CMake Performance Tips

From KitwarePublic

While CMake itself is already very fast, there are some tuning things you can do to ensure it works as fast as possible. This list is supposed to be sorted (ordered) according to relevance (severity, amount of savings). TODO: do that sorting.

Contents

- 1 Build CMake binary with optimization enabled
- 2 CMake configure time
  - 2.1 Investigation/profiling methods
  - 2.2 Pinpoint expensive operations
  - 2.3 Use LIST(APPEND ...)
  - 2.4 Reduce add_custom_command()s DEPENDS lists
  - 2.5 Use an include guard
  - 2.6 Conditional find_package()
  - 2.7 Split modules into functions/definitions
  - 2.8 Conditional switching of (dummy) methods
  - 2.9 Loop optimizations
  - 2.10 REGEX can be awfully slow

Build CMake binary with optimization enabled

Ok, this is obvious, but anyway. Let's say you build CMake yourself without any special settings, e.g.

```
$ cmake ..
$ make
```

If you do it this way, you will get a CMake with optimizations turned off. There are different ways to get an optimized build. You can select one of the predefined build types:

```
$ cmake -DCMAKE_BUILD_TYPE=RELEASE ..
$ make
```

Also possible are RELWITHDEBINFO and MINSIZEREL.

or

```
$ export CXXFLAGS=-O2
$ cmake ..
$ make
```

or

```
$ export CXXFLAGS=-O2
$ cmake ..
$ make edit_cache (or ccmake ..)
... edit CMAKE_CXX_FLAGS in the advanced view
$ make
```

CMake built with optimizations enabled can give you an almost 50% performance boost (time for running CMake on VTK went down from 25 s to 14 s).

CMake configure time

Investigation/profiling methods

Some CMake configure time performance behaviour can be observed by watching live cmake --trace output and Ctrl-C it whenever a particular line takes too long... (poor man's profiler ;)

An improved variant would be to run it through a Linux strace with relative timestamps enabled: "strace -r cmake --trace ."

A full "strace -r cmake --trace . 2>&1|sort -r|less" might be quite insightful as well.

Pinpoint expensive operations

Watch out for repeated or wasteful execute_process() in a --trace. This is very expensive (simply temporarily disable to compare), may even cause doubled configure-time execution time. Try moving as many operations from configure-time to build time (in the case of execute_process(), use a build-time add_custom_commmand() instead).

Use LIST(APPEND ...)

There are two ways to append values to a variable in CMake:

```
SET(myVar ${myVar} newItem)
```

and since CMake 2.4 there is the new LIST() command:

```
LIST(APPEND myVar newItem)
```

LIST(APPEND ...) is for large lists and appends much faster than using SET().

Reduce add_custom_command()s DEPENDS lists

If your build setup happens to contain many targets which all depend on the same sizeable list of file dependencies, then it might be useful to establish one single custom command (plus its associated target) which DEPENDS on those many files and creates one single OUTPUT "stamp file" ("one of the files changed" watchdog file) which can then be DEPENDS-fed into all affected add_custom_command()s as a single file dependency. A very nice way to figure out whether this applies to your build environment is to do:

```
ninja -t graph > /tmp/graphviz.log
dot -Tsvg /tmp/graphviz.log >/tmp/cmake_ninja.svg
```

and watch the resulting graph monstrosity in awe :)

Use an include guard

Please note that this is an advanced (somewhat cumbersome) item.

For CMake modules (files referenced via include() statement), you could use something like:

```
if(my_module_xyz_included)
  return()
endif(my_module_xyz_included)
set(my_module_xyz_included true)
```

at the beginning of your module file, to avoid repeated parsing within sibling scopes (sub directories, etc.), which also cuts down on amount of

```
cmake --trace
```

log traffic.

Note that the type of the include guard variable should never be a CACHE variable, since this would cause the variable to persist across multiple CMake configure runs, thereby causing the bug of blocking any subsequent run from reading in the module file's content. A Non-CACHE variable is not fully appropriate either, however, since foreign scopes (which did not adopt that variable definition yet) will lead to (somewhat unnecessary and potentially problematic) re-reading of the file's content. Thus the best choice is implementing this check via a GLOBAL property setting, since such settings have all of the desired characteristics: they're both globally valid and single-session only.

Second, note that only module files that are supportive of such a construct (i.e., files declaring only functions or defining all their settings as CACHE variables) can make use of include guards without issues (Non-CACHE variables, while possibly being referenced within those functions, may suddenly end up being out of scope yet the function will remain reachable).

Conditional find_package()

Some other part may already have queried this package and thus caused the corresponding CACHE variable to have been set. find_package() is quite expensive, and AFAIK this yields some nice speedup. This might be questionable, though, in case of changing requirements/requested configurations between project units (but in that case you'd probably have a conflict anyway since there's only a single CACHE variable involved).

```
if(NOT xyz_EXECUTABLE)
  find_package(xyz REQUIRED)
endif(NOT xyz_EXECUTABLE)
```

Split modules into functions/definitions

As a general hint, it might be useful to split module files into containing either clean stateless non-specific (generic) helper functions or content which defines specific settings and calls some helper functions.

Conditional switching of (dummy) methods

Rather than doing a more costly

```
function(foobar)
  if(have_feature)
    if(is_ok)
...
endfunction(foobar)
```

it may be useful to make use of such a conditional to do a whole-function switch instead:

```
if(have_feature)
  function(foobar)
    (large implementation here)
  endfunction(foobar)
else(have_feature)
  function(foobar)
    # DUMMY
  endfunction(foobar)
endif(have_feature)
```

, thereby saving on function execution time in case of many repeated invocations. Note that this obviously comes with a static/dynamic tradeoff, however: While the conditional evaluation sitting within the function can obviously react on later changes to the conditional (have_feature in this case), the static-switch method is a one-time decision only.

Loop optimizations

Use these tricks to do an initial match query over the entire list prior to iterating over each element, and return() ASAP. I did not profile it whether these tricks are indeed faster, but for large lists it should be useful.

```
if("${list}" MATCHES ${elem_query}) # shortcut  :)
  foreach(elem ${list})
    if(${elem} STREQUAL ${elem_query})
      set(elem_found true)
      return()/break() # don't forget these...
    endif(${elem} STREQUAL ${elem_query})
  endforeach(elem ${list})
endif("${list}" MATCHES ${elem_query})
```

REGEX can be awfully slow

Note that I'm talking about an incident of an apocalyptically atrocious 12s vs. 4.9s wholesale CMake configure time here!! (FindZLIB.cmake)

Rather than doing a terribly slow combination of file(READ) (read all content) with a subsequent string(REGEX REPLACE) on that full content, do a file(STRINGS) with a REGEX attribute already specified to directly narrow down on actually interesting content, then if needed do a final string(REGEX REPLACE) on the pre-filtered content only.

Retrieved from "https://cmake.org/Wiki/index.php?title=CMake_Performance_Tips&oldid=51693"
Category: CMake