort` if `DMOD` is replaced with its definition. The code generated from expanding the `__builtin_fmod` to which `DMOD` is compiled is poor. It looks as if MOD should actually be compiled as its Fortran definition, especially as the builtin is only inlined with (some component of) `-ffast-math`. (Possible GCC issue not yet raised.)

`fatigue2` has highly unstable performance, e.g. between compilation with GCC 8 and 11. The difference seems to be in whether or not the function `generalized_hookes_law` is inlined, which somehow is saving time otherwise spent in the loop using `dot_product`. It isn't immediately obvious what goes on. (Possible GCC issue not yet raised.)

`gas_dyn2` spends a lot of time in `minloc` and MAQAO suggests it suffers from unaligned allocations. It writes 169MB, so i/o routines might be significant, which I think are not a `gfortran` strong point. The main hardware counters don't suggest an immediate explanation for the `gfortran`/`ifort` difference. For `gfortran` `L1-dcache-load-misses` from `perf` is 23%, `LLC-load-misses` is 86%, and 41% and 73% respectively for `ifort`. Notably the reported `iTLB-load-misses` is 752% for `gfortran` and 1093% for `ifort`.

`mp_prop_design` is penalized by a remarkably longstanding failure to vectorize the result of optimizing `cos` and `sin` together, whether they're combined with a complex value, or `__builtin_sincos`.

`rnflow` has missing vectorization indicated by MAQAO, which the `-fopt-info-missed` option says is due to control flow in the block and not enough data refs. `perf` shows much higher `L1-dcache-load-misses` than with `ifort` and about double the `LLC-load-misses`. It suffers from integer overflow exceptions with GCC and NAG, but apparently not with `ifort`. The GCC results failed validation non-reproducibly, which was not fixed with `-O3` instead of `-Ofast`. Since there's also a failure with NAG — which is usually accepted as most likely to be correct — there probably is a problem with the code.

[6]Published material investigating the effect of the mitigations on computational systems is rather inconsistent — or at least it was, perhaps a couple of years ago.

[7]It's somewhat unclear whether one should use `-fwhole-program`, `-flto`, or perhaps both.

Program	GCC 8	GCC 11	GCC 11 W512	GCC 11 PDO	ifort 2021	NAG 7	Flang 7
`ac`	6.08	5.95	6.46	6.51	4.27	5.97	7.01
`aermod`	6.31	6.41	6.92	5.33	6.86	9.68	*6.52
`air`	2.08	2.06	2.25	2.03	1.82	2.29	*2.41
`capacita`	13.53	*12.41	*12.85	*12.14	*11.86	14.06	12.07
`channel2`†	72.56	73.44	74.11	73.97	69.84	90.23	*75.11
`doduc`†‡	*8.40	9.07	7.74	7.31	9.99	—	*8.46
`fatigue2`	26.17	43.49	46.14	21.15	*45.61	117.16	95.86
`gas_dyn2`‡	*71.94	72.87	*77.08	*66.44	*35.66	68.90	53.80
`induct2`	21.83	21.95	21.84	21.85	*27.92	41.30	43.37
`linpk`	3.22	3.22	3.11	3.10	3.16	4.46	*4.20
`mdbx`	5.43	4.35	4.38	4.12	4.03	5.55	5.13
`mp_prop_design`	65.62	59.90	63.21	*61.03	51.05	181.35	*95.46
`nf`	5.21	5.27	5.30	5.28	5.26	7.67	8.20
`protein`	16.84	16.33	16.30	15.23	19.61	19.19	18.19
`rnflow`†	16.36	16.35	17.00	15.42	9.60	—	16.40
`test_fpu2`	28.21	27.97	25.05	25.52	23.95	43.42	31.94
`tfft2`	*30.27	*28.88	*28.19	*28.61	50.31	43.22	*44.47
geometric mean	14.18	14.27	14.44	13.10	13.21	21.04	17.49
selected mean	11.37	11.39	11.59	10.46	11.55	17.16	15.24

Notes:

* Poor convergence to low variance (repeats $\gg 10$), final error typically $\gtrsim 0.1\%, \ll 1\%$;

† FP exceptions signalled with at least one compiler. Fixed for `doduc` (only) with GCC by `-fno-fast-math`. NAG-compiled `doduc` and `rnflow` crashed with a segmentation violation;

‡ Invalid source.

'selected mean' is the geometric mean excluding the dubious sources `channel2`, `doduc`, `gas_dyn2`, and `rnflow`.

GCC 8 `gfortran` 8.3.0 `-static-libgcc -static-libgfortran -g -march=native -Ofast -funroll-loops -mveclibabi=svml -flto -frecursive -lsvml -lintlc`;

GCC 11 `gfortran` 11.1.0, options as GCC 8, but with `-gdwarf-4`, not `-g`;

GCC 11 W512 `gfortran` 11.1.0, options as GCC 11, but with `-mprefer-vector-width=512`;

GCC 11 PDO Profile-directed version of GCC 11 W512. Build twice with the GCC 11 W512 flags, but add `-fprofile-generate` to the first, `-fprofile-use` to the second, and run the program between them;

ifort 'Intel Fortran 2021.2.0 20210228' from the oneAPI distribution, flags `-fast -debug`. Profile-guided results are elided because they weren't significantly different;

NAG 7 'NAG Fortran Compiler Release 7.0(Yurakucho) Build 7036', flags `-g -Bstatic -recursive -Ounroll=4 -O4 -Ounsafe -ieee=nonstd`. Using `-target=core2`, rather than the default Pentium 4(?), gave slightly worse results;

Flang 7 `flang` 7.0.1 with `-march=native -Ofast`.

Table 1: Benchmark times in seconds on SKX, 3.60 GHz W-2123, Debian 10 plus back-ported Linux 5.10, using `tuned`'s `latency-performance` profile and default kernel parameters apart from a few filesystem and networking `sysctl`s.

Program	GCC 10	GCC PDO	XLF	XLF PDO	nvfortran	PGI
`ac`	28.06	28.13	5.76	5.92	29.29	29.99
`aermod`	23.50	20.66	11.93	21.57	18.61	17.25
`air`	3.31	*2.60	*2.77	*2.70	2.90	*3.30
`capacita`†	16.25	16.59	15.96	19.37	15.43	15.40
`channel2`†	38.31	38.30	*49.76	49.81	37.00	36.96
`doduc`†	15.52	13.79	11.86	*12.35	18.65	17.48
`fatigue2`	27.00	*27.02	88.93	*98.32	172.23	178.04
`gas_dyn2`	57.10	57.52	72.04	59.55	66.08	66.10
`induct2`	72.72	72.91	76.38	84.26	130.24	132.75
`linpk`	2.80	2.83	2.80	*2.82	3.10	3.11
`mdbx`	8.37	8.17	6.97	6.60	8.12	8.14
`mp_prop_design`	*183.07	*184.22	55.61	55.36	112.12	118.09
`nf`	5.20	5.20	4.82	4.90	8.12	8.00
`protein`	22.05	20.41	36.72	35.02	22.94	22.76
`rnflow`	32.49	35.19	15.89	16.18	29.07	29.04
`test_fpu2`	52.81	52.75	36.03	36.20	61.97	59.48
`tfft2`	*40.30	*40.80	*41.51	41.93	47.42	47.50
geometric mean	22.25	21.67	18.65	19.54	25.98	26.07
selected mean	22.46	21.91	18.16	18.89	26.92	27.16

* and † as table 1

GCC 10 `gfortran` 10.2.0, `-static-libgcc -static-libgfortran -g -mcpu=native -Ofast -funroll-loops -flto -frecursive -mveclibabi=mass -lmass -lmass_simdp9`;

GCC PDO As GCC 10, but using `-fprofile-generate`/`-fprofile-use`;

XLF `xlf` 16.1.1, `-O5 -g`;

XLF PDO as XLF but profile-directed, using `-qpdf1` and `-qpdf2`;

nvfortran `nvfortran` 20.9, `-fast -g -tp=pwr9`;

PGI `pgfortran` 19.10, `-fast -g -tp=pwr9`; I don't know what the relationship is between pgfortran and nvfortran.

Table 2: Benchmark times in seconds on POWER9, revision 2.3 (pvr 004e 1203), 3783 MHz, under RHEL 7.6 with the `performance` `scaling_governor`, and default kernel parameters but (apart from networking ones) some scheduling `sysctl`s that shouldn't be relevant.

3 POWER9 measurements

Measurements were also made on POWER9 with the available compilers, similarly to x86_64, and shown in table 3. (It isn't clear to what extent the Nvidia and PGI compilers are different implementations.)

After 60+ years and subsequently pioneering many of the relevant optimizations, you might expect IBM Fortran to be good, but `gfortran` still beats it in some cases, again illustrating the danger of blanket statements about compiler optimization.

3.1 Discussion

XLF warns about potential storage association issues with calls in `aermod`, `capacita`, `doduc`, `induct`, `linpk`, `mdbx`, `rnflow`, and `test_fpu2`. The warnings seem valuable but haven't been checked. (Surprisingly, and possibly related, `gfortran` with `-fopt-info` reports versioning of loops for possible aliasing despite Fortran semantics, though less with `-flto` than without.) Floating point exceptions occur similarly to the ones seen on x86_64.

GCC's (and LLVM's via nvfortran) dramatically poor performance on `ac` compared with XL is due to failing to inline `__builtin_fmod`. Expanding `DMOD` according to its definition reduces the GCC run time to 7.8 s, similarly to x86_64.[8] GCC suffers more with `mp_prop_design` than on x86_64 through its failure to vectorize the trig functions. Perhaps the IBM vector maths implementation is better than Intel's, or GNU `libm`'s `sincos` is poorer on POWER. While XLC will pattern-match a matrix multiplication loop and replace it with a call to `dgemm`, XLF-compiled `linpk` and `test_fpu2` don't call out to the ESSL library for any of the actual BLAS implementations they contain.

Overall, GCC holds up reasonably well against XL. The inlining of `fmod` has since been fixed, though not in a released version. However, it is unfortunate that the decade-old trig functions problem remains unresolved.

[8]A standalone `fmod` implementation doesn't get used and inlined by `-flto` when linked, as one might hope.