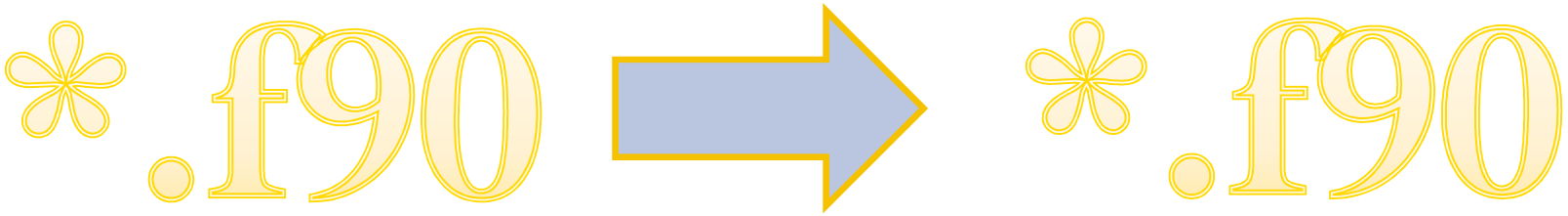
FRONT-END OPTIMIZATION IN GFORTRAN



Overview and options

FortranCon 2020

Thomas König, GNU Fortran team

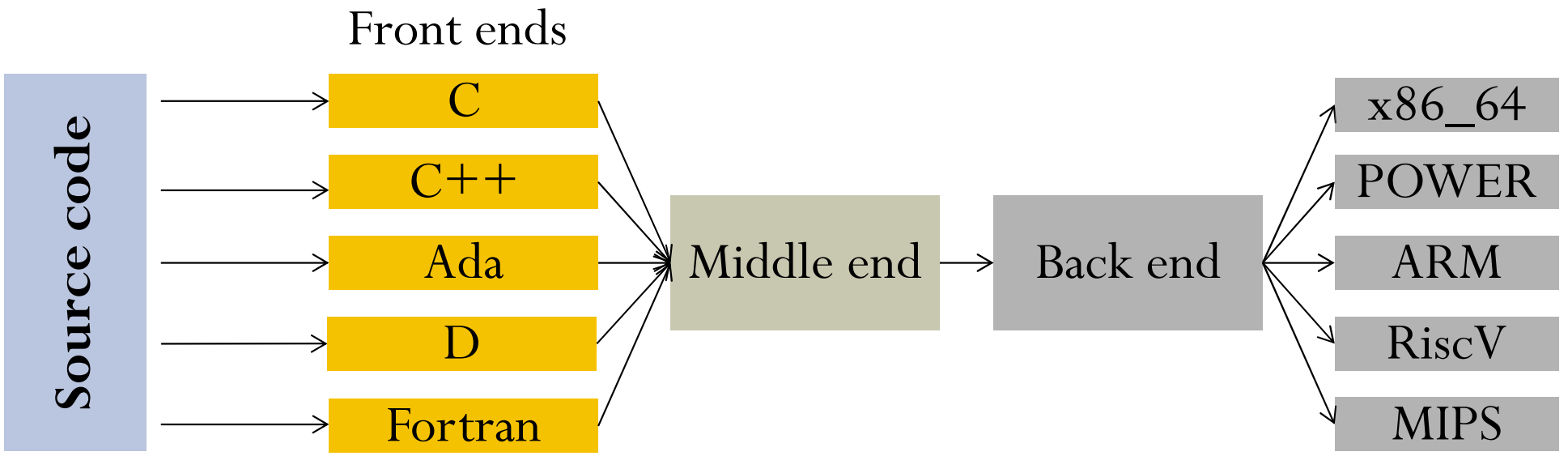
Three-stage compiler design

Standard for optimizing compilers

Parsing,
Intermediate language
generation

Optimization

Machine code
generation (+ some
optimization)



Why optimize in the front end?

Because it knows things the other parts do not

- The middle end knows nothing about the specific languages
- The middle end operates mostly on Single Static Assignment (SSA)
- Three-operand statements like `_13 = _14 + _15`
- One variable is split into multiple variables, but is only assigned once → really good for tracking and propagation of variables
- Arrays are (mostly) unknown

→ There are aspects in the Fortran language that are better optimized in the front end

Looking over the front end's shoulder

`-fdump-fortran-original` to get the syntax tree

```
program main
   integer, dimension(10) :: a
   integer :: b
   read (*,*) a
   b = maxval (a) + 1
   print *,"Maximum + 1 is ", b
end program main
```

```
READ UNIT=5 FMT=-1
TRANSFER main:a(FULL)
DT_END
ASSIGN main:b (+ _gfortran_maxval_r4[[((main:a(FULL)) ((arg not-present))
                                      ((arg not-present)))]] 1.00000000)
WRITE UNIT=6 FMT=-1
TRANSFER 'Maximum + 1 is '
TRANSFER main:b
DT_END
```

You will also see the transformations done by `-ffrontend-optimize`

Looking over the front end's shoulder

`-fdump-tree-original` to get what the front end produces

```
b = maxval (a) + 1
```



```
{
  integer(kind=4) limit.2;
  limit.2 = -2147483648;
  {
    integer(kind=8) S.3;
    S.3 = 1;
    while (1)
      {
        if (S.3 > 10) goto L.1;
        limit.2 = MAX_EXPR <a[S.3 + -1], limit.2>;
        S.3 = S.3 + 1;
      }
    L.1:;
  }
  b = limit.2 + 1;
}
```

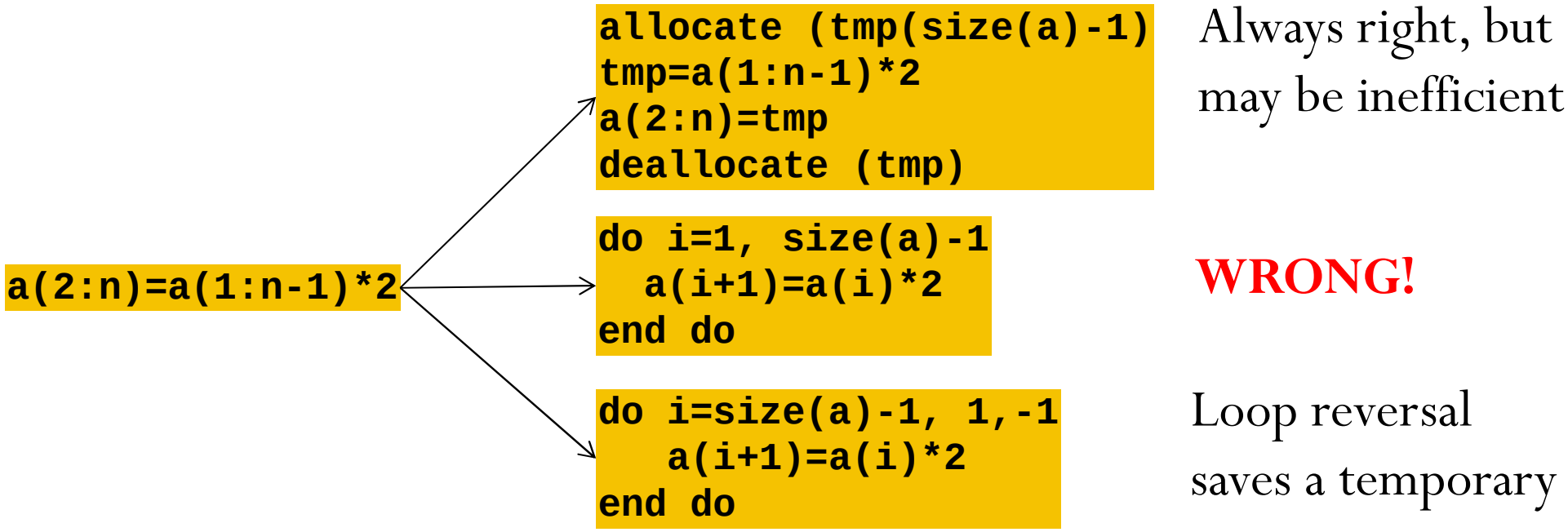
Dependency analysis for array expressions

Array temporaries can be a major source of inefficiency

Fortran language requires evaluation of the right hand side before assignment to the left hand side

→ also for arrays!

→ generating an array temporary is always right, but not always efficient or needed



Recognized cases for array dependency analysis

If performance is important, use `-Warray-temporaries` to check

- Different variables can not alias (unless in EQUIVALENCE)
- Ranges of variables do not overlap: `a(i+1:2*i)=a(1:i)*2` $i+1 > i$
- Ranges of do not intersect: `a(1:n:6)=a(2:m:9)*2` $\mathrm{mod}(2-1, \gcd(9,6)) \neq 0$
- Different components: `a%re = - a%im`
- Loop can be reversed: `a(m:n)=a(k:n-1)*2` $n-1 < n$
- Forward loop: `a(k:n-1)=a(m:n)`

It is easy to fool the analysis, though – this will generate a temporary:

```
j = 2
a(i+j:n+j)=a(i:n)*2
```

Source-code transformation

- Rewrites the Fortran source code to what the user should have written (or what the compiler writer thinks the user should have written)
- Is enabled by `-ffrontend-optimize`, disabled by `-fnofrontend-optimize`
- -O implies –ffrontend-optimize, unless specifically disabled

Make ANY, ALL and Friends More Efficient

Convenient shorthands should be efficient, too

Frequent shorthand: `if (any ([r1, r2, r3] > 0.4)) then`

is converted to `if (r1 > 0.4 .or. r2 > 0.4 .or. r3 > 0.4) then`

Similar transformations for ALL, SUM and PRODUCT

Currently, there is a limit of 4 items in a constructor

Character Variable Substitutions

Fixed-length characters can be optimized

Original	Replacement	Reason
`a = ‘ ‘`	`a = ‘‘`	Replaces memcpy by memset
`trim(a)`	`a(1:len_trim(a))`	Saves a temporary string
`A//B > A//C`	`B>C`	Saves a temporary and a comparison
`B//A>C//A`	`B>C`	Saves a temporary and a comparison

Translating I/O implied loops to array accesses

Optimization rewrites implied do loops to direct array access

```
subroutine foo(a,n)
  real :: a(i)
  integer :: i
  write (*,*) (a(i),i=1,n)
end
```

```
WRITE UNIT=6 FMT=-1
DO foo:i=1 foo:n 1
  TRANSFER foo:a(foo:i)
END DO
DT_END
```

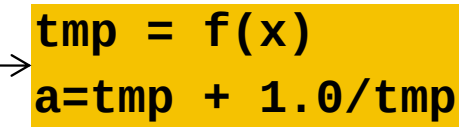
```
subroutine foo(a,n)
  real :: a(i)
  integer :: i
  write (*,*) a(1:n)
end
```

```
WRITE UNIT=6 FMT=-1
TRANSFER foo:a(1:foo:n:1)
DT_END
```

Common Function Elimination

In Fortran, functions exist to return a value (unlike in C)

```
a=f(x)+1.0/f(x)
```



```
tmp = f(x)
a=tmp + 1.0/tmp
```

According to the Fortran standard, functions need only be evaluated once if the value of the expression can be determined otherwise

- Possibility of replacing them with temporary variables
- Also works for array functions like matmul

Currently restricted to PURE functions by default when optimizing

`-faggressive-function-elimination`

sets this option also for non-PURE functions

F2018, 10.1.7:

It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise.

If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference.

Loop Interchange: FORALL, DO CONCURRENT

When you cannot specify it clearly

Everybody should know how to sort DO loops:

```
DO K=1, L
  DO J=1, M
    DO I=1, N
      C(I,J,K) = A(I,J,K)+B(I,J,K)
    END DO
  END DO
END DO
```

- consecutive memory accesses are fastest due to cache and prefetch
- Fortran specifies that first index has consecutive access

But what about

```
DO CONCURRENT (K=1:N, J=1:N I=1:N)
  C(I,J,K) = A(I,J,K)+B(I,J,K)
END DO
```

- The compiler can chose any order of looping
- If it chooses badly, code can be **very** slow
- The best ordering between compilers can also vary

→ Some sort of control is needed

Loop interchange for forall and do concurrent

If you know better, turn of with `-fno-frontend-loop-interchange`

Original code

```
DO CONCURRENT (K=1:N,I=1:N,J=1:N)
  C(I,J,K) = A(J,I,K)+B(I,J,K)
END DO
```

- Count the number of occurrences of each index variable in the subscripts
- Sort so the one with the rightmost access goes to the outermost code

Index	Pos 1	Pos 2	Pos 3	Loop position
J	1	2	0	Middle
I	2	1	0	Inner
K	0	0	3	Outer

"As if" resulting code

```
DO K=1, N
  DO J=1, N
    DO I=1, N
      C(I,J,K) = A(J,I,K)+B(I,J,K)
    END DO
  END DO
END DO
```

Inline Matrix Multiplication

Small matrices can gain from inlining

Why inline matrix multiplication?

- The library routine is fairly efficient
- External BLAS implementations can be called with `-fexternal-blas` (which will also handle TRANSPOSE and CONJG)

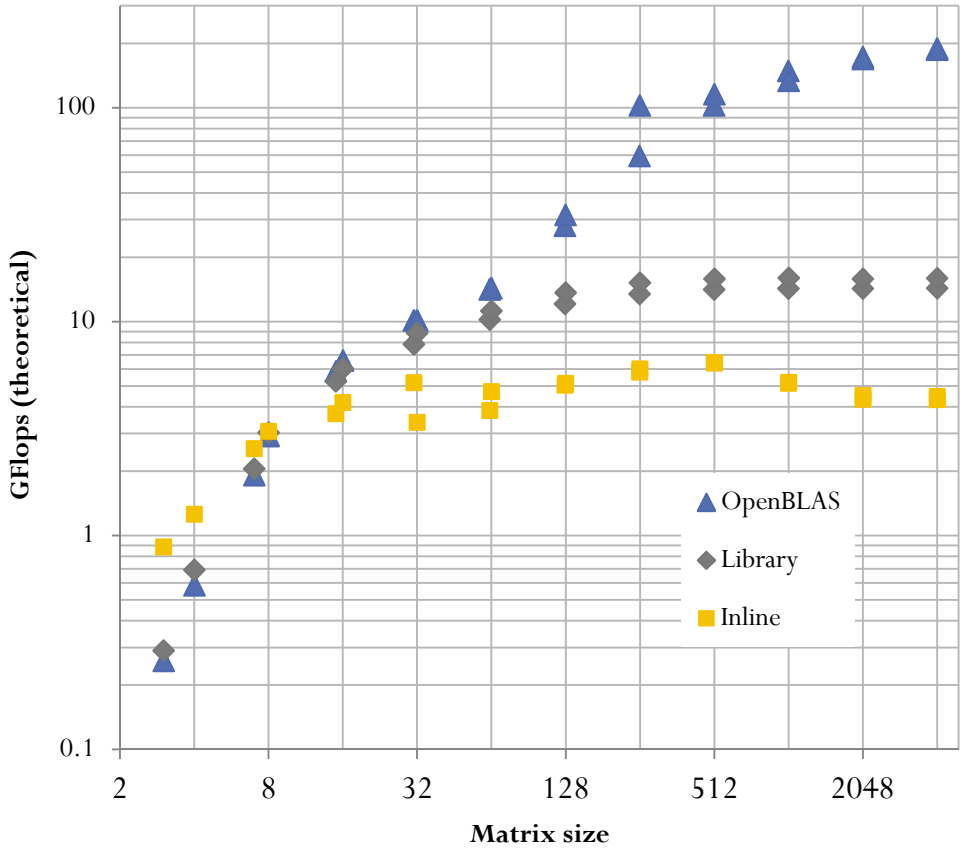
The answer: In small matrices!

TRANSPOSE and CONJG are handled, too

If matmul performance is important

a) use -`fxeternal-blas`

b) tune via `-fblas-matmul-limit`



Summary and Outlook

What's next?

- gfortran does a lot of small optimizations "under the hood" so the programmer does not have to
- If you are interested, you can dump the intermediate representations to see what's going on
- Try –fdump-fortran-original and –fdump-tree-original to see what the compiler is doing
- Inlining matmul for small sizes can have a big impact – benchmark and tune

What's next? There are always ideas

- Fixing some more character handling
- Efficiently calculating second-order difference quotients via

```
CSHIFT (A,1)-2*A+CSHIFT(A,-1)
```

So, what if you have a cool idea?

- Discuss it here
- Mail to fortran@gcc.gnu.org
- Submit a PR in gcc's bugzilla
- **Implement it yourself!**