Building and running SNAP, using LLVM Flang

With Performance analysis

Mats Petersson
3-Apr-2022

Introduction

Why Fortran?

- It is still a popular language
  - Number 17 on the TIOBE list of languages in December 2021 (lower in March 2022)
- Particularly for mathematical/scientific community
  - Lots of Maths/floating point, intrinsics for lots of functions
  - Complex type part of the language
  - Good support for array operations
  - Allows more aggressive optimisation than C/C++ (almost always not aliasing)
  - Established in 1954, with the latest standard Fortran 2018 – so both old and modern
    - The language turns 70 in 2 years! :)
  - Support for OpenMP and OpenACC
- High usage in Supercomputing
- Large code-base of existing code
  - Some of which nobody wants to rewrite... Rewrites introduces new bugs! :)

LLVM Flang

- Project to make a high quality Fortran compiler on top of LLVM
- Written in C++
- Uses MLIR – multi-level IR
  - Higher level than LLVM-IR
  - FIR dialect models Fortran constructs
  - High level optimization passes
- Currently being merged to LLVM/main from the f18-llvm-project/fir-dev repo
  - https://github.com/flang-compiler/f18-llvm-project
  - https://github.com/llvm/llvm-project
- A few months from full support for Fortran 95 and OpenMP 1.1
  - So far focus has been on feature complete rather than optimisation

SNAP - introduction

- "SNAP serves as a proxy application to model the performance of a modern discrete ordinates neutral particle transport application."
  - I just barely got a passing grade in Physics, so don't ask me exactly what that means... :)
- About 8500 lines of Fortran 95 code with a few extensions using OpenMP 1.1
  - Big enough to be interesting, but not so huge it's impossible

Building and making it run

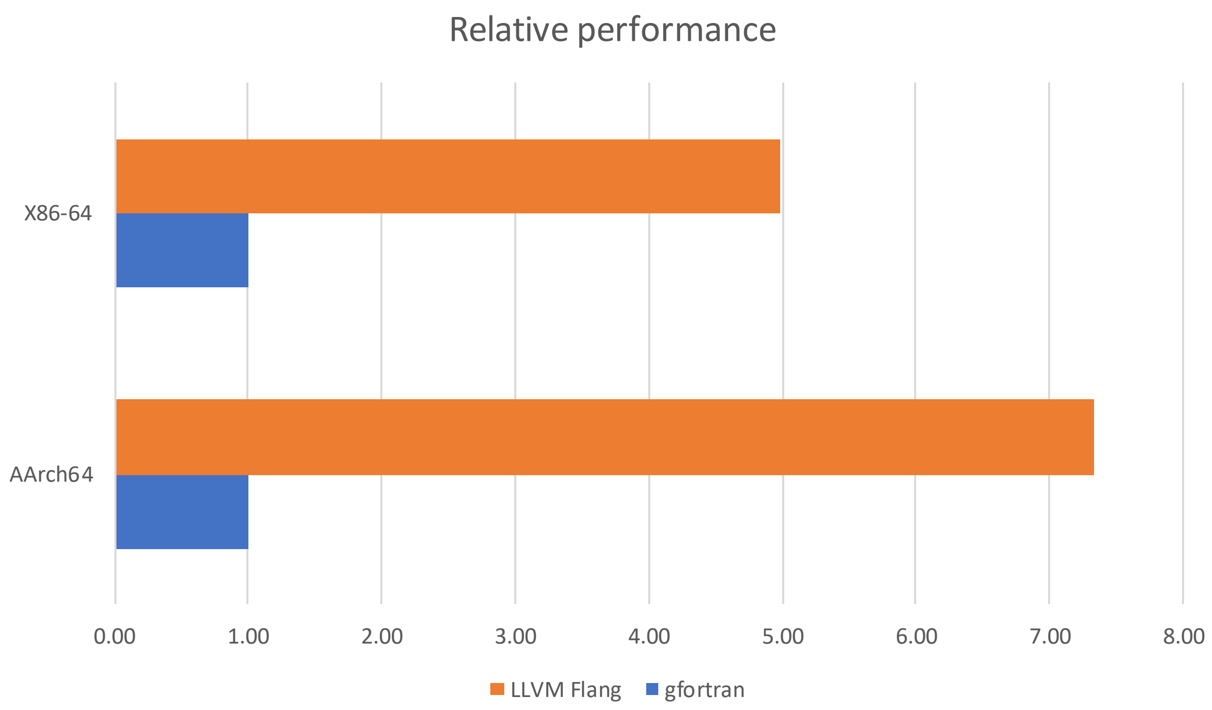
Two slides for 4 months...

Building SNAP

- Missing intrinsics (built in Fortran functions):
  - COMMAND_ARGUMENT_COUNT, GET_COMMAND_ARGUMENT, EXIT (Fortran 2003 functions)
- Bugs (see backup slides for example code):
  - Sliced arrays as output from subroutines didn’t get copied back
  - The omp_set_nested function caused ICE
  - OpenMP unstructured failed to compile
  - Induction variables are not in memory, but passed as references (this crashes!)
- Running SNAP:
  - At first, we ran SNAP with OpenMP turned off – even that didn’t work the first days
  - Once we got the basics working, turning on OpenMP increased the trouble factor
    - This was many steps of "This doesn't work, let's find a way to make it work"
  - Runs were compared with gfortran to make sure we're getting the same output
- All of this now works!

So, how fast or slow is it?

- In short: about 6 times slower compared to gfortran
- The immediate question then is "why is it that much slower?"
- And that's what the rest of this presentation is about



First pass of performance analysis

How we measured performance

- SNAP's output file: total execution time
  - `$ tail snap-output`
  - … Total Execution time 1.2345E+01 ...
- Using both x86-64 and AArch64 running Ubuntu Linux
  - Not comparing x86 with Arm, just for completeness (and the two main platforms for Flang)
- Modified the file qasnap/mms_src/2d_mms_st.inp
  - nx=80, ny=80, npey=1 (was nx=20, ny=20, npey=4)
- OpenMP enabled, but threads = 1, MPI turned off
- Using Linux perf tool to get profiling info to understand where we spend time
  - https://github.com/torvalds/linux/tree/master/tools/perf
- Presenting relative numbers rather than seconds

Perhaps various compiler tools can fix this?

- No support for –O<something> in LLVM flang at this point
- Using LLVM flang to generate MLIR:
  - `$ flang-new -fc1 -emit-mlir -S -fopenmp mms.f90`
- Use fir-opt with various options
  - ```
    $ fir-opt --basic-cse --cse --fir-memref-dataflow-opt --inline  --loop-invariant-code-motion  mms.mlir -o mms.o.mlir
    $ tco mms.o.mlir -o mms.o.ll
    $ clang -c mms.opt.ll -o mms.o
    ```
  - No real gains, and some options ICE (e.g. `--promote-to-affine`)
- Use LLVM opt with various options
  - `$ opt -O3 mms.ll -S -o mms.opt.ll && clang -c mms.opt.ll -o mms.o`
  - No real gains, no bad effects
- Use tco + clang with various options
  - `$ clang -c mms.opt.ll -O3 -o mms.o`
  - No real gains, no bad effects

These commands are examples!
    ```

So, now what do we do?

- Use perf to find where the time is spent!
  - Usual rule of 90% of time is spent in 10% of the code
- Figure out why the code is very different between gfortran and flang
- Hand-modify the generated FIR code
- Use tco + clang to compile to object file, and then use make command to link it
  - `$ tco mms-hand.mlir -o mms.opt.ll`
  - `$ clang -c -O1 mms.opt.ll -o mms.o`
  - `$ make`

Baseline perf results

- Gfortran

48.32%	gsnap	__dim3_sweep_module_MOD_dim3_sweep
23.94%	gsnap	__mms_module_MOD_mms_src_1._omp_fn.0fn.0
3.69%	libc-2.31.so	__GI___printf_fp_l
2.18%	libc-2.31.so	__vfprintf_internal
2.16%	libc-2.31.so	hack_digit
1.80%	gsnap	__expxs_module_MOD_expxs_slgg

Output results!

- Flang

54.58%	fsnap	_QMmms_modulePmms_src_1..omp_par
19.26%	fsnap	Fortran::runtime::DoTotalReduction<double, Fortran::runtime::RealSumAcc
15.35%	fsnap	_QMdim3_sweep_modulePdim3_sweep
2.91%	fsnap	_FortranASumReal8
1.64%	libc-2.31.so	_int_free
0.76%	libc-2.31.so	malloc

- Looking at `mms_src_1..omp_par` first

The mms_src_1 openmp parallel region

- This function is 124 lines of code. Most of the time is in an OpenMP parallel region that has 11 nested loops.
- There are 10 different places in the whole region that uses `qim(m,i,j,k,n,g)`
  - Each address calculation results in ~59 FIR operations or about 100 assembly instructions on Aarch64
- The innermost loop is essentially two lines:

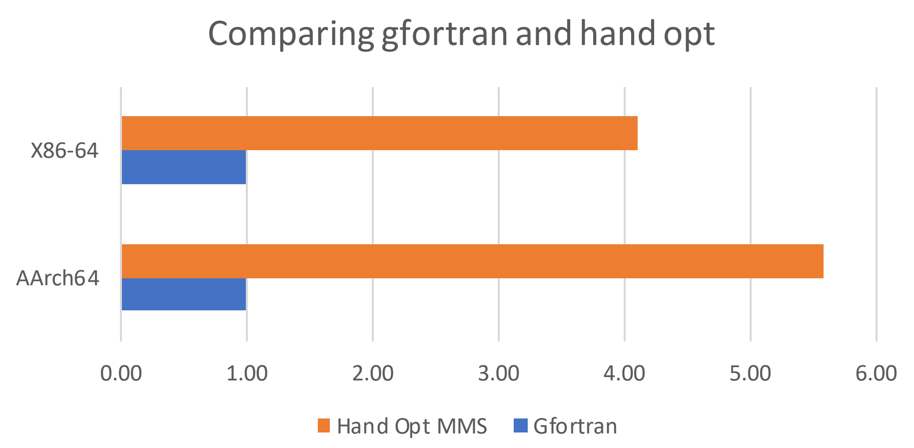
```
DO ll = 1, lma(l)
   qim(m,i,j,k,n,g) = qim(m,i,j,k,n,g) - ec(m,lm,n)*slgg(mat(i,j,k),l,gp,g)*ref_fluxm(lm-1,i,j,k,g)
   lm = lm + 1
END DO
```

- Even when using `clang -O3` on the mms.ll file
  - There are a total of 6 calculations for address of element in an array in that one line (twice for `qim(m,i,j,k,n,g)`
  - Those two lines turn into 230 FIR operations

Hoist address calculation code out of loop

- Moving the address calculation from inside the innermost loop to the next level out for all the six addresses – also only doing the `qim(…)` part once rather than twice.

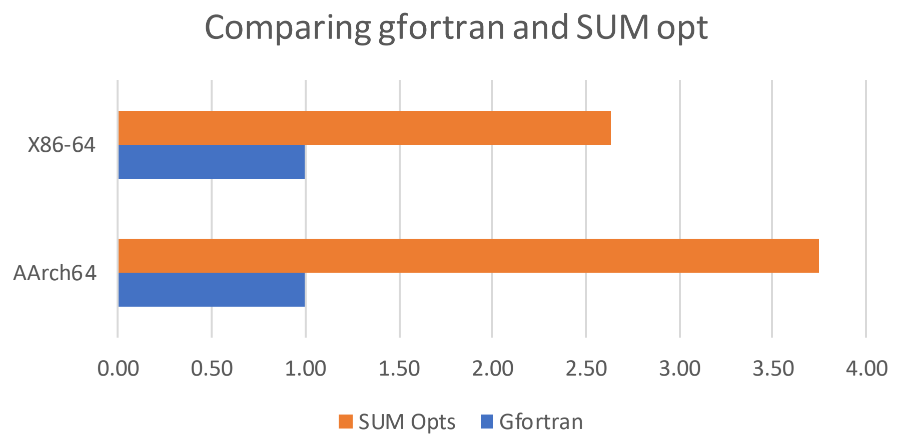
30.62%	fsnap	Fortran::runtime::DoTotalReduction<double, Fortran::runtime::RealSumAcc
27.33%	fsnap	_QMmms_modulePmms_src_1..omp_par
24.27%	fsnap	_QMdim3_sweep_modulePdim3_sweep
4.80%	fsnap	_FortranASumReal8
2.92%	libc-2.31.so	_int_free
1.20%	libc-2.31.so	malloc



Next, we attack the dim3_sweep

- Studying the code we see that the SUM() function is used in several places
- Writing simple sum1d() and sum2d() reduces the overhead over the generic variant

33.39%	fsnap	_QMmms_modulePmms_src_1..omp_par
27.26%	fsnap	_QMdim3_sweep_modulePdim3_sweep
26.15%	fsnap	_QMdim3_sweep_modulePsum1d
2.21%	libc-2.31.so	_int_free
1.52%	libc-2.31.so	malloc
1.04%	fsnap	Fortran::runtime::DoTotalReduction<double, Fortran::runtime::RealSumAcc



33% faster

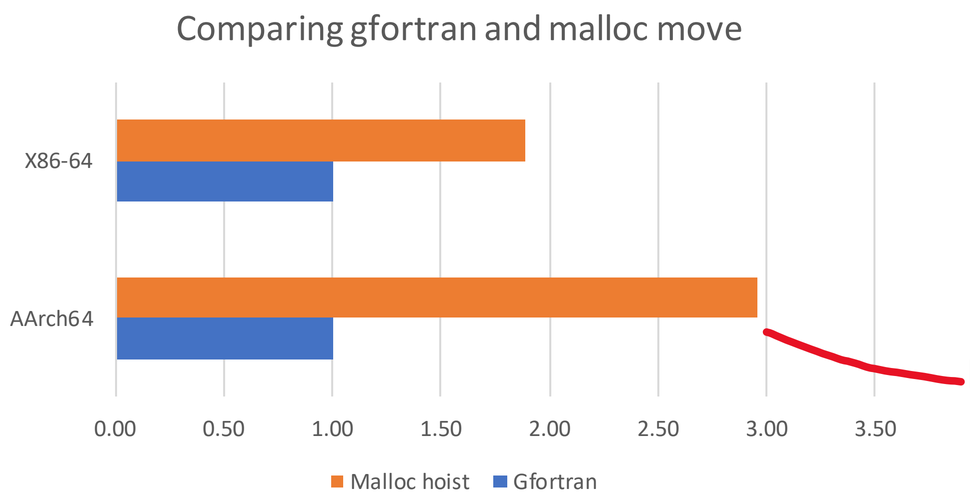
Move malloc/free out of loops

- There are several calls to malloc/free with constant(ish) sizes in the dim3_sweep code
- Moving those calls out of the loops reduces the overhead of those calls

49.74%	fsnap	_QMmms_modulePmms_src_1..omp_par
36.94%	fsnap	_QMdim3_sweep_modulePdim3_sweep
1.61%	fsnap	Fortran::runtime::DoTotalReduction<double, Fortran::runtime::NumericEx
1.43%	fsnap	_QMexpxs_modulePexpxs_slgg
1.40%	fsnap	Fortran::runtime::DoTotalReduction<double, Fortran::runtime::NumericEx
1.05%	fsnap	Fortran::decimal::BigRadixFloatingPointNumber<53, 16>::ConvertToDecima

Sum1d gets inlined

Output results!

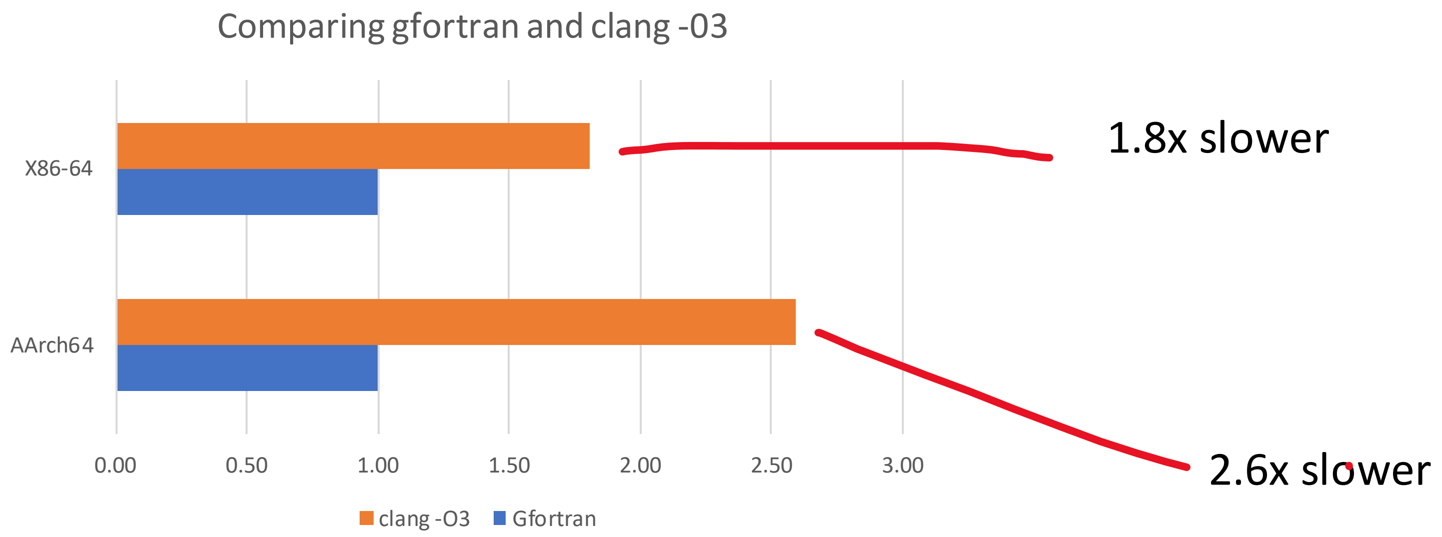


Less than 3x slower

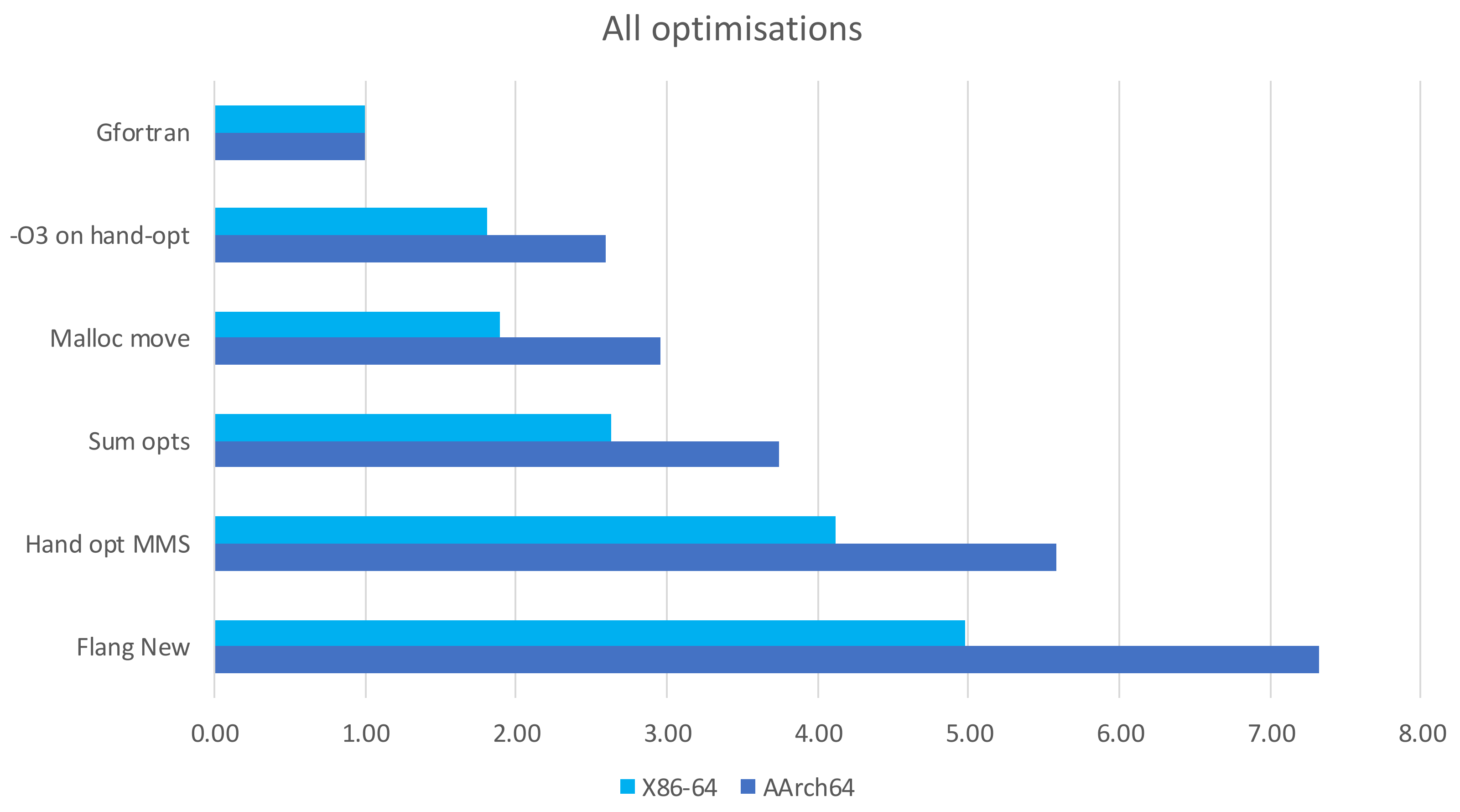
21% faster

Bonus gains

- Compiling the already optimized code with clang –O3 (instead of default opts)
  - `$ clang -O3 -c mms-hand.ll -o mms.o`



All optimisations in one graph



Summary

- Simple changes gives big improvements in performance
  - compiler SHOULD be able to do most of this
  - Lack of hoisting is due to missing alias info (confirmed)
  - The SUM() function has three calls to intrinsics, extra overhead vs inline solution
  - It would be good to avoid using malloc/free for smaller copies of arrays
- Next steps
  - Work on GitHub tickets
    - https://github.com/flang-compiler/f18-llvm-project/issues/1466,1499,1500,1501
  - SNAP CI – make sure we don't break what is working (done)
  - PR to SNAP -> flang support (done)
  - Make flang-new able to compile MLIR (in progress)
  - Implement optimisation in flang-new (in progress)
    - Add –O{0,1,2,3,…}
    - Support FIR level optimisations (e.g. library call replacements and maybe alias analysis at FIR level)

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה

The sum1d function

```
FUNCTION sum1d(arr)
    REAL(r_knd), DIMENSION(nang), INTENT(IN) :: arr
    REAL(r_knd) :: sum1d
    REAL(r_knd) :: res
    INTEGER :: i
    res = 0
    do i = 1, nang
        res = res + arr(i)
    end do
    sum1d = res
  END FUNCTION sum1d
```

The sum2d function

```
FUNCTION sum2d(arr)
  REAL(r_knd), DIMENSION(nang, 4), INTENT(IN) :: arr
  REAL(r_knd) :: sum2d
  REAL(r_knd) :: res
  INTEGER :: i, j
  res = 0
  do i = 1, nang
     do j = 1, 4
        res = res + arr(i, j)
     end do
  end do
  sum2d = res
END FUNCTION sum2d
```

Bug #1: sliced arrays not copied back

https://github.com/flang-compiler/f18-llvm-project/issues/1001

```
PROGRAM p
  INTERFACE
     SUBROUTINE fillme( a )
       REAL, DIMENSION(3, 3), INTENT(OUT) :: a
     END SUBROUTINE fillme
  END INTERFACE

  REAL, DIMENSION(3, 3, 3) :: d

  d = 2.0
  CALL fillme( d(:,:, 1) )
  print *, "d=", d
END PROGRAM p


SUBROUTINE fillme( a )
  REAL, DIMENSION(3, 3), INTENT(OUT) :: a
  a = 1.0

  print *, "A=", a
END SUBROUTINE fillme
```

Bug #2: omp_set_nested ICE

https://github.com/flang-compiler/f18-llvm-project/issues/918

```
MODULE PLIB_MODULE
  INTEGER :: nnested = 1
  LOGICAL :: do_nested

  CONTAINS

  subroutine omp_set_nested(enable) bind(c)
    import
    logical, value :: enable
  end subroutine omp_set_nested

  SUBROUTINE PINIT_OMP
    do_nested = nnested > 1
    call omp_set_nested( do_nested )

  END

END
```

Bug #3: OpenMP unstructured fail to compile

https://github.com/flang-compiler/f18-llvm-project/issues/1120
(This is one of multiple issues in this area – it's complicated!)

```
program n
  integer :: i
  !$omp parallel do schedule(static, 1) num_threads(5)
  do i = 1,5
     if (i == 1) cycle
     print *,i
  end do
  !$omp end parallel do
end program n
```

Bug #4: Induction variable not in memory

https://github.com/flang-compiler/f18-llvm-project/issues/1196

```
SUBROUTINE outer_src
  INTEGER :: k
  !$OMP PARALLEL DO SCHEDULE(STATIC,1) PRIVATE(k)
  DO k = 1, 4
     CALL outer_src_calc ( k )
  END DO
  !$OMP END PARALLEL DO
END SUBROUTINE outer_src

SUBROUTINE outer_src_calc ( p )
  INTEGER, INTENT(IN) ::  p
  print *, "p=", p
END SUBROUTINE outer_src_calc

PROGRAM crashing
  IMPLICIT NONE
  CALL outer_src
END PROGRAM crashing
```