How to build more compact OpenCV applications on Linux

The OpenCV library can be built in two variants: dynamic (shared libraries) and static (archives). Default mode on most platforms is *dynamic* and to switch to another mode one can turn `BUILD_SHARED_LIBS` cmake option off.

Both modes have their own use cases, advantages and limitations:

Shared libraries (`-DBUILD_SHARED_LIBS=ON`):

- aplication linked to the library will contain references to functions and data, so its size will be very small
- several applications can reuse the library
- application should be able to find and load the library and all dependencies (for example *libpng*)
- all dependencies should be compatible with the library

Static libraries (`-DBUILD_SHARED_LIBS=OFF`):

- application binary will contain all functionality which it uses (or even more), size will be greater
- several applications can not reuse the library, each will have its own copy
- application will have less runtime dependencies
- some embedded platforms only support this variant
- more details: https://en.wikipedia.org/wiki/Static_library

We will take a look at some of GCC and Clang compilation options and investigate their influence on application binaries linked with statically built OpenCV library.

Basic

Reduce symbol visibility

Compiler options: `-fvisibility=hidden` , `-fvisibility-inlines=hidden`

Function attributes: `__attribute__ ((visibility ("hidden")))`

OpenCV option: *none* - enabled by default

These flags were enabled by default in OpenCV for a long time but before merging PR #8198 all exported symbols had been marked visible via function attribute in all build modes. Due to this fact linker considered such symbols "needed" and was not able to remove them from the application.

> *:grey_exclamation:* ***Note:*** *Android build with enabled* `BUILD_FAT_JAVA_LIB` *option will force marking all exported symbols visible even if the build is static. This is needed to export symbols from the opencv_java.so which is produced by combining all static* `libopencv_*.a` *archives and compiled JNI wrappers together.*

Read more:

- https://gcc.gnu.org/wiki/Visibility
- https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/CppRuntimeEnv/Articles/SymbolVisibility.html

Split into smaller sections

Compiler options: `-ffunction-sections` , `-fdata-sections`

OpenCV option: *none* - enabled by default

By default compiler puts all symbols into big *.text*, *.data*, etc sections. With these options each function and data block will be placed in its own section (for example *.text.SomeSymbol*) allowing linker to remove it from resulting binary.

Garbage collect unused sections

Compiler option: `-Wl,--gc-sections` (or `--gc-sections` if linker is called directly)

OpenCV option: *none* - enabled for all executables by default

Tells linker to remove unused sections. Without this option, two previous compilation options will not have visible effect on the binary size.

Read more:

- https://sourceware.org/binutils/docs/ld/Options.html

Advanced

Link time optimization

Compiler option: `-flto`

OpenCV option: `ENABLE_LTO` (*OFF* by default)

In this mode the compiler will produce object files containing intermediate code instead of machine code. This representation then will be used by the linker during final application link stage to exclude non executed branches.

For example, function `cv::cvtColor` can perform variety of conversions from `BGR2Gray` to more complex `BayerBG2BGR` and application which uses this function will need all of the underlying implementations. With link time optimization it is possible to determine all modes which can be passed to the `cvtColor` in this concrete binary and to leave only those implementations which are actually used.

These flags should be provided during library and application building and linking. Static libraries produced in this mode can be less portable, you can find more details in the corresponding compiler documentation section.

Read more:

- https://gcc.gnu.org/wiki/LinkTimeOptimization
- https://llvm.org/docs/LinkTimeOptimization.html

Optimize for size

Compiler option: `-Os`

OpenCV option: *none* - not supported in mainline

Compiler will use optimizations targeted for size reduction at the cost of application performance.

Read more:

- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options

Results

We will measure total size of non-stripped C++ samples (~80 samples in total) produced during OpenCV build (with `-DBUILD_EXAMPLES=ON` option): `du -hc build/bin/cpp-example-* | grep total`

> *:grey_exclamation:* ***Note:*** *binaries size depends on the library configuration which in turn depends on the platform configuration and libraries installed in the build environment. Here we will compare x86_64 builds without CUDA, OpenCL and IPP:* `-DWITH_CUDA=OFF`, `-DWITH_OPENCL=OFF`, `-DWITH_IPP=OFF`.

Experi-ment	Hide symbols	Function sections	GC sections	LTO, size	Size (MiB)	Relative (%)
1	:x:	:x:	:x:		413	100
2	:x:	:white_check_mark:	:white_check_mark:		405	98
3	:white_check_mark:	:x:	:x:		413	100
4	:white_check_mark:	:white_check_mark:	:x:		413	100
5	:white_check_mark:	:x:	:white_check_mark:		412	100
6	:white_check_mark:	:white_check_mark:	:white_check_mark:		**243**	59
7	:x:	:x:	:x:	LTO	386	93
8	:white_check_mark:	:white_check_mark:	:white_check_mark:	LTO	**192**	46
9	:x:	:x:	:x:	size	272	66
10	:white_check_mark:	:white_check_mark:	:white_check_mark:	size	163	39
11	:white_check_mark:	:white_check_mark:	:white_check_mark:	both	**130**	31

As can be seen in the result table, all three conditions described in the *Basic* section should be met in order to achieve size reduction (#6 vs ##1-5). LTO allows to decrease the size even further at the cost of build time (#8). If the execution speed is not important, the binaries can be compressed to one third of the base size by using the `-Os` compiler flag instead of `-O3` (#10 and #11).