Application Note CMP01: Porting CoreMark-PRO to Bare-Metal

Version 2021.0916

Contact: peter.torelli@eembc.org

Contents

This application note assumes the reader is already familiar with CoreMark-PRO. Please refer to the User Guide and README in EEMBC's GitHub repository for an introduction:

https://github.com/eembc/coremark-pro

1 Introduction

By default, CoreMark-PRO is configured for a make-based toolchain with the assumption that the benchmark is run on the same host computer which it is built. This is useful because not only will it configure the component workloads correctly, but it will also automatically:

1. build all of the workloads;
2. run a verification iteration to check the results (`-v0`)
3. perform three performance runs (`-v1`) and take the median;
4. compute the final score and scale factors versus a single-thread run using a PERL script.
5. allow per-workload tuning of iteration count, which is required by the run rules to achieve proper runtime; and
6. allow the user to dynamically adjust the number of workers and contexts for experiments using the `XCMDS` make parameter

If the target is not the same as the host, then more steps are required. Each workload must be compiled individual, flashed to the device, verified, and then the score computed manually.

In this document, the term "bare-metal" means the application does not use an operating system. Instead, after the system initialization is performed, the user's `main` function is the entry point of execution.

2 Benchmark Architecture

It is important to understand how the source code is arranged and compiled in order to construct the project files correctly. This section covers the key concepts.

2.1 MITH

CoreMark-PRO is based on the multi-instance test harness (MITH). This framework allows the user to manipulate the size of worker and context parallelism for each workload. It also provides an adaptation layer (AL) for the specific hardware used by all of the workloads. The adaptation layer is where the developer places board-specific initialization, print, and timer code.

The source code for MITH is contained in the `mith/src` directory, and the source for the adaptation layer is in `mith/al/src`.

IMPORTANT: The adaptation layer is the only source code that may be modified, the remainder of the code must remain untouched. There is one exception related to the default entry point and arguments which will be discussed later.

2.2 Workloads

CoreMark-PRO is comprised of nine workloads. Each workload contains at least one kernel (except for the Loops test, which contains 24 in a single kernel file). Each workload provides its own `main` function. The `main` function uses MITH components to initialize, verify, and run the kernels. All of the workloads' main functions look remarkably similar because they are automatically generated by a workload builder, and they differ only in the number of kernels that are instantiated.

The workloads can be found in the `workloads` directory. There is one source file per workload. Even though the makefiles aren't used for bare-metal projects, examining them reveals the kernels that are compiled into the workload.

When compiling in local host mode, each workload is linked as its own executable under `builds/platform/toolchain/bin/*.exe`.

2.3 Kernels

Kernels are the fundamental benchmark units invoked by the workload. The kernels used by the workloads are stored in the somewhat confusingly named `benchmarks` directory. This document refers to components in this directory as "kernels" to avoid confusion. Some kernels were created specifically by EEMBC (such as CoreMark), others are real-world kernels such as numerical methods for FFT, LINPACK, or the well-known Livermore Loops algorithms for linear algebra.

In most cases, the dataset is included with the kernel, such as the JPEG test or the Radix2 FFT test. The data is stored in C files. Other benchmarks, such as the XML parser and ZIP test, create their own workloads dynamically on the heap at runtime.

IMPORTANT: The floating-point kernels come with both single- and double-precision datasets. It is important to configure the preprocessor macros to match the linked dataset, or the floating-point verification checks will fail.

All of the workloads included in CoreMark-PRO only use one kernel each, but other benchmarks, like AutoBench 2.0 and MultiBench combine up to six kernels in parallel.

3 Porting

Building and running the benchmark consists of the following steps, which will be explained in greater detail.

1. Create a basic project for the target board using the vendor's application builder
2. Add MITH and the adaptation layer source files to the project
3. Create nine different targets, each with its own macros and requisite kernels
4. Run verification mode for each workload
5. Run performance mode for each workload and adjust the iterations to achieve correct runtime
6. Optimize
7. Compute the final score

3.1 Creating the basic project

Most vendors supply some type of project boot-strapping application, for example: STMicroelectronics has CubeMX, Infineon has Modus Toolbox, and Silicon Labs has Simplicity studio. These applications sometimes allow the user to create a basic project for standard IDE-based toolchains, such as Keil or IAR, or their own Eclipse-based IDE. The advantage to these bootstrap applications is that they setup the initialization code for bare-metal execution, such as clocks, GPIOs, default linker maps, etc.

First, create a project and verify that `printf` works. By default, CoreMark-PRO uses the function `th_printf`, which if you follow it through the MITH framework, ultimately calls the SDK's `vsprintf` function. If `vsprintf` is not implemented, but `printf` is, it is recommended to simply define `th_printf` as `printf` using a preprocessor macro.

IMPORTANT: Floating-point formatters (`%f` and `%g`) are required.

Later on, it will be necessary to implement a timing function, now would be a good time to ensure that some sort of CPU ticker or interrupt is configured to generate an incrementing 32-bit counter of at least 1 kHz (one millisecond period).

Lastly, some of the workloads require a large amount of stack and heap memory. Set up the linker maps to supply at least 300 KB of heap and 100 KB of stack. This can be optimized per-workload later on.

3.2 Add the MITH and adaptation-layer source code

Next add all of the files in `mith/src` to the project. These files cannot be modified, all changes are made in the adaptation layer.

Add `mith/al/src` to the project. Two key files exist in this directory: `th_cfg.h` and `th_al.c`. The former contains configuration macros for MITH, and the latter is a place to add custom code.

3.2.1 Macros in `th_cfg.h`

There are quite a few macros in `th_cfg.h`. The ones called out below are most frequently responsible for causing confusion when porting to a new build system, but it is by no means a complete list. IDE and SDKs change frequently, which may require additional tuning.

In a few instances, setting one of these macros will enable a preprocessor error message, this can be commented out and ignored, it simply provided as a reminder.

```
FAKE_FILEIO=0
HAVE_FILEIO=0
```

File IO is not used by CoreMark-PRO, but it will still try to compile and link file functions. Many embedded SDKs do not provide these functions, so setting these macros to zero resolves the many undefined references.

```
HAVE_SYS_STAT=0
```

Similar to file IO, there are a few calls to get file statistics, again, not used by CoreMark-PRO. It is a good idea to set this to zero and use `STUB_STAT=1` (below) in the toolchain macro definition.

```
HAVE_PTHREAD=0
USE_NATIVE_PTHREAD=0
USE_SINGLE_CONTEXT=1
```

Since we are running in single-thread mode, setting these three macros will disable all reference to the OS `pthread` functions. This document does not describe how to implement symmetric multiprocessing, however, MITH uses a POSIX interface.

```
HAVE_STDRUP=0
```


Again, CoreMark-PRO does not use `stdrup` but other MITH benchmarks do, it is advised to disable this macro if your SDK does not provide an implementation.

3.2.2 Toolchain macros

These macros are not in `th_cfg.h`, and must be set in the toolchain settings.

`STUB_STAT=1`

This turns on a stub for the `stat` function, mentioned above.

`NO_ALIGNED_MALLOC=1`

Many SDKs do not provide an aligned malloc operation, but it is common in operating systems like Linux using GCC. Set this macro to avoid the malloc-related compile failure.

`USE_CLOCK=1`

This is can be a bit tricky. If your SDK provides an implementation of `time.h` function clock that has 1ms resolution, then that can be used in the adaptation layer. If not, disable this and implement it in the `th_al.c` section on clocking.

`HOST_EXAMPLE_CODE=1`

This macro disables the host-port code for running on the local host. In some cases, it may be helpful to leave this enabled because time functions may be provided by the SDK, or the floating-point manipulation code is sufficient. In other cases, it will allow greater flexibility for implementing timing functionality. It is recommended to leave this turned on and simply replace the verbose preprocessor conditionals in the `th_al.c` timing functions with your own ticker functionality.

3.2.3 Kernel-specific macros

Some kernels have their own macros, in particular, `USE_FP32` and `USE_FP64`. These macros will be explained during the kernel porting section.

3.2.4 Datatypes in `th_types.h`

The size of the long datatype, as well as typedefs for integer and float sizes are also defined in `th_cfg.h` but impact `th_types.h`. For example, `EE_SIZEOF_LONG` is set to 4 by default, which should be sufficient. This in turn assigns standard datatypes to custom datatypes like `e_u32` and `e_s16`. The C standard sizes header was not yet official at the time of development, so there may need to be some adjustments made here.

3.2.5 Timing

CoreMark-PRO performs timing using the `al_signal_*` functions in `th_al.c`. These functions should ultimately return the number of milliseconds elapsed in between `al_signal_start`, `al_signal_now` and `al_signal_finish`. The clocking macro section of `th_al.c` attempts to provide some default methods depending on the SDK. If none of these are applicable, comment out this section, declare storage for variables initial and final, which keep running counts.

3.2.6 Floating-point manipulation

Some of the kernels load floating-point data from a structure of sign, mantissa, and exponent, and shape the value based on the machine hardware. The floating-point SNR verification check also manipulates the value's components directly. If you have disabled `HOST_EXAMPLE_CODE`, you will need to provide functionality for floating-point manipulation in the form of `load_dp`, `store_dp`, `load_sp` and `store_sp` (all located in `th_al.c`). It is recommended to utilize the built-in functions, unless there the ISA or SDK provides a more optimal strategy.

3.3 Create the targets

By this point, the project should have only one target, and should compile with no errors, assuming your application builder provided a main function.

Depending on your IDE, subsets of the main project that include or exclude certain files and folders may be called targets, workspaces, projects, etc. This document uses the term "target", which refers to a method for partitioning the source into various build configurations within the main project, each with their own source files, and compile and link options. Different targets will have different preprocessor macros for their kernel, and may have different linker maps for the stack and heap.

If a `main` function already exists, there are two strategies:

1. Move the initialization code from the default `main` function into `al_init`, and remove the default `main` function; or
2. Rename the `main` function in each workload file to something other than `main` and invoke it from within the default `main`

The first option is preferred, although if your IDE does not allow passing `argc` and `argv` through the debugger, you will need to manually create `argc` and `argv` before the invocation to `al_init`.

IMPORTANT: This is the only situation in which you are allowed to modify workload code.

Since the `make` flow is not in use, we need to manually construct each target by adding the workload file, and the required kernel files as well as setting the proper macros. The following table illustrates which files to include for each workload, and macros:

Workload	**Macros**	**Kernel (benchmark/) files**
cjpeg-rose7-preset	SELECT_PRESET_ID=1, USE_PRESET	consumer_v2/cjpeg/*.c consumer_v2/cjpeg/data/Rose256_bmp.c
core		core/core_*.c
linear_alg-mid-100x100-sp	USE_FP32=1	fp/linpack/linpack.c fp/linpack/ref/inputs_f32.c
loops-all-mid-10k-sp	USE_FP32=1	fp/loops/loops.c fp/loops/ref-sp/*.c
nnet_test	USE_FP64=1	fp/nnet/nnet.c fp/nnet/ref/*.c
parser-125k		darkmark/parser/*.c
radix2-big-64k	USE_FP64=1	fp/fft_radix2/fft_radix2.c fp/fft_radix2/ref/*.c
sha-test		darkmark/sha/*.c

`zip-test`	`MITH_MEMORY_ONLY_VERSION, ZLIB_COMPAT_ALL, ZLIB_ANSI`	`darkmark/zip/zip_darkmark.c darkmark/zip/zlib-1.2.8/*.c` but `exclude gzread.c and gzwrite.c`

Table 1. Macros and kernel files required for each workload target.

Here are two examples of what the workspace might look like in two different IDEs.

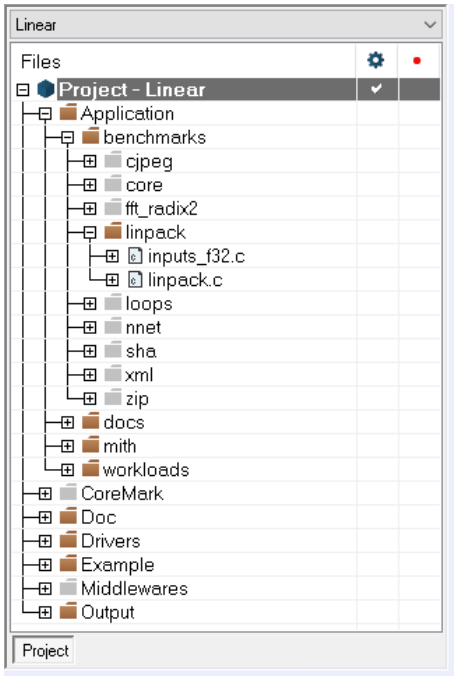

Figure 1. Example IAR project showing excluded components when the Linear workload is selected.

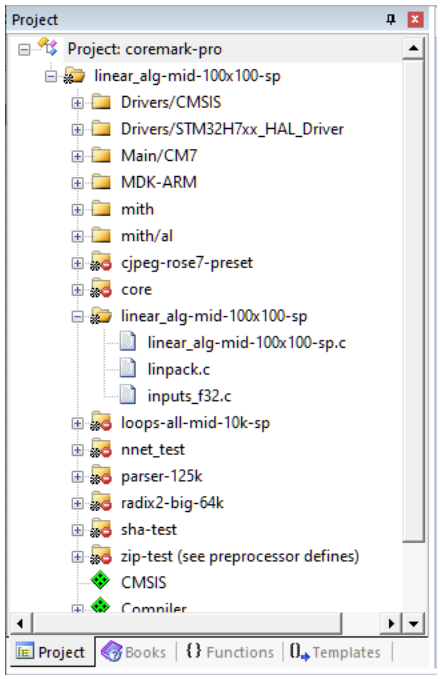

Figure 2. Example Keil project showing excluded components when the Linear workload is selected.

3.4 Run a verification iteration

Pick a workload and configure the arguments to the main function to be `-v1`. This will run a single iteration and perform any validation checks. Download the binary to the target and verify correct operation. If there were any errors in the implementation, the output will indicate any failures. Several macros in `th_cfg.h` enable debugging: `BMDEBUG` and `THDEBUG` enable debug print messages in the workloads and kernels; `DEBUG_ACCURATE_BITS` is helpful for identifying floating-point mismatches during verification mode.

3.5 Run in performance mode

Switch the `-v0` to `-v1` and run the workload. It should run for at least 10 seconds. If it doesn't, increase the number of iterations by adding `-i100`, which would set the number of iterations to 100, for example.

Run all workloads and create a baseline set of scores.

3.6 Optimize

With the baseline set of scores collected, experiment with the linker map, or the compiler/linker options to achieve optimal performance.

3.7 Compute the overall score

The score for CoreMark-PRO is a scaled geometric mean of scaled workload scores. Each workload is normalized to a reference value, first. The following table illustrates this:

Component	Scale Factor (xN)	Reference Score (rN)
cjpeg-rose7-preset	1	40.3438
core	10000	2855
linear_alg-mid-100x100-sp	1	38.5624
loops-all-mid-10k-sp	1	0.87959
nnet_test	1	1.45853
parser-125k	1	4.81116
radix2-big-64k	1	99.6587
sha-test	1	48.5201
zip-test	1	21.3618

Table 2. Scaling and reference factors for score computation.

$$Overall\ Score = 1000 \times \sqrt[n]{\prod_{i=1}^{n} \frac{C_i}{R_i} S_i}$$

Where C is the workload component score, S is the workload scale factor and R is the workload reference score for each of the n workloads. The final scalar of 1000 moves the number into a large, more manageable integer.

