Myriad 2 MDK Kernel Unit Testing

User Manual

v1.21

Intel® Movidius™
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Table of Contents

1 Introduction

1.1 Overview

This document contains all the information needed to run and implement unitary tests for MDK kernels.

1.2 Directory structure

The following table shows directories that are part of the testing framework:

Directory	Contents
`mdk/regression/unittest`	Test framework classes.
`mdk/regression/unittest/testrunner`	General framework classes unrelated to MDK kernels.
`mdk/regression/unittest/datagenerator`	Data generator classes.
`mdk/regression/unittest/g_test`	Google test source code.

2 Running existing tests

2.1 Prerequisites

Kernel testing framework uses Google test C++ testing framework for building unit tests. Google test sources must be available in order to build unit tests.

Additionally, the framework uses `moviDebugDll` library that allows modifying and getting data from `.elf` files. The library for both Windows and Linux can be found in the `lib` directory from the tools used by the MDK.

In order to compile on PC the code that uses built-in functions into moviCompile it is needed the `libintrinsics.a` which contains the PC versions of the built-ins, and for half data type. The library for both Windows and Linux can be found in the `common/swCommon/pcModel/compilerIntrinsics` location.

There is also a matrix library, which for now contains only the PC versions of the transpose and rotate functions, and is needed by the unit tests. It can be found in the `pcModel/compilerVectorFunctions` location.

2.2 Building and running tests

Using the `SampleUnitTesting` example for illustration, make the following steps:

1. Move to unit test's folder
   `examples/HowTo/SampleUnitTesting/unittest`
2. Start `moviDebugServer` or `moviSim`
   either `make start_simulator`, or `make start_server`
3. To build the unit test run the following command :
   `make all`
   Additionally the debug interface may be changed. For this the `DBGI` variable can be used.

Examples:

`make all`

The command will build the test for the Myriad 2 platform using moviSim (the default values).

`make all DBGI=jtag`

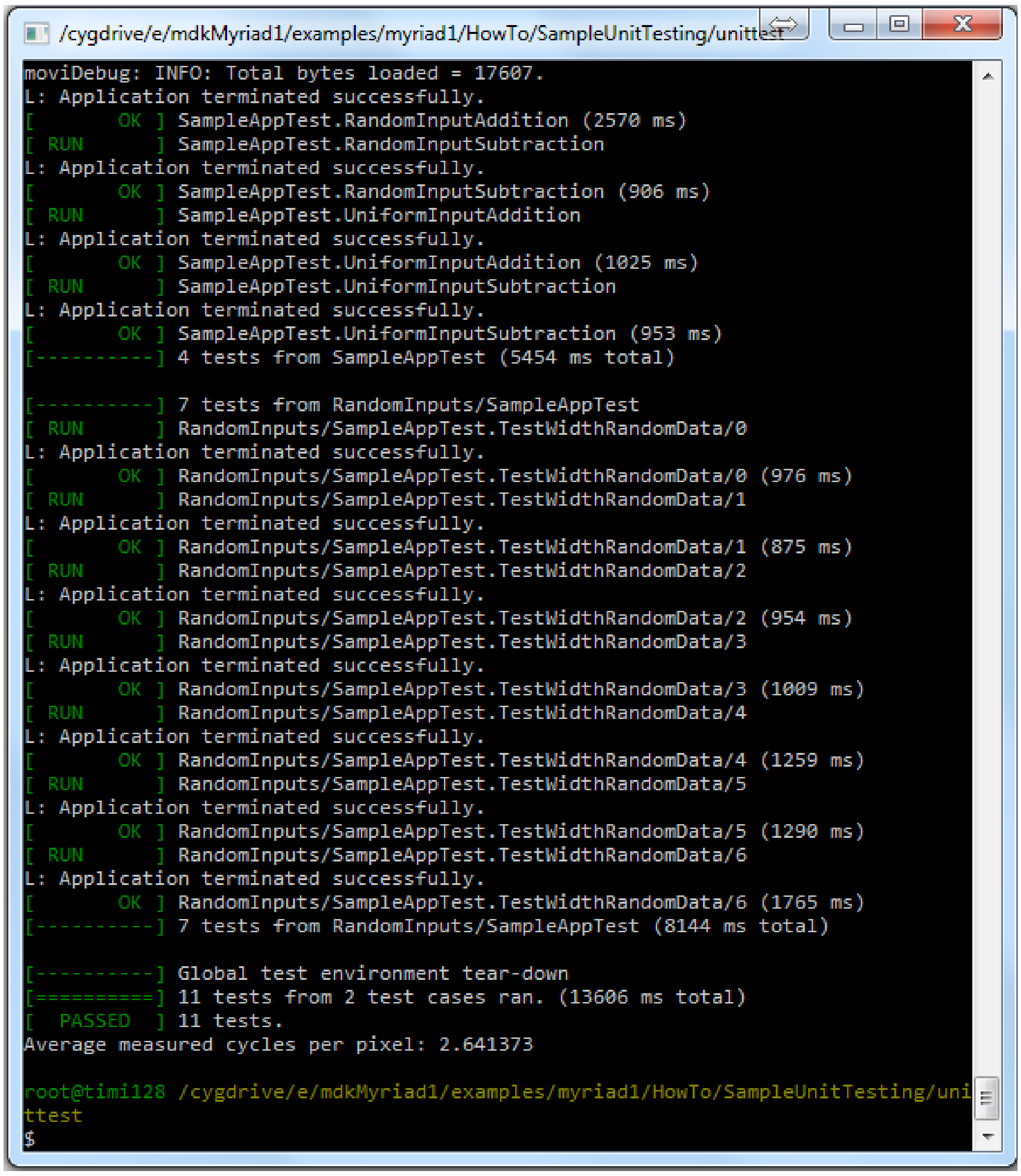
The command will build the test for the Myriad 2 platform using moviDebugServer.

NOTE: For further builds there is no need to rebuild the entire list of kernels, if they are not modified. The library `mvcv.a` is created in `MvCV/shared/unittest/output/myriad2` and to remove it run the command: `make clean_lib`.

4. To run a build test:

   `make run_test`

   If everything works, an output similar with the one below should appear:



/cygdrive/e/mdkMyriad1/examples/myriad1/HowTo/SampleUnitTesting/unittest

```
moviDebug: INFO: Total bytes loaded = 17607.
L: Application terminated successfully.
[       OK ] SampleAppTest.RandomInputAddition (2570 ms)
[ RUN      ] SampleAppTest.RandomInputSubtraction
L: Application terminated successfully.
[       OK ] SampleAppTest.RandomInputSubtraction (906 ms)
[ RUN      ] SampleAppTest.UniformInputAddition
L: Application terminated successfully.
[       OK ] SampleAppTest.UniformInputAddition (1025 ms)
[ RUN      ] SampleAppTest.UniformInputSubtraction
L: Application terminated successfully.
[       OK ] SampleAppTest.UniformInputSubtraction (953 ms)
[----------] 4 tests from SampleAppTest (5454 ms total)

[----------] 7 tests from RandomInputs/SampleAppTest
[ RUN      ] RandomInputs/SampleAppTest.TestWidthRandomData/0
L: Application terminated successfully.
[       OK ] RandomInputs/SampleAppTest.TestWidthRandomData/0 (976 ms)
[ RUN      ] RandomInputs/SampleAppTest.TestWidthRandomData/1
L: Application terminated successfully.
[       OK ] RandomInputs/SampleAppTest.TestWidthRandomData/1 (875 ms)
[ RUN      ] RandomInputs/SampleAppTest.TestWidthRandomData/2
L: Application terminated successfully.
[       OK ] RandomInputs/SampleAppTest.TestWidthRandomData/2 (954 ms)
[ RUN      ] RandomInputs/SampleAppTest.TestWidthRandomData/3
L: Application terminated successfully.
[       OK ] RandomInputs/SampleAppTest.TestWidthRandomData/3 (1009 ms)
[ RUN      ] RandomInputs/SampleAppTest.TestWidthRandomData/4
L: Application terminated successfully.
[       OK ] RandomInputs/SampleAppTest.TestWidthRandomData/4 (1259 ms)
[ RUN      ] RandomInputs/SampleAppTest.TestWidthRandomData/5
L: Application terminated successfully.
[       OK ] RandomInputs/SampleAppTest.TestWidthRandomData/5 (1290 ms)
[ RUN      ] RandomInputs/SampleAppTest.TestWidthRandomData/6
L: Application terminated successfully.
[       OK ] RandomInputs/SampleAppTest.TestWidthRandomData/6 (1765 ms)
[----------] 7 tests from RandomInputs/SampleAppTest (8144 ms total)

[----------] Global test environment tear-down
[==========] 11 tests from 2 test cases ran. (13606 ms total)
[  PASSED  ] 11 tests.
Average measured cycles per pixel: 2.641373

root@timi128 /cygdrive/e/mdkMyriad1/examples/myriad1/HowTo/SampleUnitTesting/uni
ttest
$
```

The executable file of a test is generated in the **run** folder of the unit test directory. It can be run directly from there instead of using **`make`** `run_test` command.

3 Adding new tests

3.1 Create a new test

The following steps should be followed to add a new test:

1. Create and build an application that calls the tested function:
   a. An application should be created that allocates memory for variables and buffers needed for tests.
   b. A separate variable should be defined for each parameter of the tested function.
   c. Defined variables must not be initialized.
   d. The function must be called with the defined variables as parameters.
   e. Separate buffers should be defined to pass the input data (as image lines) and store the output results (it is also possible to use the same buffer for both input and output data); the size allocated for these variables must be big enough to contain the biggest allowed input.

Look at the `dummy` app from the `SampleUnitTesting` example.

The Leon code is general for a Myriad 2 application, and the shave code in the `dummy/shave/init.c` file is:

```
#include <sampleApp.h>
#include <svuCommonShave.h>

// define your variables HERE
#define MAX_WIDTH 1920
unsigned char __attribute__ ((aligned (16))) in[MAX_WIDTH];
unsigned int __attribute__ ((aligned (16))) width;
unsigned char __attribute__ ((aligned (16))) value;
unsigned char __attribute__ ((aligned (16))) op;
unsigned int __attribute__ ((aligned (16))) pxToChange;
/*output pre pad marker*/
uint32_t __attribute__((section(".kept.data"))) out_u32prePad[4]
__attribute__ ((aligned (16)));
/*output data marker*/
u8 __attribute__((section(".kept.data"))) out[MAX_WIDTH] __attribute__
((aligned (16)));
/*output post pad marker*/
uint32_t __attribute__((section(".kept.data"))) out_u32postPad[4]
__attribute__ ((aligned (16)));

int main(void)
{

  //your code HERE
  sampleApp_asm(in, width, value, op, pxToChange, out);

  SHAVE_HALT;
  return 0;
}
```

As you can see, the defined variables are not initialized, and the size allocated for `in` and `out` arrays is `MAX_WIDTH`, considering that this is the biggest allowed input.

Then the `asm` function (`sampleApp_asm`) is called with the defined variables as parameters.

The `dummy/Makefile` for this app includes `mdk/common/generic.mk` and it uses the internal tools for the build. As a result of the build, it creates the `dummy.elf` file.

But how is this application related to the unit test framework? The file you will create in step 2, and the reason of doing this will enlighten you.

2. For asm functions, create a function wrapper that allows function parameters to be modified by the test framework. The function wrapper file should be placed in the same directory with the unit test file; it should use TestRunner class to add input and output parameters to the function and retrieve output data.

   Example:

```
#include "TestRunner.h"
#include "RandomGenerator.h"
#include "moviDebugDll.h"
#include "FunctionInfo.h"
#include <cstdio>

#define EXPECTED_CC (3)

TestRunner sampleAppTestRunner(APP_PATH, APP_ELFPATH, DBG_INTERFACE);

unsigned int SampleAppCycleCount;

void SampleApp_asm_test(unsigned char *in, unsigned int width,
unsigned char value, unsigned char operation, unsigned int pxToChange,
unsigned char *out)
{

  FunctionInfo& functionInfo = FunctionInfo::Instance();
  sampleAppTestRunner.Init();
  sampleAppTestRunner.SetInput("in", in, width, SHAVE0);
  sampleAppTestRunner.SetInput("width", width, SHAVE0);
  sampleAppTestRunner.SetInput("value", value, SHAVE0);
  sampleAppTestRunner.SetInput("op", operation, SHAVE0);
  sampleAppTestRunner.SetInput("pxToChange", pxToChange, SHAVE0);

  sampleAppTestRunner.GuardInsert(string("out"), SHAVE0, width, out);
  sampleAppTestRunner.Run(SHAVE0);
  SampleAppCycleCount =
sampleAppTestRunner.GetVariableValue(std::string("cycleCount"));
  functionInfo.AddCyclePerPixelInfo((float)(SampleAppCycleCount - 2)/
(float)width);
  functionInfo.setExpectedCycles((float)EXPECTED_CC);
  sampleAppTestRunner.GetOutput(string("out"), SHAVE0, width, out);
  sampleAppTestRunner.GuardCheck(string("out"), SHAVE0, width, out);
}
```

The constructor sampleAppTestRunner(APP_PATH, APP_ELFPATH, DBG_INTERFACE) creates a TestRunner object, and it initializes the "elf path string", that it is used to load the .elf file.

APP_ELFPATH is defined in mdk/regression/unittest/untittest.mk:

```
CPPFLAGS += -D"APP_ELFPATH=\"$(KERNEL_APP_PATH)/output/$
(KERNEL_APP_NAME).elf\""
```

and KERNEL_APP_PATH and KERNEL_APP_NAME are defined in the unit test Makefile for PC (examples/HowTo/SampleUnitTesting/unittest/Makefile):

```
KERNEL_APP_PATH = dummy
```

```
        KERNEL_APP_NAME = dummy
```

Therefore,

```
        APP_ELFPATH = dummy/output/dummy.elf
```

`TestRunner.Init()` initializes a `TestRunner` object, makes the connection to the debug server and it loads the elf file (`dummy/output/dummy.elf`).

Then it initializes the data defined in `dummy/shave/init.c` using `TestRunner.SetInput()` function. For example:

```
        sampleAppTestRunner.SetInput("op", operation, SHAVE0);
```

it goes to the address of `op` variable in the `.elf` file and replaces one byte in the memory with the value of the `operation` variable from this function.

After the inputs are set, the application can be run:

```
        sampleAppTestRunner.Run(SHAVE0);
```

The next part is used to determine how many cycles/pixels are needed to execute the asm code.

Eventually, after the application terminates successfully, it gets the value of the `out` variable from the app into `out` defined in this function.

In the `TestRunner.GetOutput()` function, the first parameter is a name of a variable from the unit test application (dummy/shave/init.c). The last parameter is a variable about where to store the output.

In `sampleAppTestRunner.GuardInsert()`, a two marker buffers (`out_prepad[4]` and `out_postpad[4]`) is filled with `0xdeadbeef` value and in and `sampleAppTestRunner.GuardCheck()` the value written in those buffers is verified if it is modified to trigger a test error if the tested code writes outside the desired buffers. The tested output buffers or output variables should have `__attribute__((section(".kept.data")))` and `__attribute__ ((aligned (16)))` to be sure they are in the proper tested area and are proper aligned.

3. Create the unit test

   Create the unit test file. It should include the header file that exports the wrapped asm function. The function can now be tested as if it would be a normal C++ function.

   Example:

```
#include "sampleApp.h"
#include "sampleApp_asm_test.h"
#include "InputGenerator.h"
#include "UniformGenerator.h"
#include "RandomGenerator.h"
#include <stdlib.h>
#include "ArrayChecker.h"

using ::testing::TestWithParam;
using ::testing::Values;

class SampleAppTest :  public ::testing::TestWithParam< unsigned int >
{
 protected:

   virtual void SetUp()
   {
         randGen.reset(new RandomGenerator);
         uniGen.reset(new UniformGenerator);
         inputGen.AddGenerator(std::string("random"), randGen.get());
         inputGen.AddGenerator(std::string("uniform"), uniGen.get());
   }
```

```
    unsigned char *input, *outAsm, *outC, value, operation;
    unsigned int pxToChange, width;

    RandomGenerator dataGenerator;
    InputGenerator inputGen;
    ArrayChecker outputCheck;
    std::auto_ptr<RandomGenerator>  randGen;
    std::auto_ptr<UniformGenerator>  uniGen;

  // virtual void TearDown() {}
};

TEST_F(SampleAppTest, RandomInputAddition)
{

   inputGen.SelectGenerator("random");
   width = 64;
   input  = inputGen.GetLine(width, 0, 255);
   value  = randGen->GenerateUChar(0,255);
   outAsm = inputGen.GetEmptyLine(width);
   outC = inputGen.GetEmptyLine(width);
   pxToChange = 16;
   operation = 1; //addition
   SampleApp_asm_test(input, width, value, operation, pxToChange,
outAsm);
   sampleApp(input, width, value, operation, pxToChange, outC);
   RecordProperty("CycleCount", SampleAppCycleCount);
   outputCheck.CompareArrays(outC, outAsm, width);
}

...
//---------------- parameterized tests -------------------------

INSTANTIATE_TEST_CASE_P(RandomInputs, SampleAppTest,
        Values(8, 32, 320, 640, 800, 1280, 1920);
);

TEST_P(SampleAppTest, TestWidthRandomData)
{
   inputGen.SelectGenerator("random");
   width = GetParam();
   input  = inputGen.GetLine(width, 0, 255);
   value  = randGen->GenerateUChar(0,255);
   outAsm = inputGen.GetEmptyLine(width);
   outC = inputGen.GetEmptyLine(width);
   pxToChange = (width == 8) ? 8 : randGen->GenerateUInt(8,width, 8);
   operation = 0; //subtraction
   SampleApp_asm_test(input, width, value, operation, pxToChange,
outAsm);
   sampleApp(input, width, value, operation, pxToChange, outC);
   RecordProperty("CycleCount", SampleAppCycleCount);
   outputCheck.CompareArrays(outC, outAsm, width);
}
```

Because multiple tests need to share common objects, a test fixture class is defined. For regular tests your fixture has to be derived from `testing::Test` and for parameterized tests, it has to be derived from `testing::TestWithParam<>`:

```
class SampleAppTest :  public ::testing::TestWithParam< unsigned int >
```

In the example there are some regular tests (`RandomInputAddition`, `RandomInputSubtraction`, `UniformInputSubtraction`, `UniformInputAddition`) and a parameterized test (`TestWidthRandomData`). To access the test parameter, `GetParam()` is used.

The code in the `SetUp()` function will be called right before each test. For `SampleAppUniTesting`, two data generators are defined (one random and another uniform), so you can generate random and uniform values. More details about InputGenerator is in section 3.4.

The idea of the unittest is to compare the result from the asm code run on Myriad against the result of the C code run on PC. The tests use `ArrayChecker.CompareArrays()` function that contains some assertion to compare two arrays.

The `SampleUnitTesting/unittest/Makefile` use the `GNU GCC` compiler to build: the GTest Framework files, the useful classes for the framework, the `sampleApp_unittest.cpp`, and the `sampleApp_asm_test.cpp`. It includes `mdk/regression/unittest/unittest.mk` and the result of the build is the executable `SampleUnitTesting/unittest/run/sampleApp_unittest(.exe)`.

More details on how to write unit tests using Google test framework are available on the project wiki:
https://code.google.com/p/googletest/wiki/Documentation

3.2 Overview of the unit test structure

Bellow is shown an overview of the needed files that show how to implement a unit test application:

/SampleUnitTesting

sampleApp.c	<---	the reference code tested against
sampleApp.asm	<---	the tested code (Myriad2 assembly)
unittest/	**<---**	**the unittest directory**
dummy/	**<---**	**a dummy app (see Section 3.1 – 1)**
leon/	**<---**	**leon code**
output/	**<---**	**directory for object files**
shave/	**<---**	**shave code (is calling the sampleApp.asm)**
Makefile	<---	for building app
out/	**<---**	**directory for object files from PC files**
run/	**<---**	**the location of unittest executable**
Makefile	<---	for building PC files
sampleApp_asm_test.cpp	<---	function wrapper (see Section 3.1 – **2**)
sampleApp_asm_test.h		
sampleApp_unittest.cpp	<---	unit test file (see Section 3.1 – **3**)

3.3 Debugging

Running the test executable using "-debug" options allows test debugging using `moviDebug`. Running the test this way shows a command prompt for each individual test after all the input data was loaded. In this way all `moviDebug` commands are available and can be executed. The "quit" command runs the current test (with all the changes that may have been made using `moviDebug` commands) and goes to the next test, stops after the data is loaded and the same process repeats. If you want to exit debug mode, use the "exit" command at the debug command prompt. This command finishes the current test and runs the remaining tests normally (not in debug mode).

3.4 Useful classes for writing kernel tests

A few classes are provided to simplify the process of writing test cases for MDK kernel. They are used to create new input data for the tested functions and generate different data patterns.

3.4.1 Class InputGenerator

This class contains useful methods to generate input lines which are usually used as parameters to MDK kernels. It automatically allocates memory for the requested number of lines with the requested size. Generated lines can be uninitialized or filled with different kind of patterns. The output of line generating functions is a simple or a double pointer to one or more C style array.

Currently there are two kinds of patterns that could be used to fill the generated lines: uniform and random. Uniform generator fills all lines with the same given value. Random generator fills the lines with random data in a given range.

All the methods of the `InputGenerator` class are exported by header `InputGenerator.h`

To select a data generator, the `SelectGenerator` method is used. It takes a single string parameter which is the name of the generator. Currently only two generators are implemented: uniform and random. Below is an example on how to select the needed generator:

```
std::auto_ptr<RandomGenerator>  randomGen;
std::auto_ptr<UniformGenerator>  uniformGen;
InputGenerator inputGen;

randomGen.reset(new RandomGenerator);
uniformGen.reset(new UniformGenerator);
inputGen.AddGenerator(std::string("random"), randomGen.get());
inputGen.AddGenerator(std::string("uniform"), uniformGen.get());
…
inputGen.SelectGenerator("uniform");
```

First, two data generators, one random and one uniform are added by using the `AddGenerator` function. Then the generator can be selected by its name which was previously provided by the call to `AddGenerator`.

Line generator functions:

`GetEmptyLine` returns a pointer to a single uninitialized line. It takes one input parameter: the line size.

Example:

```
unsigned char* line;
line = inputGen.GetEmptyLine(width);
```

`GetLine`: returns a pointer to a single line filled with the currently selected generator. The first parameter of the function is the line size. If the current generator is *uniform* the second parameter should be the initialization value. If the current generator is *random* then up to three parameters can be passed. Two parameters are for the low and high limits of the range in which the random values are generated. The third parameter is optional and can, if set, generate only random numbers which are multiple of that parameter value.

Example:

```
unsigned char* line;
```

For uniform generator:

```
line = inputGen.GetLine(width, 7);
```

For random generator:

```
line = inputGen.GetLine(width, 0, 255, 16);
```

`GetLines`: works the same as `GetLine`, but has an additional parameter, the second, which is the number of generated lines.

Example:

```
unsigned char** lines;
```

For uniform generator:

```
lines = inputGen.GetLine(width, height, 7);
```

For random generator:

```
lines = inputGen.GetLine(width, height, 0, 255, 16);
```

Two additional variations of the functions above exist for generating 16- and 32-bit floating point numbers: `GetLine[s]Float, GetLine[s]Float16`

The `InputGenerator` class also provides a function that can be used to fill an existing array with data generated by the current selected generator. The functions used for this are `FillLine`, `FillLines` and `FillLineRange`. The third form of the function fills a given range of an array.

Another function useful to add padding to a set of lines is `GetOffsettedLines`. The function takes three parameters: the first is a double pointer to the lines, the second is the number of lines and the third is an offset. The function returns another double pointer. Each line pointer returned by the function points to an offset position in the original line.

3.4.2 Class TestRunner

It is used for writing wrappers for assembly functions.

`TestRunner.Init` initializes a `TestRunner` object and makes the connection to the debug server.

`TestRunner.SetInput`. The first parameter of this function is a name of a variable from the unit test application (described in Section 3.1). The `SetInput` function goes to the address of this variable in the `.elf` file and replaces one or more values in the memory. The number of values to be replaced and their value is provided by the next parameters of the function. There are different variants of the functions for the most common variable types used as function parameters, including vector and matrix types. The prototypes of these functions can be found in the `regression\unittest\datagenerator\include\TestRunner.h` header file.

`TestRunner.Run` runs the application with the debugger using parameters set by calls to `TestRunner.Init`.

`TestRunner.GetOutput` – The first parameter of this function is a name of a variable from the unit test application (described in Section 3.1). The function returns one or more values from the address of the variable given as the first parameter. A number of overloaded variants of this function exist for different types of parameters.

3.4.3 Class FunctionInfo

Is a singleton class used to gather statistics related to the tested function. Currently it can only receive the number of cycles per pixels for a function, using `AddCyclesPerPixelInfo` and calculates their average which is printed in the test report.

Example:

```
//get an instance of the class
FunctionInfo& functionInfo = FunctionInfo::Instance();
functionInfo.AddCyclePerPixelInfo((float)(minMaxCycleCount/ width);
```