KGEN User's Guide for KGEN ver. 0.6.3

CONTENTS

0. Changes from KGEN ver. 0.6.2

0.1 User Interface

- Python version > 2.7 and < 3.0 is required
- Invocation KGEN option is changed to support MPI rank, OpenMP number and invocations more precisely.
- "enable" sub-option for "--mpi" KGEN flag should be used for kernel extraction from MPI application
- "enable" sub-option for "--openmp" KGEN flag should be used for kernel extraction from MPI application

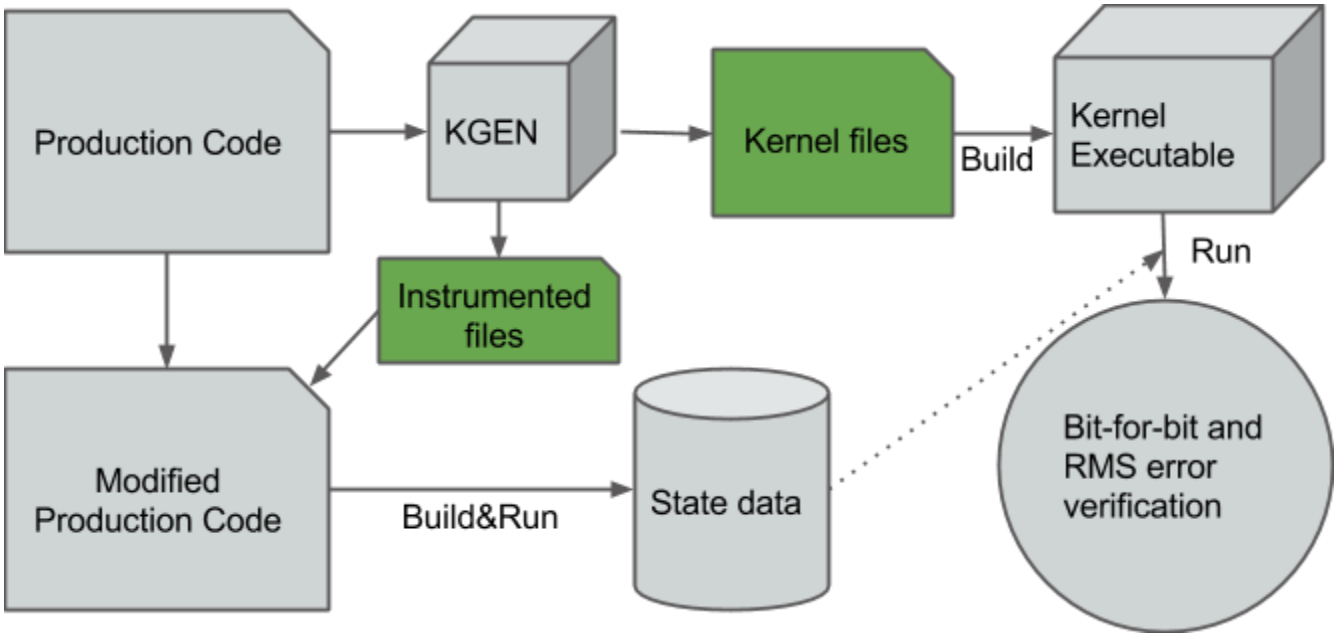
0.2 Major Improvements

- Added support for class type declaration type in state generation
- Added unit tests for GNU and PGI compilers
- Supported detail control for state generation from MPI ranks and OpenMP threads
- Supported Statement Function Statement
- Fixed a but for Sumcheck having NaN entities
- Splitted codebase to KGen core codes and KGen applications
- Refactored to support plugins framework for source-to-source translation on Abstract Syntax Tree

1. Introduction

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

General workflow for using KGEN is shown in following figure. User executes KGEN in command line with arguments that specify which part to be extracted. As an initial step of kernel generation, KGEN creates kernel files as well as modified source file(s) as illustrated as green boxes in the figure. 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 executed as a stand-alone application and its outcome from execution is automatically verified against the generated 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 the steps.

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".

2. How to use

2.1 Installation

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

2.1.1 Requirements

KGEN is developed mainly in Python 2.7, which is a minimum requirement 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:

$PATH_TO_KGEN/bin/kgen --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 C preprocessor("cpp") or Fortran preprocessor ("fpp"). Currently cpp is a default one.

2.2 User interface for specifying a kernel to be extracted

KGEN will support two ways of user-interface for specifying a kernel to be extracted: command-line and KGEN directive.

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

2.2.1 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 action. "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 not supported yet.

Three KGEN directives are implemented as of this version.

"callsite" directive

syntax: !$kgen 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" on command-line. "namepath" is a colon-separated names.

The directive should be placed just before the callsite line. However, blank line(s) and other comment lines are allowed in-between. name is a user-provided string for kernel name.

example:

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

“begin_callsite” and “end_callsite” directives

syntax:

```
!$kgen begin_callsite name
… fortran statements…
!$kgen end_callsite
```

meaning: begin_callsite and end_callsite directives specify a region of Fortran statements in a source file to be extracted as a kernel.

example:

```
!$kgen begin_callsite calc
DO i=1, COL
   DO j=1, ROW
      CALL calc(i, j, output)
   END DO
END DO
!$kgen end_callsite calc
```

2.2.2 KGEN command-line user interface

KGEN can be invoked using command line interface too.

>> $PATH_TO_KGEN/bin/kgen [KGEN flags] <filepath>:<namepath>

“filepath” is a path to a source file that contains a code region to be extracted.” “namepath” tells KGEN of the region of code to be extracted. Please see namepath section 2.3.1 below for more detail.

2.3 Supporting kernel generation for large-sized software

As we are interested in extraction a kernel from large-sized code, it is likely that the software adopts modular approaches with multiple source files. To extract a kernel from the software

having multiple source files, it needs to know all required paths for searching modules and correct set of macro definitions.

2.3.1 Namepath

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

Namepath is consecutive names with colons between them as a separator. For example, if name B is inside of A, then it can be represented by “A:B”. In practice, it is used to specify “ identifiers” in KGEN. For example, a kernel extraction region can be specified as following:

example)
```
module A
        subroutine B
                CALL C(...)
        end subroutine B
end module A
```
To specify “CALL C(...)” statement in above sample code, user can use “A:B:C” namepath.

To increase the usability of “namepath”, several syntactic features are added to above basic usage.

The separator of colon can be used as a metacharacter meaning of “any” similar to “*” in “ls” linux command. First, leading colon means any names can be placed before a name placed next to the colon. For example, “:name_a” means any namepath that ends with “name_a”. Similarity, colon at the end of a namepath means any names can be followed after a name placed before the colon. For example, “name_a:” matches to any namepath that starts with “name_a”. Finally, double colons between names means any names can be placed between the two names. For example, “name_a::name_b” matched to any namepaths that starts with “name_a” and ends with “name_b”

Namepath examples)
C => A name that has only one-level whosename is “C” such as “module C”
:C => any name ends with “C” such as any variable in a subroutine in a module
C: => any name whose top-level name is “C” and may contains lower-level names such as all variables in a subroutine of “C”
:C: => any names of “C” in any levels
A::C => Any names whose top-level name is A and whose lowest-level name is C

2.3.2 User-provided “include” information

KGEN accepts an INI-format file with “-i” command-line option. In the INI file, user can provide KGEN with additional information including macro-definitions, include directories and others. 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, an option is added in a line or over multiple lines. Each option has a format of key and value pair with a separator of “=”. Value part can be missed depending on the type of option.

To automate the creation of this “include” ini file for two specific application(CESM and MPAS), two Python scripts are created in “bin” sub-directory of KGEN distribution. The syntax for using the scripts is:

>> ${KGEN_HOME}/bin/gen_include_[cesm|mpas] <logfile>

The output from executing the script is “include.ini”. The INI file contains macro definitions and include paths collected from log file(s). User may use the generated “include.ini” as a base for kernel extraction.

2.3.2.1 INI sections applicable to 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.

2.3.2.2 INI sections applicable 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.

2.3.2.3 INI sections applicable to KGEN operations

User can provided additional information through “import” section in a INI file.

```
[import]
filepath = action
```

“source” and “object” actions are implemented as of this version. “source” action in import section provides KGEN with paths to additional files to be analyzed before starting main parsing

tasks. “object” action specifies a path to an object file that will be copied to “kernel” output directory.

[import]
/path/to/source/file.F90 = source
/path/to/object/file.o = object

2.3.3 User-provided “exclude” information

KGEN accepts an INI-format file with “-e” command-line option. In the INI file, user can provide KGEN with information of names(or namepaths) 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.

2.3.3.1 namepath section

When “namepath” is specified in a section of INI file, “actions” specified in an option are applied to namepath in the option.

[namepath]
namepath = [action]

“namepath” in an option line specifies target of action. The syntax of “namepath” is explaiend in section 2.3.1.

Regardless of actions specified in an option line, any name in execution part of Fortran source codes that matches to namepaths will be skipped from name resolution in KGEN. This is also true if there is no action is specified.

There are two “actions” defined in this version.

When “skip_module” is specified as “action”, KGEN will not use a module specified by the namepath during name resolution tasks. This actions is useful when an module implemented in external library is used but not relevant to kernel extraction. By specifying this action, user can prune search tasks.

When “remove_state” is specified as “action”, KGEN will not save state data specified by the namepath. This action may be useful discarding components of a derived type. If not all

components of a derived type should be used for saving state data, user can specify components of the derived to be excluded from the state saving. This action may be used together with above “skip_module” action.

3. Kernel extraction examples

3.1 simple example

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 “simple” 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.

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

```
>> ${KGEN} \
        -D ROW=4,COL=4 \
        -I ${SRC_DIR} \
        --timing repeat=1000 \
         --invocation 0:0:1,0:0:2,0:0:3 \
        --kernel-compile FC='${FC}',FC_FLAGS='${FC_FLAGS}' \
        --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.

“>> **${KGEN}**” : This invokes KGEN. It is assumed that ${KGEN} has valid path to “kgen” in “bin” directory.
“-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”

“--timing repeat=1000”: “--timing” flag specifies information for timing measurement. “repeat” sub-flag specifies how many calls to be made to kernel to measure timing. Larger number increases the resolution of measurement but increase time to measure.

“--invocation 0:0:1,0:0:2,0:0:3”: “--invocation” flag specifies when to save state data in terms of calls to the kernel. In this example, state data will be saved first, second, and third calls to the kernel. First and second numbers in colon separated tripes represents MPI ranks and OpenMP number each. “0” is used for non-MPI application and non-OpenMP application as like this example. Please see a section below that explains KGEN options in more detail.

“--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.

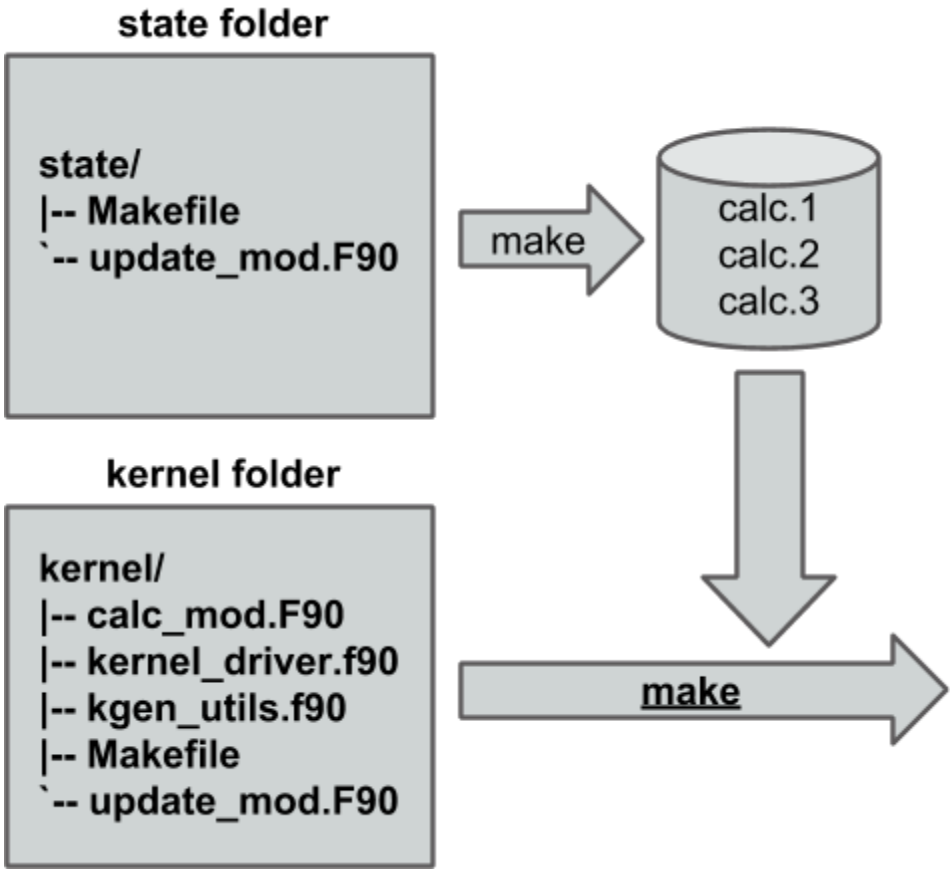
“--state-build cmds="cd ${SRC_DIR}; make build"” : “--state-build” KGEN flag provides command how to build instrumented original production application. KGEN will generate Makefile that contains the content of “cmds” sub-flag as a part of building task. In this example, it is assumed that Makefile is 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, three data files, “calc.0.0.1”, “calc.0.0.2”, and “calc.0.0.3” 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 kgen_utils.o kgen_utils.f90
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 kgen_utils.o calc_mod.o

./kernel.exe

 ***************** Verification against 'calc.1' *****************

 Number of verified variables:           3
 Number of identical variables:          3
 Number of non-identical variables within tolerance:     0
 Number of non-identical variables out of tolerance:     0
 Tolerance:   1.000000000000000E-015

 Verification PASSED

 calc : Time per call (usec):   3.700000047683716E-002

 ***************** Verification against 'calc.2' *****************
...
 ***************** Verification against 'calc.3' *****************
...
 ************************************************************
 calc summary: Total number of verification cases: 3
 calc summary: Average call time of all calls (usec): 3.3E-2
 ************************************************************
```

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. This verification repeated per each of three data files. It also shows that the kernel execution took 3.7E-002 usec for the first verification and 3.3E-2 usec for all three verifications.

3.2 simple-MPI example

In this section, an example of generating a kernel of “calc” from “calc_mod.F90” source file in simple-MPI folder is explained. This example shows a workflow of using KGEN for MPI

applications. Especially, this example includes the usage example for using “-e” flags which is an important part of supporting large-sized application.

program.F90

```
PROGRAM calc
  USE update_mod, only : &
    update
  INCLUDE 'mpif.h'
  INTEGER t, rank, N, err
  CALL mpi_init(err)
  CALL mpi_comm_size(...)
  CALL mpi_comm_rank(...)
  DO t=1,10
    CALL update(rank, N)
  END DO
  CALL mpi_finialize(err)
END PROGRAM
```

update_mod.F90

```
MODULE update_mod
  USE calc_mod, only : calc
  PUBLIC update
CONTAINS
  SUBROUTINE update(rank, N)
    INCLUDE 'mpif.h'
    INTEGER, INTENT(IN)::rank,N
    INTEGER :: i, j,err,lsum,gsum(N)
    INTEGER :: output(COL,ROW)
    gsum = 0
    DO i=1,COL
      DO j=1,ROW
        !$KGEN callsite calc
        CALL calc(i, j, output)
      END DO
    END DO
    lsum = SUM(output)
    CALL mpi_gather(lsum, …)
    IF (rank==0) THEN
      print *,'global sum=', SUM(gsum)
    END IF
  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
    CALL print_msg('start')
    output(i,j) = i + j
    CALL print_msg('finish')
  END SUBROUTINE
  SUBROUTINE print_msg(msg)
    CHARACTER(*),INTENT(IN):: &
      msg
    PRINT *, msg
  END SUBROUTINE
END MODULE
```

In this example callsite is “CALL calc(i, j, output)” in update_mod.F90.

Please change FC and FC_FLAGS in Makefile and src/Makefile if required to meet your compiler and MPI environment.

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

NOTE: Following macros are assumed in the command line.
SRC := ${SRC_DIR}/update_mod.F90
CALLSITE := update_mod:update:calc

```
>> ${KGEN} \
        -D ROW=4,COL=4 \
        -I ${SRC_DIR}:${MPI_INC} \
        -i include.ini \
        --timing repeat=1000 \
        --mpi enable \
         --invocation 0:0:1,0:0:3,1:0:1,1:0:3 \
        --kernel-compile FC='${FC}',FC_FLAGS='${FC_FLAGS}' \
        --state-build cmds="cd ${SRC_DIR}; make build" \
```

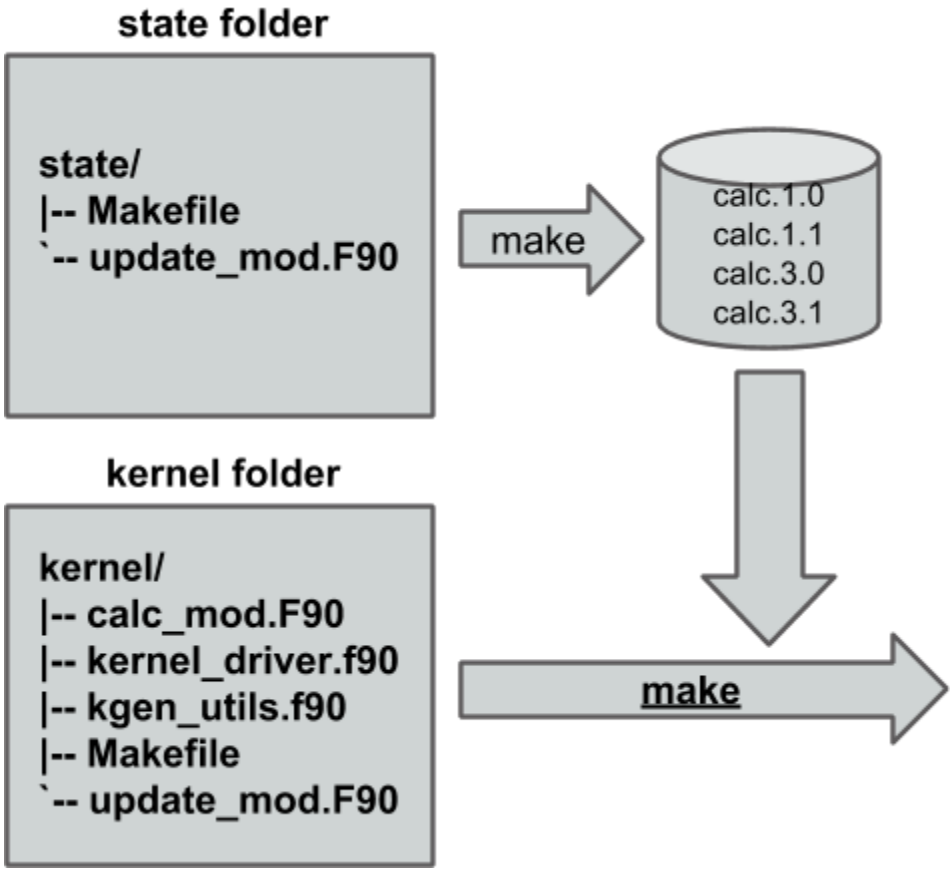
```
--state-run cmds="cd ${SRC_DIR}; make run" \
${SRC}:${CALLSITE}
```

Only “--mpi” flag and “--invocation” flag are explained in this section. Please see a section for “simple” example for the other flags.

“--mpi enable”: “--mpi” flag specifies MPI-related information. “enable” sub-flags specifies that KGEN extracts a kernel from MPI application.

“--invocation 0:0:1,0:0:3,1:0:1,1:0:3”: “--invocation” flag specifies when to generate state data in terms of MPI ranks, OpenMP numbers and Nth calls to the kernel region. In this example, four data files will be created from second and fourth invocation from MPI rank 0 and 3. “0” for OpenMp number is used to specify non-OpenMP application such as “1:0:3”

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, four data files, “calc.0.0.1”, “calc.0.0.3”, “calc.3.0.1”, and “calc.3.0.1” 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 kgen_utils.o kgen_utils.f90
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 kgen_utils.o calc_mod.o

./kernel.exe

 ***************** Verification against 'calc.1.0' *****************

 Number of verified variables:           1
 Number of identical variables:          1
 Number of non-identical variables within tolerance:          0
 Number of non-identical variables out of tolerance:          0
 Tolerance:   1.000000000000000E-015

 Verification PASSED

 calc : Time per call (usec):   6.000000052154064E-03

 ***************** Verification against 'calc.1.1' *****************
...
 ***************** Verification against 'calc.3.0' *****************
...
 ***************** Verification against 'calc.3.1' *****************
...

 ****************************************************************
 calc summary: Total number of verification cases: 4
 calc summary: Average call time of all calls (usec):   6.0E-03
 ****************************************************************
```

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. This verification repeated per each of three data files. It also shows that the kernel execution took 6.00E-003 usec for the first verification and 2.19E-002 usec for all four verifications.

3.3 simple-region example

In this section, an example of generating a kernel of “calc” from “calc_mod.F90” source file in simple-region folder is explained. This example shows a workflow of extraction a region of source codes.

program.F90

```
PROGRAM calc
  USE update_mod, only : &
    update
  INCLUDE 'mpif.h'
  INTEGER t, rank, N, err
  CALL mpi_init(err)
  CALL mpi_comm_size(...)
  CALL mpi_comm_rank(...)
  DO t=1,10
    CALL update(rank, N)
  END DO
  CALL mpi_finialize(err)
END PROGRAM
```

update_mod.F90

```
MODULE update_mod
  USE calc_mod, only : calc
  PUBLIC update
CONTAINS
  SUBROUTINE update(rank, N)
    INCLUDE 'mpif.h'
    INTEGER, INTENT(IN)::rank,N
    INTEGER :: i, j,err,lsum,gsum(N)
    INTEGER :: output(COL,ROW)
    gsum = 0
    !$KGEN begin_callsite calc
    DO i=1,COL
      DO j=1,ROW
        CALL calc(i, j, output)
      END DO
    END DO
    !$KGEN end_callsite
    lsum = SUM(output)
    CALL mpi_gather(lsum, …)
    IF (rank==0) THEN
      print *,'global sum=', SUM(gsum)
    END IF
  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
    CALL print_msg('start')
    output(i,j) = i + j
    CALL print_msg('finish')
  END SUBROUTINE
  SUBROUTINE print_msg(msg)
    CHARACTER(*),INTENT(IN):: &
      msg
    PRINT *, msg
  END SUBROUTINE
END MODULE
```

This example is exactly the same to simple-MPI example except it has “begin_callsite” and “end_callsite” KGEN directives in update_mod.F90. Please see simple-MPI example section for the explanations of extracting, building, and executing kernel.

4. 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

[--invocation]
meaning : specifies when to generated state data from which MPI ranks and OpenMP threads.
syntax: mpi_rank:openmp_num:invocation[,mpi_rank:openmp_num:invocation[...]]
examples
--invocation 0:1:2 => mpi rank0, openmp num 1, and third invocation of the kernel(starts from 0)
--invocation 1-2:3-4:5-6 => mpi rank1 and 2, openmp num 3 and 4, and sixth and seventh invocations of the kernel
Use 0 for "non MPI application" and use 0 for "non OpenMP application" in the first and second part of the syntax.

[--mpi]
meaning : Turns on MPI supports in KGEN. There are several sub-options: “enable”, “comm”, “use”, and “header”. “enable” specifies that KGEN extracts a kernel from MPI application. This is a mandatory for MPI application. “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"

[--openmp]
meaning : Turns on OpenMP supports in KGEN. There is one sub-option: “enable”. “enable” specifies that KGEN extracts a kernel from OpenMP application. This is a mandatory for OpenMP application.

[--intrinsic]
meaning : flags to let KGEN skip searching for names of intrinsic-procedures. At minimum, one of “skip” or “noskip” should be provided. With “except” sub-flag, user can specify which “namepath” should be considered as exception. With “add-intrinsic” sub-flag, user can add new intrinsic function names. default: --intrinsic skip
example) --intrinsic skip,except=mod_A.subr_B.sum

[--source]
meaning : this flags specifies information related to source file. “format” sub-flag specifies the Fortran format of source files “fixed” is used for F77 format and “free” used for F90 and later format. With this sub-flag, KGEN forces to use the specified format regardless of file extension. “strict” format let parser of KGEN informe to apply format strictly or not. Default is of the sub-flag is “no”, “alias” sub-flag create path alias. This is useful if you have one file physical location but has two different paths that points the same physical path.
example) format=free,strict=no,alias=/path/A:/path/B

[--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

[--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 ./”

[--outdir]

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

[--timing]
meaning : provides information about performance measurement. One sub-flag is defined in this version: repeat. “repeat” provides the number of invocations to the kernel subprogram to enhance the measurement resolution.
example) --timing repeat=1000

[--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. The variable to be printed is specified as namepath explained in section 2.3.1
example) --debug printvar=module_a:subroutine_b:var_c,module_1:subroutine_2:var_3

[--verbose]
meaning: This flag sets the initial verbosity level in the generated kernel. Default value is 1. User can modify the verbosity level by changing the verbosity value that is hard coded in the generated callsite file.

[--check]
meaning: This flag provides KGEN with correctness check-related information. Current implementation only allows perturbation related information. “pert_invar” sub-flag select an input variable for perturbation test. “Pert_lim” sub-flag sets the magnitude of perturbation. Default value is '1.0E-15'.
example) --check pert_invar=varname,pert_lim=1.0E-7

[--add-mpi-frame]
meaning: This flag specify to create MPI framework for replicating kernel execution across multiple MPI ranks. This is simple duplication of kernel execution without having any communication among kernels. Two sub-options are allowed: “np” and “mpiexec”. “np” sets the number of MPI ranks and “mpiexec” sets the path to mpiexec.
example) --add-mpi-frame=np=4,mpiexec=mpirun

6. Known Issues

- Cyclic linked list is not supported yet.
- Pointer variable that is associated with part of input state to the kernel may (or may not) generate issues depending on the usage of the variable within the extracted kernel