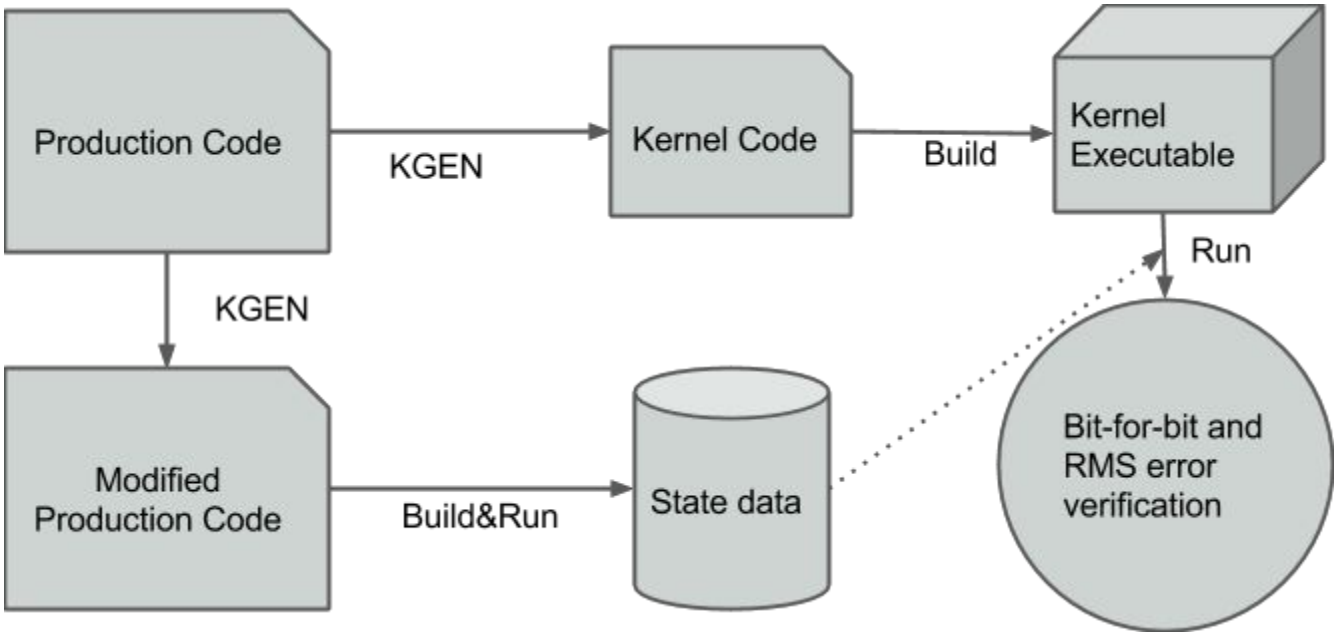
KGEN User's Guide for KGEN ver. 0.4.7

0. Introduction

KGEN is a software tool that extracts a Fortran subprogram from a larger software and make it as a stand-alone application, aka kernel. It also saves input data to and output data from the extracted part of source code that are used in the kernel for verification purpose.

Generic workflow using KGEN is shown in following figure. User executes KGEN in command line with arguments that specify which subprogram to be extracted and where is a call site to the subprogram. As an initial step of kernel generation, KGEN creates kernel files as well as modified source file(s). The modified source files are used to generate input & output data. User replaces original source files with these modified source files and run the modified production code. Once input & output data are generated successfully, a kernel can be built and executed as a stand-alone application and its outcome from execution is automatically verified against the generated input & output data.



In practice, user may adopt iterative workflow in using KGEN. User first provides KGEN with minimum information and tries to generate a kernel. If it fails, in return, KGEN may provide an user with what information is missed. In addition, KGEN tries to create a kernel even though it can not find all required information and user may investigate the generated kernel further on missed information. Then user may re-try KGEN by providing additional information, and repeat this workflow.

KGEN is being developed actively as of writing this document. It would be very helpful if you can share your experience on using KGEN by sending relevant information to “kgen@ucar.edu”.

1. How to use

1.1 Installation

KGEN is written in Python as an extension of a Fortran parser distributed in F2PY python package.

1.1.1 Requirements

KGEN is developed mainly in Python 2.6.6, which is a minimum required for using KGEN. A Fortran parser of F2PY is extensively used in KGEN and is a part of KGEN distribution. Therefore, there is no additional work needed for installing the F2PY Fortran Parser.

Once you have unzipped KGEN distribution file, then you can run KGEN as below:

python $PATH_TO_KGEN/kgen.py --help

If you are an user of Yellowstone cluster of NCAR, you can use a pre-installed KGEN at “/glade/u/tdd/asap/contrib/kgen/std”

KGEN requires either Fortran preprocessor (“fpp”) or C preprocessor(“cpp”). Currently fpp is default.

1.2 User interface

KGEN will support two ways of user-interface: command-line and KGEN directive.

KGEN directives are inserted in source codes to directs what KGEN to do. Command-line inputs overwrites KGEN directives.

1.2.1 KGEN directive user interface

In general, the syntax of KGEN directive follows OpenMP syntax. Case is not sensitive if not specified.

SYNTAX: !$kgen directive name/clause(s)

directive specifies the type of setting. name or clause(s) is additional information for the directive. Name is a non-space ascii string and clause is a name with having parenthesis.

Continuation of kgen directive is supported as follow:

!$kgen directive name/clause(s) &
!$kgen& [additional name/clause(s)]

“callsite” is the only KGEN directive implemented in this version.

Callsite directive

syntax: callsite name

meaning: callsite directive specifies the location of callsite Fortran statement in a source file. If this directive is specified in source file, user does not have to provide “namepath” after colon on command-line. “namepath” is a dot-seperated names. Please look at command line user interface for details

The directive should be placed just before the callsite line. However, blank line(s) and other comment lines are allowed in-between. name is the subroutine or function name to extract as a kernel.

example:

```
!$kgen callsite calc
CALL calc(i, j, output)
```

1.2.2 KGEN command-line user interface

Once you have KGEN files in an accessible directory, you can run generate a kernel using KGEN by issuing following command without adding KGEN directive in a source file.

>> python $PATH_TO_KGEN/kgen.py [KGEN flags] <filepath>:<callsite identifier>

“filepath” is a path to a source file that contains a call to the kernel subprogram to be extracted.” “callsite identifier” tells KGEN which Fortran line is the callsite. The simplest way to set “callsite identifier” is to use the name of subroutine or function name. For example, if the callsite is “CALL calc(i, j, output)”, then “callsite identifier” is “calc”. If there is name conflict, user can specify the hierarchy of name space, which is explained in section 1.4.1.

1.3 Example of generating “calc” kernel

In this section, an example of generating a kernel of “calc” from “calc_mod.F90” source file is explained. This example shows a basic workflow of using KGEN with information on using essential KGEN flags. You can find this example in “calc” folder of KGEN distribution.

program.F90

```
PROGRAM demo
  USE update_mod, &
      only : update
  INTEGER t
  DO t=1,10
    CALL update
  END DO
END PROGRAM
```

update_mod.F90

```
MODULE update_mod
  USE calc_mod, only : calc
  PUBLIC update
CONTAINS
  SUBROUTINE update()
    INTEGER :: i, j
    INTEGER :: output(COL,ROW)
    DO i=1,COL
      DO j=1,ROW
        !$kgen callsite calc
        CALL calc(i, j, output)
      END DO
    END DO
  END SUBROUTINE
END MODULE
```

calc_mod.F90

```
MODULE calc_mod
  PUBLIC calc
CONTAINS
  SUBROUTINE calc(i, j, output)
    INTEGER, INTENT(IN) :: i, j
    INTEGER, INTENT(OUT), &
        dimension(:,:) :: output
    output(i,j) = i + j
  END SUBROUTINE
END MODULE
```

In this example callsite is “CALL calc(i, j, output)” in update_mod.F90. Callsite should be placed in a subprogram of a module.

To extract “calc” subroutine and make it stand-alone application, following KGEN command is used.

```
>> python ${KGEN} \
        -D ROW=4,COL=4 \
        -I ${SRC_DIR} \
        --kernel-compile FC=ifort,FC_FLAGS='-O3' \
        --state-build cmds="cd ${SRC_DIR}; make build" \
        --state-run cmds="cd ${SRC_DIR}; make run" \
        ${SRC_DIR}/update_mod.F90
```

Each lines of the command are explained below.

“**>> python ${KGEN}** \” : This invokes KGEN. It is assumed that ${KGEN} has valid path to “kgen.py” Python source file in KGEN distribution. KGEN is developed using Python 2.6.6.

“-D ROW=4,COL=4 \” : “-D” KGEN flag specifies the macro definitions that are used for preprocessing Fortran source files. In this example, “ROW” is defined as “4” and “COL” is defined as “4”. This information will be used by “fpp” or “cpp” preprocessors before KGEN starts to parse input source files.

“-I ${SRC_DIR} \” : “-I” KGEN flag specifies the directories for KGEN to search Fortran source files. In this example, it is assumed that ${SRC_DIR} has a valid path to a directory that contains all of three input Fortran source files, “program.F90”, “update_mod.F90” and “calc_mod.F90”

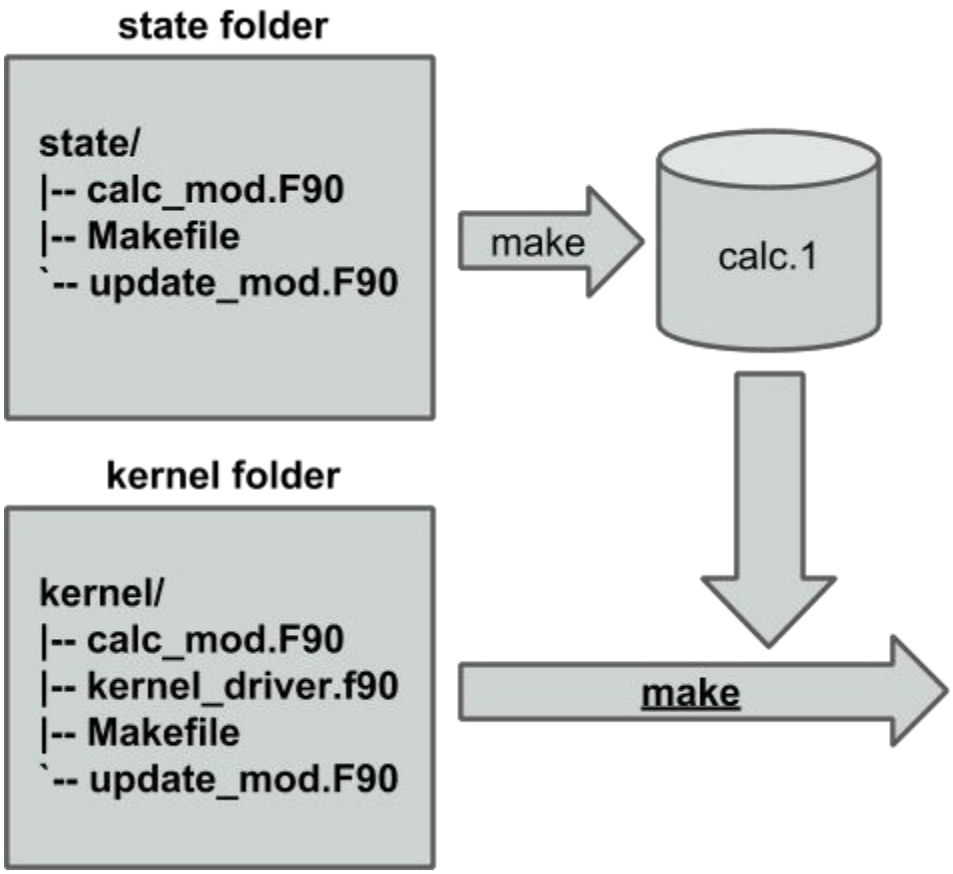
“--kernel-compile FC=ifort,FC_FLAGS='-O3' \”: “--kernel-compile” KGEN flag provides the Fortran compiler and compiler flags to compile generated kernel. It is prefered to use the same compiler and compiler flags to original production compilation.

“--state-build cmds="cd ${SRC_DIR}; make build" \” : “--state-build” KGEN flag provides command how to build instrumented original production application. KGEN will generate Makefile that contains the content of “cmds” sub-flag as a part of building task. In this example, it is assumed that Makefile is provided in original application.

“--state-run cmds="cd ${SRC_DIR}; make run" \” : “--state-run” KGEN flag provides command how to run the instrumented original production application. KGEN will generate Makefile that contains the content of “cmds” sub-flag as a part of execution task. In this example, it is assumed that Makefile is provided in original application.

“**${SRC_DIR}/update_mod.F90**” : This line specifies the path to a source file that contains “callsite”. In this example, the exact location of callsite is specified by KGEN “callsite” directive in source file.

Once KGEN completes kernel generation successfully, two subfolders will be created in current working directory as shown below



Next step is to generate data from executing original production code with replacing some of files with KGEN-generated files in “state” folder. This process is automated by Makefile in the “state” folder. Run “make” in “state” folder. Once it completes with success, a data file, “calc.1” in this example, will be created in “kernel” folder.

Final step is to build and run the “calc” kernel. Move to “kernel” folder and run “make”. If it completes with success, you will see something similar to following figure on screen.

```
>> make
ifort -O3 -c -o calc_mod.o calc_mod.F90
ifort -O3 -c -o update_mod.o update_mod.F90
ifort -O3 -c -o kernel_driver.o kernel_driver.f90
ifort -O3 -o kernel.exe kernel_driver.o
  update_mod.o calc_mod.o

./kernel.exe

** Verification against 'calc.1' **

 calc: Tolerance for normalized RMS:
9.9999998245167OOE-015
 calc: Number of variables checked:          1
 calc: Number of Identical results:          1
 calc: Number of warnings detected:          0
 calc: Number of fatal errors detected:      0
 calc: verification PASSED

 Elapsed time (sec):   1.0000000E-07
```

This shows that the kernel is compiled with success and the correctness check shows that kernel generated data is identical to data generated from original production application. It also shows that the kernel execution took 1.7E-7 seconds.

1.4 Supporting kernel generation for large-sized software

As we are interested in large-sized code, it is likely that the software adopts modular approaches with multiple source files. KGEN can correctly analyze source codes only when all required paths for searching modules and correct set of macro definitions are provided.

1.4.1 Namepath

To address possible name conflict among different levels of namespace, KGEN introduces a hierarchical representation of a name, called “namepath.”

Namepath is consecutive names with a period between them as a separator. For example, if name B is inside of A, the it can be represented by “A.B”. In practice, it is used to specify “callsite identifier” in KGEN. callsite identifier is a namepath that locates “callsite” to kernel.

an example of callsite identifier: “F90:module_A.function_B.subroutine_C”

In above example, user specified a Fortran line that contains “subroutine_C”, whose upper namespace is “function_B”, and again one more upper level is “module_A”.

If there is no name conflict, for example that there is only one “subroutine_C” exists in the given file, user can only specify “subroutine_C” without upper-level names.

1.4.2 User-provided inclusion information

KGEN accepts an INI-format file with “-i” command-line option. In the INI file, user can provide KGEN with information of macro-definitions and include directories. Details of using the INI file are explained in following sections.

Command line option format: “-i <user-providing INI format file>”

Syntax of the INI file follows conventional INI file syntax. Brackets are used to specify sections. In a section, a option is added in a line or over multiple lines. Each option has a format of key and value pair with a separator of either “:” or “=”. Value part can be missed.

1.4.2.1 Information for each source file

Some information has to be provided per each source file separately. As of this version, there are two types of information are identified in this category: macro definition and include directory. Following convention is used to provide these information in the INI file.

```
[ Path-to-source-file ]
include = [directory path]:[directory path]:...
macro_name = macro_value
...
```

example) When “program.F90” uses a module in “./module” directory, and “program.F90” needs macro definition of “N=10”

```
[program.F90]
```

include = ./module
N = 10

There can be multiple macro_name options but only one include option is allowed per each file.

By providing these options, user allows KGEN to analyze source code correctly. In addition, KGEN will analyze files specified in this INI file first, which may help to reduce analysis time.

1.4.2.2 Common information to all source files

There are several types of information that can be applied to all source files that KGEN analyzes.

Common macro definitions and include directories

In some cases, all source files may share the same macro definitions and/or include directories. In the cases, instead of specifying the information per each source file sections, user can use following sections.

[macro]
macro_name = macro_value
…

These macros will be added to each source file during KGEN analysis.

[include]
include_path1 =
include_path2 =
…

These include_paths will be added to each source file during KGEN analysis. Note that each path should be specified per each line, which is different from the syntax of separate section for each file. Value part of each option should be blank for this version and is reserved for later use.

1.4.3 User-provided exclusion information

KGEN accepts an INI-format file with “-e” command-line option. In the INI file, user can provide KGEN with information of names to be excluded during name search. Details of using the INI file are explained in following sections.

Command line option format: “-e <user-providing INI format file>”

Syntax of the INI file follows conventional INI file syntax. Brackets are used to specify sections. In a section, a option is added in a line or over multiple lines.

1.4.3.1 Common names that apply to all source files

When “common” is specified in a section of INI file, the options in the section will be applied to all source files.

[common]
name = action

“name” in above example will be excluded during name search and “action” will be applied when KGEN finds the name.

There is only one “action” is defined in this version: “comment.” When “comment” is specified as “action”, KGEN will comment-out the statement.

1.4.3.1 Specific names that apply to a particular name-path

“name-path” in this INI file has the same format when user specifies name-path for call site in command line

[path.to.name]
name = action

This provides KGEN with the same information to what “common” section does, but this only applies to a particular name under the “path.to.name”

1.5 Command line options

The syntax of each options generally follows the following convention:

General KGEN option syntax:
-[-]<option-name> [<suboption-name>=<suboption-value>[,[<suboption-name>=]<suboption-value>]]

If there are multiple information in <suboption-value>, each information would be separated by colon, “:”. Double or single quotation marks can be used to use some of the separation symbols, such as equal sign, comma, colon, in option value.

[-i, --include]

meaning: provides information of what to be included during KGEN analysis. Information includes macro definitions and include directories.
example) -i include.ini

[-I]
meaning: provides include path information where KGEN search files. This information applies to all source files
example) -I /path/to/directory/a:/path/to/directory

[-D]
meaning: provides macro definitions. This information applies to all source files
example) -D USE_A=1

[--ordinal-numbers]
meaning : specifies Nth kernel invocation to save input/output data
example) --ordinal-numbers 1,10,100

[--mpi]
meaning : Turns on MPI supports in KGEN. There are several sub-options: “ranks”, “comm”, “use”, and “header”. “ranks” specifies the MPI ranks on which data is collected. Default “ranks” is 0. “comm” specified the names of variable that is used when MPI call is made. Default “comm” is “MPI_COMM_WORLD”. “use” specifies Fortran module name whose name is inserted in additional Fortran use statement. There is no default value for “use”. “header” specifies the path to MPI header file. Default “header” is “mpif.h”.
example) --mpi ranks=0,comm=mpicom,use="spmd_utils:mpicom"

[--[no]skip-intrinsic]
meaning : flags to let KGEN skip searching for names of intrinsic-procedures. There is one sub-option, “except”. With “except”, user can specify which “namepath” should be considered as exception. default: --skip-intrinsic
example) --skip-intrinsic except=mod_A.subr_B.sum

[--kernel-compile]
meaning : compile-specific information used in generating Makefile for kernel. Two sub-flags are defined in this version: “FC” and “FC_FLAGS”. User can choose which Fortran compiler to be used in the kernel makefile with “FC” flag..
example) --kernel-compiler FC=ifort,FC_FLAGS=-O3

[--kernel-link]
meaning : compile-specific information used in linking-phase. Three sub-flags are defined in this version: “pre_cmds”, “include”, and “lib”. Values of “pre_cmds” sub- flags will be executed before the generated Makefile builds the kernel. “include” sub-flags provides include paths for libraries.

The paths of this flag will be added after “-L” compiler flag. “lib” sub-flags provides the name of libraries. The names of this flag will be added after “-l” compiler flag.
example) --kernel-link include=”./”,lib=mylib

[--state-build]
meaning : build-specific information used in generating Makefile for original program. One sub-flags are defined in this version: cmds. User can run Linux command specified in cmds.
example) --state-build cmds=”cd ${WORK_DIR}; module load mkl; make”

[--state-run]
meaning : execution-specific information used in generating Makefile for original program. One sub-flags are defined in this version: cmds. User can run Linux command specified in cmds.
example) --state-run cmds=”cd ${WORK_DIR}; make run”

[--state-switch]
meaning : provides information how to use KGEN-generated instrumented source files during compiling original production code. Two sub-flags are defined in this version: “type” and “cmds”. Two string values, “replace” and “copy” are valid for “type” sub-flag. “replace” let KGEN replace the files in original production files with instrumented files. “copy” let KGEN copies instrumented files into a folder. When “copy” is used, the other sub-flag, “cmds” should be also provided that actually copy the instrumented files a folder.
example) --state-switch type=copy,cmds=”cd ${WORK_DIR}; rm -f ./*; cp -f ${GEN_DIR}/*.F90 ./”

[--check]
meaning : provides information related to correctness check. Two sub-flags are defined in this version: “pert_invar” and “pert_lim”. “pert_invar” provides name to support perturbation test. “pert_lim” provides maximum range of perturbation value.
example) --check pert_lim=var_a:var_b,pert_lim=1.D-15

[--outdir]
meaning : KGEN output directory
example) --outdir /path/to/output/directory

[--debug]
meaning: This flag specifies information related to debugging KGEN. One useful option is “printvar”, which let KGEN add codes for displaying the content of specified variables during kernel execution as well as original production code execution with instrumented codes.
example) --debug printvar=var_a,var_b

Appendix 1. MG2 kernel generation

In this appendix, steps for generating MG2 kernels are explained.

If you are an user of Yellowstone of NCAR, you can find the latest release of KGEN in "/glade/u/tdd/asap/contrib/kgen/std". In this example, this directory is referenced as $KGEN_ROOT.

MG2 is a microphysics in CESM that calculates rain and snow precipitation.

Following provides a guideline how to generate MG2 kernel using KGEN version 0.4.7. It is assumed that a CESM case of "FC5-cam5-mg2-SNB" is created with configuration for MG2 physics usage and FC5 compset of CESM.

STEP 1: Generating include.ini file with include paths and macro definitions from CESM atm and csm_share log files

Until automated method is developed, following manual method is required. However, to make it a bit easier for the manual method, a Python script is developed and located in $KGEN_ROOT/src/gen_include.py that extract include paths and macro definitions from CESM log files.

Usage: gen_include.py [-o <output file>] <path to log file>

Log files of CESM are under $CASE_OUTPUT_DIR/bld. First, move to $CASE_OUTPUT_DIR/bld and run "$KGEN_ROOT/bin/gen_include.py -o atm.ini atm.bldlog.XXXXX-XXXXX". Then, it will create atm.ini file in the same directory. Under this directory, move to a subdirectory having csm_share.bldlog.XXXXX-XXXXX and run "$KGEN_ROOT/bin/gen_include.py -o csm_share.ini csm_share.bldlog.XXXXX-XXXXX". Merge two ini files as include.ini and put the merged file in a folder where "genMG2kernel.sh" script is (see at the bottom of this document). You may need to add following lines into the merged file to help KGEN to find "mpif.h" header file. An example of "include.ini" file is at the end of this document.

[include]
/ncar/opt/intel/12.1.0.233/impi/4.0.3.008/intel64/include =

STEP 2: Generating kernel

Set "CAM5_HOME" to the top folder of CAM5 distribution and run "genMG2kernel.sh". If successful, two directories(kernel and state) and one log file will be created under ${OUT_DIR}. Please find explanations on each flags in "Command line options" section.

STEP 3: Generating data

Move to ${OUT_DIR}/state and run “make”. If successful, nine data files will be created in ${OUT_DIR}/kernel. First, this command will copy instrumented files into a folder under “SourceMod” folder and save original copies of the files in “state” folder with “kgen_org” extension, and execute "cd ${CASE_DIR}; ./FC5-cam5-mg2-SNB.clean_build; ./FC5-cam5-mg2-SNB.build" in ${CASE_DIR} as specified by “--state-build” KGEN option and finaly run "cd ${CASE_DIR}; ./FC5-cam5-mg2-SNB.submit" as specified by “--state-run” KGEN option.

STEP 4: build and run kernel

Make sure that Intel compiler is usable. Move to ${OUT_DIR}/kernel and run "make". If successful, this command will build and run the kernel and display output message on screen similar to:

```
** Verification against './micro_mg_tend2_0.50.300' **

micro_mg_tend KGENPrtCheck: Tolerance for normalized RMS:
 9.999999824516700E-015
micro_mg_tend KGENPrtCheck: Number of variables checked:          86
micro_mg_tend KGENPrtCheck: Number of Identical results:          86
micro_mg_tend KGENPrtCheck: Number of warnings detected:           0
micro_mg_tend KGENPrtCheck: Number of fatal errors detected:           0
micro_mg_tend KGENPrtCheck: verification PASSED

Elapsed time (sec):   1.4535000E-03
```

********** genMG2kernel.sh **********

```
#!/bin/bash
#
# kgen_run.sh

KGEN := /glade/u/tdd/asap/contrib/kgen/std/src/kgen.py
CASE_DIR := ${CAM5_HOME}/cime/scripts/FC5-cam5-mg2-SNB
SOURCE_MODS := ${CASE_DIR}/SourceMods/src.cam
SRC := ${CAM5_HOME}/components/cam/src/physics/cam/micro_mg_cam.F90
OUTPUT_DIR := ${HOME}/kgen_mg2

python ${KGEN} \
  -i include.ini \
  --outdir ${OUTPUT_DIR} \
  --ordinal-numbers 10,50,100 \
  --mpi ranks=0:100:300,comm=mpicom,use="spmd_utils:mpicom" \
  --state-switch type=copy,cmds="rm -f ${SOURCE_MODS}/*; cp -f
${OUTPUT_DIR}/state/*.F90 ${SOURCE_MODS}" \
  --state-build cmds="cd ${CASE_DIR}; ./FC5-cam5-mg2-SNB.clean_build;
./FC5-cam5-mg2-SNB.build" \
  --state-run cmds="cd ${CASE_DIR}; ./FC5-cam5-mg2-SNB.submit" \
  --kernel-compile FC=ifort,FC_FLAGS='-no-opt-dynamic-align -fp-model source
-convert big_endian -assume byterecl -ftz -traceback -assume realloc_lhs -xHost -O2' \
  ${SRC}:micro_mg_cam.micro_mg_cam_tend.micro_mg_tend2_0
```

```
********** include.ini for MG2 kernel **********
; NOTE
; - replace <CAM_HOME> with proper path to CAM root folder.
; - If symbolic links are used, each symbolic paths should be specified in 'include' section

[include]
/ncar/opt/intel/12.1.0.233/impi/4.0.3.008/intel64/include =
<CAM_HOME>/components/cam/src/physics/cam =
<CAM_HOME>/cime/share/csm_share/shr =

[macro]
PSUBCOLS = 1
HAVE_F2003_PTR_BND_REMAP = 1
LSMLAT = 1
HORIZ_OPENMP = 1
_PRIM = 1
HAVE_TIMES = 1
_WK_GRAD = 1
CAM = 1
HAVE_MPI = 1
HAVE_SLASHPROC = 1
HAVE_NANOTIME = 1
MAXPATCH_PFT = numpft+1
NC = 4
HAVE_GETTIMEOFDAY = 1
N_RAD_CNST = 30
NP = 4
PLEV = 30
NDEBUG = 1
THREADED_OMP = 1
PLAT = 1
PCNST = 29
NO_R16 = 1
MODAL_AERO = 1
MODAL_AERO_3MODE = 1
FORTRANUNDERSCORE = 1
MCT_INTERFACE = 1
HAVE_COMM_F2C = 1
SPMD = 1
CO2A = 1
BIT64 = 1
CPRINTEL = 1
_MPI = 1
```

PTRN = 1
LSMLON = 1
PTRM = 1
HAVE_VPRINTF = 1
PTRK = 1
LINUX = 1
USE_CONTIGUOUS = contiguous,
PLON = 13826
PCOLS = 16
NUM_COMP_INST_ICE = 1
NUM_COMP_INST_GLC = 1
NUM_COMP_INST_WAV = 1
NUM_COMP_INST_LND = 1
NUM_COMP_INST_OCN = 1
NUM_COMP_INST_ATM = 1
NUM_COMP_INST_ROF = 1