Modernizing legacy codes

Youngsung Kim,
Srinath Vadlamani,
Benjamin Jamroz,
and John Dennis

ASAP/TDD/CISL/NCAR

Workflow and tools
We’ve learned some about

- how to get performance from new processors
Can we apply what we’ve learned immediately?

- to large legacy codes

NOT REALLY...
What makes it hard to apply?

- Key data structures are used in many parts of codes
  - One change in a key data structure causes to rewrite considerable parts of entire codes
- We need experts who know well across multiple disciplines
  - Programming skill on a specific domain
  - Performance optimization skill on possibly new processors
  - Also have time and motivation for this work
- And the size of code also matters
  - Long cycle time for debugging
  - ex) CESM: 1.5 mil. lines
How to overcome the difficulties?

- Split entire work into smaller tasks
  - to reduce complexity of work
  - may not need for “Ninja-level” expert
- Parallelize tasks
  - to utilize the power of parallelization
  - A practical way to deal with the size of legacy code
- Automate the process as much as possible
  - Kernel generation
  - Optimization
  - Verification
Workflow for refactoring legacy code

Legacy Code

Kernel Generation
(ex: KGEN*)

Stand-alone Kernel

Kernel Optimization
(ex: OpenCase**)

Optimized kernel

Performance Analysis
(ex: Extrae/Paraver)

Verification: Bit-for-bit, RMS error check, Legacy code-own verification, or domain-specific methods

KGEN*: Semi-automated Kernel Generator from NCAR
OpenCase**: User-directed Performance Optimizer from author’s Ph.D Research
Performance Analysis

● Primary Goals
  ○ To choose parts of codes to be extracted
  ○ To get an idea how to optimize the extracted codes

● Profilers/Tracers
  ○ There are many commercial and non-commercial tools for performance analysis
  ○ Personal experience and preference could be major factors in choosing a tool among others.
  ○ NCAR found that Extrae/Parver from Barcelona Supercomputing Center is useful on investigating very short period of execution with low-overhead.
Folding Analysis
How many resource stalls happen per a CPU cycle during executing a subroutine that lasts around 10 ms.
What lines of source code cause the peaks of resource stalls?
Optimize the lines identified from previous step.

!DEC$ vector always
do i = 1, ncol
  extinct(i,k) = extinct(i,k) + dopaer(i)*air_density(i,k)/mass(i,k)
  absorb(i,k) = absorb(i,k) + pabs(i)*air_density(i,k)
  ... 
end do

!DEC$ vector always
do i = 1, ncol
  if (wetvol(i) > 1.e-40_r8) then
    dustaodmode(i) = dustaodmode(i) + dopaer(i)*dustvol(i)/wetvol(i)
    ... 
  end if
end do

!DEC$ vector always
do i = 1, ncol
  if (wetvol(i) > 1.e-40_r8) then
    sumscat(i) = scatso4(i) + scatpom(i) + scatsoa(i) + scatbc(i) + scatdust(i) + scatseasalt(i) + scath2o(i)
    ... 
  end if
end do

! repeats loop split in similar way 
...
Verify that the modification improves the metrics
**KGEN**
Semi-automated Kernel Generator

We want extract a function (funcA) from a.f90

```
>> kgen <kernel source file>:<kernel name> <state generation source file>:<line number>
```

Example: `>> kgen a.f90:funcA b.f90:10`

![Diagram of the KGEN process]

- **Legacy Code**
- **Modified Legacy Code**
- **Kernel Code**
- **Kernel Executable**
- **State data**
- **Build & Run**

**Bit-for-bit and RMS error verification**
Ex: PORT (a stand-alone code of CESM rrtmg)

“binterp” subroutine identified as one of hotspots in PORT using Extrae/Paraver

KERNEL_FILE := modal_aer_opt.F90
KERNEL_FUNCTION := binterp

kgen -p include.ini -i user.ini -n 1 -m 0,1,2 ${KERNEL_FILE}:${KERNEL_FUNCTION} ${KERNEL_FILE}:639
KGEN
Ex: PORT - Continue

modal_aer_opt.F90

subroutine modal_aero_sw(list_idx, state, pbuf, nnite, idxnite, tauxar, wa, ga, fa)

! save input state
WRITE(UNIT = kgen_unit) itab
WRITE(UNIT = kgen_unit) lbound(refitabsw, 1)
WRITE(UNIT = kgen_unit) ubound(refitabsw, 1)
WRITE(UNIT = kgen_unit) lbound(refitabsw, 2)
WRITE(UNIT = kgen_unit) ubound(refitabsw, 2)
WRITE(UNIT = kgen_unit) refitabsw

! call kernel
CALL binterp(extpsw(:, :, :, isw), ncol, ...)

! save output state
WRITE(UNIT = kgen_unit) utab
WRITE(UNIT = kgen_unit) lbound(refitabsw, 1)
WRITE(UNIT = kgen_unit) ubound(refitabsw, 1)
WRITE(UNIT = kgen_unit) lbound(refitabsw, 2)
WRITE(UNIT = kgen_unit) ubound(refitabsw, 2)
WRITE(UNIT = kgen_unit) refitabsw

end subroutine

kernel_binterp.F90

program kernel_binterp

! type declaration statements
INTEGER :: itab(pcols)
REAL(KIND = r8) :: utab(pcols)
REAL(KIND = r8), POINTER :: refitabsw(:, :)

! read state
READ(UNIT = kgen_unit) utab
READ(UNIT = kgen_unit) kgen_bound(1, 1)
READ(UNIT = kgen_unit) kgen_bound(2, 1)
READ(UNIT = kgen_unit) kgen_bound(1, 2)
READ(UNIT = kgen_unit) kgen_bound(2, 2)
ALLOCATE(refitabsw(kgen_bound(2, 1) - kgen_bound(1, 1) + 1,
kgen_bound(2, 2) - kgen_bound(1, 2) + 1))
READ(UNIT = kgen_unit) refitabsw

! call kernel
CALL binterp(extpsw(:, :, :, isw), ncol, ...)

! verify output from a generated kernel
IF ( ALL(outstate_itab == itab) ) THEN
  WRITE(*,*) "itab is IDENTICAL."
ELSE
  WRITE(*,*) "itab is NOT IDENTICAL."
  WRITE(*,*) "RMS of difference is ", &
  sqrt(sum((outstate_itab - itab)**2)/real(size(outstate_itab)))
... end program
KGEN

Current status

● Overview
  ○ Written in Python
  ○ Extension of Fortran parser distributed in F2PY python package

● Semi-automated approach
  ○ KGEN tries to generate kernel fully automated way
  ○ If it fails, user provide required information through familiar INI configuration file
    ■ Fortran specification, files to search first, names to ignore, etc.

● Current limitations
  ○ Derived type having allocatable or pointer component is not supported
  ○ Array of derived types is not supported
OpenCase
OpenCase
User-directed performance optimizer

- Programming techniques for manual optimization
  - Source modification
    - loop unrolling, loop splitting, loop merge, data structure refactoring…
  - Compiler options
    - alignment options, binary generation options, vectorization options, prefetching options, …
  - Intrinsic or assembly modification
- Can we mimic what programmer manually does in more automated way?
  - Try every combinations of the techniques
OpenCase
ex) divergence_sphere() from HOMME

```fortran
subroutine divergence_sphere()

100    do j=1,np
200       do i=1,np
300          gv(i,j,1)=elem_metdet(i,j)*(elem_DinvC(i,j,1,1)*v(i,j,1) + elem_DinvC(i,j,1,2)*v(i,j,2))
400          gv(i,j,2)=elem_metdet(i,j)*(elem_DinvC(i,j,2,1)*v(i,j,1) + elem_DinvC(i,j,2,2)*v(i,j,2))
500       enddo
600    enddo

! compute d/dx and d/dy
700    do j=1,np
800       do l=1,np
900          dudx00=0.0d0
1000         dvdy00=0.0d0
1100       do i=1,np
1200             dudx00 = dudx00 + deriv_Dvv(i,l  )*gv(i,j  ,1)
1300             dvdy00 = dvdy00 + deriv_Dvv(i,l  )*gv(j  ,i,2)
1400       end do
1500       kgen_div(l  ,j  ) = dudx00
1600       vvtemp(j  ,l  ) = dvdy00
1700    end do
1800    end do

1900    do j=1,np
2000       do i=1,np
2100           kgen_div(i,j)=(kgen_div(i,j)+vvtemp(i,j))*(elem_rmetdet(i,j)*rrearth)
2200       enddo
2300    enddo
```
OpenCase
Case generation

- Environmental variables
  - KMP_AFFINITY=(balanced; compact)

- Compiler Options
  - (-O2; -O3)
  - {-align array64byte, -opt-prefetch=(0;3), '-no-prec-sqrt', '-no-prec-div'}*

- Source transformation
  - Loop unrolling, merge, split, interchange
  - Line insert/delete
  - ...

=> 356,483,072 cases
OpenCase

**Speed-ups during 6-hour run**

The best speed-ups of divergence_sphere during 6-hour run
(Completed 0.4% out of 356,483,072 cases)
subroutine divergence_sphere()

100    do j=1,np
!DEC$ VECTOR ALWAYS ALIGNED
200    DO i=1,np,1
300        gv(i,j,1) = elem_metdet(i,j)*(elem_dinvc(i,j,1,1)*v(i,j,1) + elem_dinvc(i,j,1,2)*v(i,j,2))
END DO
!DEC$ VECTOR ALWAYS ALIGNED
DO i=1,np,1
400        gv(i,j,2) = elem_metdet(i,j)*(elem_dinvc(i,j,2,1)*v(i,j,1) + elem_dinvc(i,j,2,2)*v(i,j,2))
600    enddo

! compute d/dx and d/dy
!DEC$ VECTOR ALWAYS ALIGNED
!DEC$ UNROLL (np)
800    do i=1,np
700        do j=1,np
900            dudx00=0.0d0
1000           dvdy00=0.0d0
i = 1
    dudx00 = dudx00 + deriv_dvv(i, l) * gv(i, j, 1)
    dvdy00 = dvdy00 + deriv_dvv(i, l) * gv(j, i, 2)
    i = i + 1
    dudx00 = dudx00 + deriv_dvv(i, l) * gv(i, j, 1)
    dvdy00 = dvdy00 + deriv_dvv(i, l) * gv(j, i, 2)
i = i + 1
    ...
OpenCase - Features

- **Source code transformation**
  - loop transformation: unrolling, split, merge, interchange
  - primitive: line insertion/deletion
  - array: index swap

- **Compiler flags**
  - any arrangement of compiler flags

- **Performance measurement and verification**
  - Measure performance result and sort
  - Measure output and verify against tolerance or other factors

- **Case generation algorithm**
  - current research topic
Acknowledgement

KGEN development is a part of The Intel® Parallel Computing Center for Weather and Climate Simulation (IPCC-WACS) funded by Intel Parallel Computing Center.

NCAR, intel, Boulder
Thank YOU

Q & A