NVIDIA CUDA Compiler Driver

Release 12.0

NVIDIA

Jan 27, 2023

Contents

1 Overview 3
- 1.1 CUDA Programming Model 3
- 1.2 CUDA Sources 3
- 1.3 Purpose of NVCC 4

2 Supported Host Compilers 5

3 Compilation Phases 7
- 3.1 NVCC Identification Macro 7
- 3.2 NVCC Phases 7
- 3.3 Supported Input File Suffixes 8
- 3.4 Supported Phases 8

4 The CUDA Compilation Trajectory 11

5 NVCC Command Options 13
- 5.1 Command Option Types and Notation 13
- 5.2 Command Option Description 14
  - 5.2.1 File and Path Specifications 14
    - `--output-file file` (-o) 14
    - `--objdir-as-tempdir` (-objtemp) 14
    - `--pre-include file,...` (-include) 14
    - `--library library,...` (-l) 14
    - `--define-macro def,...` (-D) 14
    - `--undefine-macro def,...` (-U) 15
    - `--include-path path,...` (-I) 15
    - `--system-include path,...` (-isystem) 15
    - `--library-path path,...` (-L) 15
    - `--output-directory directory` (-odir) 15
    - `--dependency-output file` (-MF) 15
    - `--generate-dependency-targets` (-MP) 15
    - `--compiler-bindir directory` (-ccbin) 16
    - `--allow-unsupported-compiler` (-allow-unsupported-compiler) 16
    - `--archiver-binary executable` (-arbin) 16
    - `--cudart` {none|shared|static} (-cudart) 16
    - `--cudadevrt` {none|static} (-cudadevrt) 16
    - `--libdevice-directory directory` (-ldir) 17
    - `--target-directory string` (-target-dir) 17
  - 5.2.2 Options for Specifying the Compilation Phase 17
    - `--link` (-link) 17
    - `--lib` (-lib) 17
    - `--device-link` (-dlink) 17

--device-c (-dc) 18
--device-w (-dw) 18
--cuda (-cuda) 18
--compile (-c) 18
--fatbin (-fatbin) 18
--cubin (-cubin) 19
--ptx (-ptx) 19
--preprocess (-E) 19
--generate-dependencies (-M) 19
--generate-nonsystem-dependencies (-MM) 19
--generate-dependencies-with-compile (-MD) 20
--generate-nonsystem-dependencies-with-compile (-MMD) 20
--optix-ir (-optix-ir) 20
--run (-run) 20
5.2.3 Options for Specifying Behavior of Compiler/Linker 20
--profile (-pg) 20
--debug (-g) 21
--device-debug (-G) 21
--extensible-whole-program (-ewp) 21
--no-compress (-no-compress) 21
--generate-line-info (-lineinfo) 21
--optimization-info kind,... (-opt-info) 21
--optimize level (-O) 21
--dopt kind (-dopt) 22
--dlink-time-opt (-dlto) 22
--ftemplate-backtrace-limit limit (-ftemplate-backtrace-limit) . . 22
--ftemplate-depth limit (-ftemplate-depth) 22
--no-exceptions (-noeh) 22
--shared (-shared) 23
--x {c|c++|cu} (-x) 23
--std {c++03|c++11|c++14|c++17|c++20} (-std) 23
--no-host-device-initializer-list (-nohdinitlist) 23
--expt-relaxed-constexpr (-expt-relaxed-constexpr) 24
--extended-lambda (-extended-lambda) 24
--expt-extended-lambda (-expt-extended-lambda) 24
--machine {64} (-m) 24
--m64 (-m64) 24
--host-linker-script {use-lcs|gen-lcs} (-hls) 24
--augment-host-linker-script (-aug-hls) 25
--host-relocatable-link (-r) 25
5.2.4 Options for Passing Specific Phase Options 25
--compiler-options options,... (-Xcompiler) 25
--linker-options options,... (-Xlinker) 25
--archive-options options,... (-Xarchive) 26
--ptxas-options options,... (-Xptxas) 26
--nvlink-options options,... (-Xnvlink) 26
5.2.5 Options for Guiding the Compiler Driver 26
--forward-unknown-to-host-compiler (-forward-unknown-to-host-compiler) 26
--forward-unknown-to-host-linker (-forward-unknown-to-host-linker) 26
--dont-use-profile (-noprof) 27
--threads number (-t) 27
--dryrun (-dryrun) 27
--verbose (-v) 27
--keep (-keep) 27

--keep-dir directory (-keep-dir) . . . 27
--save-temps (-save-temps) . . . 27
--clean-targets (-clean) . . . 27
--run-args arguments,... (-run-args) . . . 28
--use-local-env (-use-local-env) . . . 28
--input-drive-prefix prefix (-idp) . . . 28
--dependency-drive-prefix prefix (-ddp) . . . 28
--drive-prefix prefix (-dp) . . . 28
--dependency-target-name target (-MT) . . . 28
--no-align-double . . . 29
--no-device-link (-nodlink) . . . 29
--allow-unsupported-compiler (-allow-unsupported-compiler) . . . 29
5.2.6 Options for Steering CUDA Compilation . . . 29
--default-stream {legacy|null|per-thread} (-default-stream) . . . 29
5.2.7 Options for Steering GPU Code Generation . . . 29
--gpu-architecture {arch|native|all|all-major} (-arch) . . . 29
--gpu-code code,... (-code) . . . 30
--generate-code specification (-gencode) . . . 31
--relocatable-device-code {true|false} (-rdc) . . . 31
--entries entry,... (-e) . . . 31
--maxrregcount amount (-maxrregcount) . . . 31
--use_fast_math (-use_fast_math) . . . 32
--ftz {true|false} (-ftz) . . . 32
--prec-div {true|false} (-prec-div) . . . 32
--prec-sqrt {true|false} (-prec-sqrt) . . . 32
--fmad {true|false} (-fmad) . . . 33
--extra-device-vectorization (-extra-device-vectorization) . . . 33
--compile-as-tools-patch (-astoolspatch) . . . 33
--keep-device-functions (-keep-device-functions) . . . 33
5.2.8 Generic Tool Options . . . 33
--disable-warnings (-w) . . . 33
--source-in-ptx (-src-in-ptx) . . . 34
--restrict (-restrict) . . . 34
--Wno-deprecated-gpu-targets (-Wno-deprecated-gpu-targets) . . . 34
--Wno-deprecated-declarations (-Wno-deprecated-declarations) . . . 34
--Wreorder (-Wreorder) . . . 34
--Wdefault-stream-launch (-Wdefault-stream-launch) . . . 34
--Wmissing-launch-bounds (-Wmissing-launch-bounds) . . . 34
--Wext-lambda-captures-this (-Wext-lambda-captures-this) . . . 34
--Werror kind,... (-Werror) . . . 34
--display-error-number (-err-no) . . . 35
--no-display-error-number (-no-err-no) . . . 35
--diag-error errNum,... (-diag-error) . . . 35
--diag-suppress errNum,... (-diag-suppress) . . . 35
--diag-warn errNum,... (-diag-warn) . . . 36
--resource-usage (-res-usage) . . . 36
--help (-h) . . . 36
--version (-V) . . . 36
--options-file file,... (-optf) . . . 36
--time filename (-time) . . . 36
--qpp-config config (-qpp-config) . . . 36
--list-gpu-code (-code-ls) . . . 36
--list-gpu-arch (-arch-ls) . . . 37
5.2.9 Phase Options . . . 37

Ptxas Options . . . 37
--allow-expensive-optimizations (-allow-expensive-optimizations) 37
--compile-only (-c) . . . 37
--def-load-cache (-dlcm) . . . 37
--def-store-cache (-dscm) . . . 37
--device-debug (-g) . . . 37
--disable-optimizer-constants (-disable-optimizer-consts) . . 38
--entry entry,... (-e) . . . 38
--fmad (-fmad) . . . 38
--force-load-cache (-flcm) . . . 38
--force-store-cache (-fscm) . . . 38
--generate-line-info (-lineinfo) . . . 38
--gpu-name gpuname (-arch) . . . 38
--help (-h) . . . 39
--machine (-m) . . . 39
--maxrregcount amount (-maxrregcount) . . . 39
--opt-level N (-O) . . . 39
--options-file file,... (-optf) . . . 39
--position-independent-code (-pic) . . . 39
--preserve-relocs (-preserve-relocs) . . . 39
--sp-bound-check (-sp-bound-check) . . . 39
--verbose (-v) . . . 40
--version (-V) . . . 40
--warning-as-error (-Werror) . . . 40
--warn-on-double-precision-use (-warn-double-usage) . . . 40
--warn-on-local-memory-usage (-warn-lmem-usage) . . . 40
--warn-on-spills (-warn-spills) . . . 40
--compile-as-tools-patch (-astoolspatch) . . . 40
NVLINK Options . . . 40
--disable-warnings (-w) . . . 41
--preserve-relocs (-preserve-relocs) . . . 41
--verbose (-v) . . . 41
--warning-as-error (-Werror) . . . 41
--suppress-arch-warning (-suppress-arch-warning) . . . 41
--suppress-stack-size-warning (-suppress-stack-size-warning) 41
--dump-callgraph (-dump-callgraph) . . . 41
5.3 NVCC Environment Variables . . . 41

6 GPU Compilation 43
6.1 GPU Generations . . . 43
6.2 GPU Feature List . . . 43
6.3 Application Compatibility . . . 44
6.4 Virtual Architectures . . . 44
6.5 Virtual Architecture Feature List . . . 46
6.6 Further Mechanisms . . . 46
6.6.1 Just-in-Time Compilation . . . 46
6.6.2 Fatbinaries . . . 48
6.7 NVCC Examples . . . 48
6.7.1 Base Notation . . . 48
6.7.2 Shorthand . . . 48
Shorthand 1 . . . 48
Shorthand 2 . . . 49
Shorthand 3 . . . 49
6.7.3 Extended Notation . . . 49

6.7.4 Virtual Architecture Macros 50

7 Using Separate Compilation in CUDA 51

7.1 Code Changes for Separate Compilation 51
7.2 NVCC Options for Separate Compilation 51
7.3 Libraries 52
7.4 Examples 52
7.5 Optimization Of Separate Compilation 54
7.6 Potential Separate Compilation Issues 54
7.6.1 Object Compatibility 54
7.6.2 JIT Linking Support 55
7.6.3 Implicit CUDA Host Code 55
7.6.4 Using __CUDA_ARCH__ 55
7.6.5 Device Code in Libraries 56

8 Miscellaneous NVCC Usage 57

8.1 Cross Compilation 57
8.2 Keeping Intermediate Phase Files 57
8.3 Cleaning Up Generated Files 57
8.4 Printing Code Generation Statistics 58

9 Notices 59

9.1 Notice 59
9.2 OpenCL 60
9.3 Trademarks 60
9.4 Copyright 60

NVIDIA CUDA Compiler Driver NVCC

The documentation for nvcc, the CUDA compiler driver.

Chapter 1. Overview

1.1. CUDA Programming Model

The CUDA Toolkit targets a class of applications whose control part runs as a process on a general purpose computing device, and which use one or more NVIDIA GPUs as coprocessors for accelerating *single program, multiple data* (SPMD) parallel jobs. Such jobs are self-contained, in the sense that they can be executed and completed by a batch of GPU threads entirely without intervention by the host process, thereby gaining optimal benefit from the parallel graphics hardware.

The GPU code is implemented as a collection of functions in a language that is essentially C++, but with some annotations for distinguishing them from the host code, plus annotations for distinguishing different types of data memory that exists on the GPU. Such functions may have parameters, and they can be called using a syntax that is very similar to regular C function calling, but slightly extended for being able to specify the matrix of GPU threads that must execute the called function. During its life time, the host process may dispatch many parallel GPU tasks.

For more information on the CUDA programming model, consult the CUDA C++ Programming Guide.

1.2. CUDA Sources

Source files for CUDA applications consist of a mixture of conventional C++ host code, plus GPU device functions. The CUDA compilation trajectory separates the device functions from the host code, compiles the device functions using the proprietary NVIDIA compilers and assembler, compiles the host code using a C++ host compiler that is available, and afterwards embeds the compiled GPU functions as fatbinary images in the host object file. In the linking stage, specific CUDA runtime libraries are added for supporting remote SPMD procedure calling and for providing explicit GPU manipulation such as allocation of GPU memory buffers and host-GPU data transfer.

1.3. Purpose of NVCC

The compilation trajectory involves several splitting, compilation, preprocessing, and merging steps for each CUDA source file. It is the purpose of nvcc, the CUDA compiler driver, to hide the intricate details of CUDA compilation from developers. It accepts a range of conventional compiler options, such as for defining macros and include/library paths, and for steering the compilation process. All non-CUDA compilation steps are forwarded to a C++ host compiler that is supported by nvcc, and nvcc translates its options to appropriate host compiler command line options.

Chapter 2. Supported Host Compilers

A general purpose C++ host compiler is needed by nvcc in the following situations:

- During non-CUDA phases (except the run phase), because these phases will be forwarded by nvcc to this compiler.
- During CUDA phases, for several preprocessing stages and host code compilation (see also The CUDA Compilation Trajectory).

nvcc assumes that the host compiler is installed with the standard method designed by the compiler provider. If the host compiler installation is non-standard, the user must make sure that the environment is set appropriately and use relevant nvcc compile options.

The following documents provide detailed information about supported host compilers:

- NVIDIA CUDA Installation Guide for Linux
- NVIDIA CUDA Installation Guide for Microsoft Windows

On all platforms, the default host compiler executable (gcc and g++ on Linux and cl.exe on Windows) found in the current execution search path will be used, unless specified otherwise with appropriate options (see File and Path Specifications).

Chapter 3. Compilation Phases

3.1. NVCC Identification Macro

nvcc predefines the following macros:

__NVCC__ Defined when compiling C/C++/CUDA source files.

__CUDACC__ Defined when compiling CUDA source files.

__CUDACC_RDC__ Defined when compiling CUDA source files in relocatable device code mode (see NVCC Options for Separate Compilation).

__CUDACC_EWP__ Defined when compiling CUDA source files in extensible whole program mode (see Options for Specifying Behavior of Compiler/Linker).

__CUDACC_DEBUG__ Defined when compiling CUDA source files in the device-debug mode (see Options for Specifying Behavior of Compiler/Linker).

__CUDACC_RELAXED_CONSTEXPR__ Defined when the `--expt-relaxed-constexpr` flag is specified on the command line. Refer to the CUDA C++ Programming Guide for more details.

__CUDACC_EXTENDED_LAMBDA__ Defined when the `--expt-extended-lambda` or `--extended-lambda` flag is specified on the command line. Refer to the CUDA C++ Programming Guide for more details.

__CUDACC_VER_MAJOR__ Defined with the major version number of nvcc.

__CUDACC_VER_MINOR__ Defined with the minor version number of nvcc.

__CUDACC_VER_BUILD__ Defined with the build version number of nvcc.

__NVCC_DIAG_PRAGMA_SUPPORT__ Defined when the CUDA frontend compiler supports diagnostic control with the `nv_diag_suppress`, `nv_diag_error`, `nv_diag_warning`, `nv_diag_default`,`nv_diag_once`, and `nv_diagnostic` pragmas.

3.2. NVCC Phases

A compilation phase is the a logical translation step that can be selected by command line options to nvcc. A single compilation phase can still be broken up by nvcc into smaller steps, but these smaller steps are just implementations of the phase: they depend on seemingly arbitrary capabilities of the internal tools that nvcc uses, and all of these internals may change with a new release of the CUDA Toolkit. Hence, only compilation phases are stable across releases, and although nvcc provides options to display the compilation steps that it executes, these are for debugging purposes only and must not be copied and used into build scripts.

nvcc phases are selected by a combination of command line options and input file name suffixes, and the execution of these phases may be modified by other command line options. In phase selection, the input file suffix defines the phase input, while the command line option defines the required output of the phase.

The following paragraphs list the recognized file name suffixes and the supported compilation phases. A full explanation of the nvcc command line options can be found in NVCC Command Options.

3.3. Supported Input File Suffixes

The following table defines how nvcc interprets its input files:

Input File Prefix	Description
.cu	CUDA source file, containing host code and device functions
.c	C source file
.cc, .cxx, .cpp	C++ source file
.ptx	PTX intermediate assembly file (see Figure 1)
.cubin	CUDA device code binary file (CUBIN) for a single GPU architecture (see Figure 1)
.fatbin	CUDA fat binary file that may contain multiple PTX and CUBIN files (see Figure 1)
.o, .obj	Object file
.a, .lib	Library file
.res	Resource file
.so	Shared object file

Note that nvcc does not make any distinction between object, library or resource files. It just passes files of these types to the linker when the linking phase is executed.

3.4. Supported Phases

The following table specifies the supported compilation phases, plus the option to nvcc that enables execution of this phase. It also lists the default name of the output file generated by this phase, which will take effect when no explicit output file name is specified using option --output-file:

Phase	nvcc Option		Default Output File Name
	Long Name	Short Name	
CUDA compilation to C/C++ source file	`--cuda`	`-cuda`	`.cpp.ii` appended to source file name, as in `x.cu.cpp.ii`. This output file can be compiled by the host compiler that was used by `nvcc` to preprocess the `.cu` file.
C/C++ preprocessing	`--preprocess`	`-E`	*<result on standard output>*
C/C++ compilation to object file	`--compile`	`-c`	Source file name with suffix replaced by `o` on Linux or `obj` on Windows
Cubin generation from CUDA source files	`--cubin`	`-cubin`	Source file name with suffix replaced by `cubin`
Cubin generation from PTX intermediate files.	`--cubin`	`-cubin`	Source file name with suffix replaced by `cubin`
PTX generation from CUDA source files	`--ptx`	`-ptx`	Source file name with suffix replaced by `ptx`
Fatbinary generation from source, PTX or cubin files	`--fatbin`	`-fatbin`	Source file name with suffix replaced by `fatbin`
Linking relocatable device code.	`--device-link`	`-dlink`	`a_dlink.obj` on Windows or `a_dlink.o` on other platforms
Cubin generation from linked relocatable device code.	`--device-link --cubin`	`-dlink -cubin`	`a_dlink.cubin`
Fatbinary generation from linked relocatable device code	`--device-link --fatbin`	`-dlink -fatbin`	`a_dlink.fatbin`
Linking an executable	*<no phase option>*		`a.exe` on Windows or `a.out` on other platforms
Constructing an object file archive, or library	`--lib`	`-lib`	`a.lib` on Windows or `a.a` on other platforms
`make` dependency generation	`--generate-dependencies`	`-M`	*<result on standard output>*
`make` dependency generation without headers in system paths.	`--generate-nonsystem-dependencies`	`-MM`	*<result on standard output>*
Compile CUDA source to OptiX IR output.	`--optix-ir`	`-optix-ir`	Source file name with suffix replaced by `optixir`
Running an executable	`--run`	`-run`	

Notes:

- The last phase in this list is more of a convenience phase. It allows running the compiled and linked executable without having to explicitly set the library path to the CUDA dynamic libraries.
- Unless a phase option is specified, nvcc will compile and link all its input files.

Chapter 4. The CUDA Compilation Trajectory

CUDA compilation works as follows: the input program is preprocessed for device compilation compilation and is compiled to CUDA binary (`cubin`) and/or PTX intermediate code, which are placed in a fatbinary. The input program is preprocessed once again for host compilation and is synthesized to embed the fatbinary and transform CUDA specific C++ extensions into standard C++ constructs. Then the C++ host compiler compiles the synthesized host code with the embedded fatbinary into a host object. The exact steps that are followed to achieve this are displayed in Figure 1.

The embedded fatbinary is inspected by the CUDA runtime system whenever the device code is launched by the host program to obtain an appropriate fatbinary image for the current GPU.

CUDA programs are compiled in the whole program compilation mode by default, i.e., the device code cannot reference an entity from a separate file. In the whole program compilation mode, device link steps have no effect. For more information on the separate compilation and the whole program compilation, see Using Separate Compilation in CUDA.

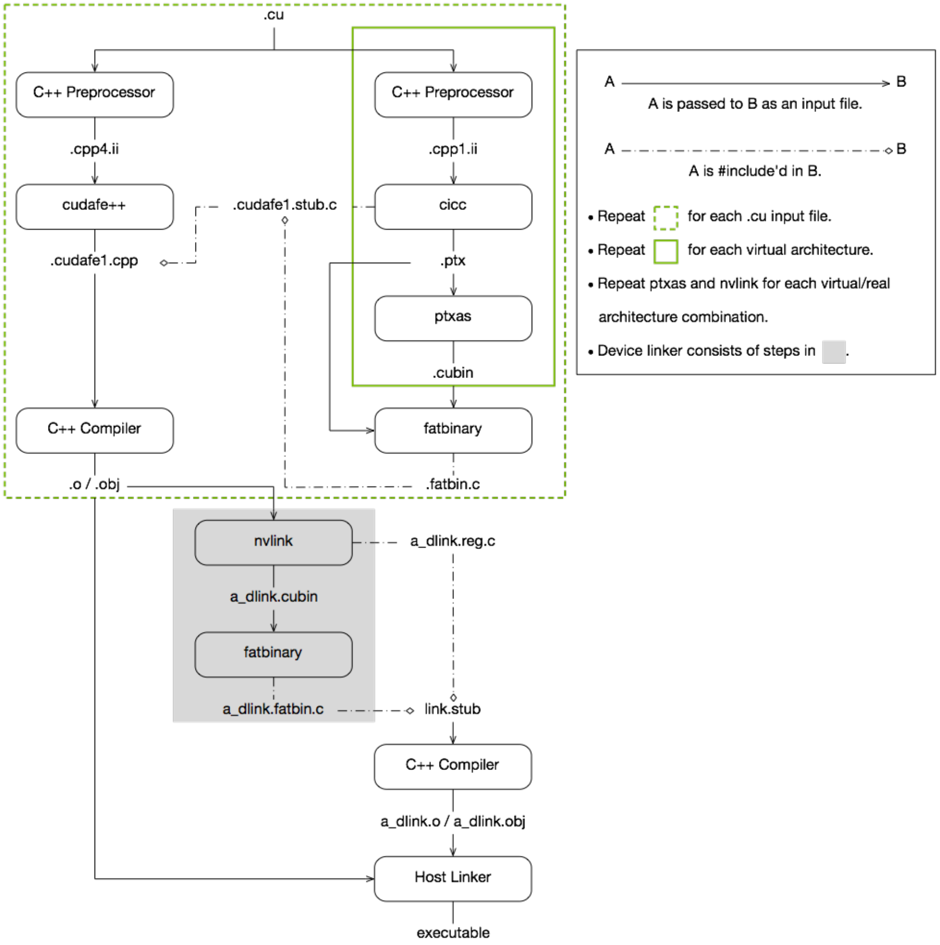

Fig. 1: CUDA Compilation Trajectory

Chapter 5. NVCC Command Options

5.1. Command Option Types and Notation

Each nvcc option has a long name and a short name, which are interchangeable with each other. These two variants are distinguished by the number of hyphens that must precede the option name: long names must be preceded by two hyphens, while short names must be preceded by a single hyphen. For example, -I is the short name of --include-path. Long options are intended for use in build scripts, where size of the option is less important than descriptive value. In contrast, short options are intended for interactive use.

nvcc recognizes three types of command options: boolean options, single value options, and list options.

Boolean options do not have an argument; they are either specified on a command line or not. Single value options must be specified at most once, and list options may be repeated. Examples of each of these option types are, respectively: --verbose (switch to verbose mode), --output-file (specify output file), and --include-path (specify include path).

Single value options and list options must have arguments, which must follow the name of the option itself by either one of more spaces or an equals character. When a one-character short name such as -I, -l, and -L is used, the value of the option may also immediately follow the option itself without being seperated by spaces or an equal character. The individual values of list options may be separated by commas in a single instance of the option, or the option may be repeated, or any combination of these two cases.

Hence, for the two sample options mentioned above that may take values, the following notations are legal:

```
-o file
```

```
-o=file
```

```
-Idir1,dir2 -I=dir3 -I dir4,dir5
```

Long option names are used throughout the document, unless specified otherwise; however, short names can be used instead of long names to have the same effect.

5.2. Command Option Description

This section presents tables of nvcc options. The option type in the tables can be recognized as follows: boolean options do not have arguments specified in the first column, while the other two types do. List options can be recognized by the repeat indicator `,...` at the end of the argument.

Long options are described in the first columns of the options tables, and short options occupy the second columns.

5.2.1. File and Path Specifications

`--output-file file` (`-o`)

Specify name and location of the output file.

`--objdir-as-tempdir` (`-objtemp`)

Create all intermediate files in the same directory as the object file. These intermediate files are deleted when the compilation is finished. This option will take effect only if -c, -dc or -dw is also used. Using this option will ensure that the intermediate file name that is embedded in the object file will not change in multiple compiles of the same file. However, this is not guaranteed if the input is stdin. If the same file is compiled with two different options, ex., 'nvcc -c t.cu' and 'nvcc -c -ptx t.cu', then the files should be compiled in different directories. Compiling them in the same directory can either cause the compilation to fail or produce incorrect results.

`--pre-include file,...` (`-include`)

Specify header files that must be pre-included during preprocessing.

`--library library,...` (`-l`)

Specify libraries to be used in the linking stage without the library file extension.

The libraries are searched for on the library search paths that have been specified using option `--library-path` (see Libraries).

`--define-macro def,...` (`-D`)

Define macros to be used during preprocessing.

def can be either *name* or *name=definition*.

- *name* - Predefine *name* as a macro.
- *name=definition* - The contents of *definition* are tokenized and preprocessed as if they appear during translation phase three in a `#define` directive. The definition will be truncated by embedded new line characters.

`--undefine-macro def,... (-U)`

Undefine an existing macro during preprocessing or compilation.

`--include-path path,... (-I)`

Specify include search paths.

`--system-include path,... (-isystem)`

Specify system include search paths.

`--library-path path,... (-L)`

Specify library search paths (see Libraries).

`--output-directory directory (-odir)`

Specify the directory of the output file.

This option is intended for letting the dependency generation step (see `--generate-dependencies`) generate a rule that defines the target object file in the proper directory.

`--dependency-output file (-MF)`

Specify the dependency output file.

This option specifies the output file for the dependency generation step (see `--generate-dependencies`). The option `--generate-dependencies` or `--generate-nonystem-dependencies` must be specified if a dependency output file is set.

`--generate-dependency-targets (-MP)`

Add an empty target for each dependency.

This option adds phony targets to the dependency generation step (see `--generate-dependencies`) intended to avoid makefile errors if old dependencies are deleted. The input files are not emitted as phony targets.

`--compiler-bindir directory` (-ccbin)

Specify the directory in which the default host compiler executable resides.

The host compiler executable name can be also specified to ensure that the correct host compiler is selected. In addition, driver prefix options (`--input-drive-prefix`, `--dependency-drive-prefix`, or `--drive-prefix`) may need to be specified, if `nvcc` is executed in a Cygwin shell or a MinGW shell on Windows.

`--allow-unsupported-compiler` (-allow-unsupported-compiler)

Disable nvcc check for supported host compiler versions.

Using an unsupported host compiler may cause compilation failure or incorrect run time execution. Use at your own risk. This option has no effect on MacOS.

`--archiver-binary executable` (-arbin)

Specify the path of the archiver tool used create static library with `--lib`.

`--cudart` {none|shared|static} (-cudart)

Specify the type of CUDA runtime library to be used: no CUDA runtime library, shared/dynamic CUDA runtime library, or static CUDA runtime library.

Allowed Values

- `none`
- `shared`
- `static`

Default

The static CUDA runtime library is used by default.

`--cudadevrt` {none|static} (-cudadevrt)

Specify the type of CUDA device runtime library to be used: no CUDA device runtime library, or static CUDA device runtime library.

Allowed Values

- `none`
- `static`

Default

The static CUDA device runtime library is used by default.

`--libdevice-directory directory` (`-ldir`)

Specify the directory that contains the libdevice library files.

Libdevice library files are located in the `nvvm/libdevice` directory in the CUDA Toolkit.

`--target-directory string` (`-target-dir`)

Specify the subfolder name in the targets directory where the default include and library paths are located.

5.2.2. Options for Specifying the Compilation Phase

Options of this category specify up to which stage the input files must be compiled.

`--link` (`-link`)

Specify the default behavior: compile and link all input files.

Default Output File Name

`a.exe` on Windows or `a.out` on other platforms is used as the default output file name.

`--lib` (`-lib`)

Compile all input files into object files, if necessary, and add the results to the specified library output file.

Default Output File Name

`a.lib` on Windows or `a.a` on other platforms is used as the default output file name.

`--device-link` (`-dlink`)

Link object files with relocatable device code and `.ptx`, `.cubin`, and `.fatbin` files into an object file with executable device code, which can be passed to the host linker.

Default Output File Name

`a_dlink.obj` on Windows or `a_dlink.o` on other platforms is used as the default output file name. When this option is used in conjunction with `--fatbin`, `a_dlink.fatbin` is used as the default output file name. When this option is used in conjunction with `--cubin`, `a_dlink.cubin` is used as the default output file name.

--device-c (-dc)

Compile each .c, .cc, .cpp, .cxx, and .cu input file into an object file that contains relocatable device code.

It is equivalent to --relocatable-device-code=true --compile.

Default Output File Name

The source file name extension is replaced by .obj on Windows and .o on other platforms to create the default output file name. For example, the default output file name for x.cu is x.obj on Windows and x.o on other platforms.

--device-w (-dw)

Compile each .c, .cc, .cpp, .cxx, and .cu input file into an object file that contains executable device code.

It is equivalent to --relocatable-device-code=false --compile.

Default Output File Name

The source file name extension is replaced by .obj on Windows and .o on other platforms to create the default output file name. For example, the default output file name for x.cu is x.obj on Windows and x.o on other platforms.

--cuda (-cuda)

Compile each .cu input file to a .cu.cpp.ii file.

Default Output File Name

.cu.cpp.ii is appended to the basename of the source file name to create the default output file name. For example, the default output file name for x.cu is x.cu.cpp.ii.

--compile (-c)

Compile each .c, .cc, .cpp, .cxx, and .cu input file into an object file.

Default Output File Name

The source file name extension is replaced by .obj on Windows and .o on other platforms to create the default output file name. For example, the default output file name for x.cu is x.obj on Windows and x.o on other platforms.

--fatbin (-fatbin)

Compile all .cu, .ptx, and .cubin input files to device-only .fatbin files.

nvcc discards the host code for each .cu input file with this option.

Default Output File Name

The source file name extension is replaced by .fatbin to create the default output file name. For example, the default output file name for x.cu is x.fatbin.

--cubin (-cubin)

Compile all .cu and .ptx input files to device-only .cubin files.

nvcc discards the host code for each .cu input file with this option.

Default Output File Name

The source file name extension is replaced by .cubin to create the default output file name. For example, the default output file name for x.cu is x.cubin.

--ptx (-ptx)

Compile all .cu input files to device-only .ptx files.

nvcc discards the host code for each .cu input file with this option.

Default Output File Name

The source file name extension is replaced by .ptx to create the default output file name. For example, the default output file name for x.cu is x.ptx.

--preprocess (-E)

Preprocess all .c, .cc, .cpp, .cxx, and .cu input files.

Default Output File Name

The output is generated in *stdout* by default.

--generate-dependencies (-M)

Generate a dependency file that can be included in a Makefile for the .c, .cc, .cpp, .cxx, and .cu input file.

nvcc uses a fixed prefix to identify dependencies in the preprocessed file ('#line 1' on Linux and '# 1' on Windows). The files mentioned in source location directives starting with this prefix will be included in the dependency list.

Default Output File Name

The output is generated in *stdout* by default.

--generate-nonsystem-dependencies (-MM)

Same as --generate-dependencies but skip header files found in system directories (Linux only).

Default Output File Name

The output is generated in *stdout* by default.

--generate-dependencies-with-compile (-MD)

Generate a dependency file and compile the input file. The dependency file can be included in a Makefile for the .c, .cc, .cpp, .cxx, and .cu input file.

This option cannot be specified together with -E. The dependency file name is computed as follows:

- If -MF is specified, then the specified file is used as the dependency file name.
- If -o is specified, the dependency file name is computed from the specified file name by replacing the suffix with '.d'.
- Otherwise, the dependency file name is computed by replacing the input file names's suffix with '.d'.

If the dependency file name is computed based on either -MF or -o, then multiple input files are not supported.

--generate-nonsystem-dependencies-with-compile (-MMD)

Same as --generate-dependencies-with-compile but skip header files found in system directories (Linux only).

--optix-ir (-optix-ir)

Compile CUDA source to OptiX IR (.optixir) output. The OptiX IR is only intended for consumption by OptiX through appropriate APIs. This feature is not supported with link-time-optimization (-dlto), the lto_NN -arch target, or with -gencode.

Default Output File Name

The source file name extension is replaced by .optixir to create the default output file name. For example, the default output file name for x.cu is x.optixir.

--run (-run)

Compile and link all input files into an executable, and executes it.

When the input is a single executable, it is executed without any compilation or linking. This step is intended for developers who do not want to be bothered with setting the necessary environment variables; these are set temporarily by nvcc.

5.2.3. Options for Specifying Behavior of Compiler/Linker

--profile (-pg)

Instrument generated code/executable for use by gprof.

`--debug` (`-g`)

Generate debug information for host code.

`--device-debug` (`-G`)

Generate debug information for device code.

If `--dopt` is not specified, then this option turns off all optimizations on device code. It is not intended for profiling; use `--generate-line-info` instead for profiling.

`--extensible-whole-program` (`-ewp`)

Generate extensible whole program device code, which allows some calls to not be resolved until linking with libcudadevrt.

`--no-compress` (`-no-compress`)

Do not compress device code in fatbinary.

`--generate-line-info` (`-lineinfo`)

Generate line-number information for device code.

`--optimization-info kind,...` (`-opt-info`)

Provide optimization reports for the specified kind of optimization.

The following tags are supported:

`inline`

> Emit remarks related to function inlining. Inlining pass may be invoked multiple times by the compiler and a function not inlined in an earlier pass may be inlined in a subsequent pass.

`--optimize level` (`-O`)

Specify optimization level for host code.

`--dopt kind` (`-dopt`)

Enable device code optimization. When specified along with `-G`, enables limited debug information generation for optimized device code (currently, only line number information). When `-G` is not specified, `-dopt=on` is implicit.

Allowed Values

- on: enable device code optimization.

`--dlink-time-opt` (`-dlto`)

Perform link-time optimization of device code. The option '-lto' is also an alias to '-dlto'. Link-time optimization must be specified at both compile and link time; at compile time it stores high-level intermediate code, then at link time it links together and optimizes the intermediate code. If that intermediate is not found at link time then nothing happens. Intermediate code is also stored at compile time with the `--gpu-code='lto_NN'` target. The options `-dlto -arch=sm_NN` will add a lto_NN target; if you want to only add a lto_NN target and not the compute_NN that `-arch=sm_NN` usually generates, use `-arch=lto_NN`.

`--ftemplate-backtrace-limit limit` (`-ftemplate-backtrace-limit`)

Set the maximum number of template instantiation notes for a single warning or error to limit.

A value of `0` is allowed, and indicates that no limit should be enforced. This value is also passed to the host compiler if it provides an equivalent flag.

`--ftemplate-depth limit` (`-ftemplate-depth`)

Set the maximum instantiation depth for template classes to limit.

This value is also passed to the host compiler if it provides an equivalent flag.

`--no-exceptions` (`-noeh`)

Disable exception handling for host code.

Disable exception handling for host code, by passing "-EHs-c-" (for cl.exe) and "–fno-exceptions" (for other host compilers) during host compiler invocation. These flags are added to the host compiler invocation before any flags passed directly to the host compiler with "-Xcompiler"

Default (on Windows)

- On Windows, nvcc passes /EHsc to the host compiler by default.

Example (on Windows)

- `nvcc --no-exceptions -Xcompiler /EHa x.cu`

--shared (-shared)

Generate a shared library during linking.

Use option `--linker-options` when other linker options are required for more control.

--x {c|c++|cu} (-x)

Explicitly specify the language for the input files, rather than letting the compiler choose a default based on the file name suffix.

Allowed Values

- `c`
- `c++`
- `cu`

Default

The language of the source code is determined based on the file name suffix.

--std {c++03|c++11|c++14|c++17|c++20} (-std)

Select a particular C++ dialect.

Allowed Values

- `c++03`
- `c++11`
- `c++14`
- `c++17`
- `c++20`

Default

The default C++ dialect depends on the host compiler. `nvcc` matches the default C++ dialect that the host compiler uses.

--no-host-device-initializer-list (-nohdinitlist)

Do not consider member functions of `std::initializer_list` as `__host__ __device__` functions implicitly.

--expt-relaxed-constexpr (-expt-relaxed-constexpr)

Experimental flag: *Allow host code to invoke ``__device__ constexpr`` functions, and device code to invoke ``__host__ constexpr`` functions.*

Note that the behavior of this flag may change in future compiler releases.

--extended-lambda (-extended-lambda)

Allow `__host__`, `__device__` annotations in lambda declarations.

--expt-extended-lambda (-expt-extended-lambda)

Alias for `--extended-lambda`.

--machine {64} (-m)

Specify 64-bit architecture.

Allowed Values

- 64

Default

This option is set based on the host platform on which `nvcc` is executed.

--m64 (-m64)

Alias for `--machine=64`

--host-linker-script {use-lcs|gen-lcs} (-hls)

Use the host linker script (GNU/Linux only) to enable support for certain CUDA specific requirements, while building executable files or shared libraries.

Allowed Values

`use-lcs`

Prepares a host linker script and enables host linker to support relocatable device object files that are larger in size, that would otherwise, in certain cases, cause the host linker to fail with relocation truncation error.

`gen-lcs`

Generates a host linker script that can be passed to host linker manually, in the case where host linker is invoked separately outside of nvcc. This option can be combined with `-shared` or `-r` option to generate linker scripts that can be used while generating host shared libraries or host relocatable links respectively.

The file generated using this options must be provided as the last input file to the host linker.

> The output is generated to stdout by default. Use the option -o filename to specify the output filename.

A linker script may already be in used and passed to the host linker using the host linker option --script (or -T), then the generated host linker script must augment the existing linker script. In such cases, the option -aug-hls must be used to generate linker script that contains only the augmentation parts. Otherwise, the host linker behaviour is undefined.

A host linker option, such as -z with a non-default argument, that can modify the default linker script internally, is incompatible with this option and the behavior of any such usage is undefined.

Default Value

use-lcs is used as the default type.

--augment-host-linker-script (-aug-hls)

Enables generation of host linker script that augments an existing host linker script (GNU/Linux only). See option --host-linker-script for more details.

--host-relocatable-link (-r)

When used in combination with -hls=gen-lcs, controls the behaviour of -hls=gen-lcs and sets it to generate host linker script that can be used in host relocatable link (ld -r linkage). See option -hls=gen-lcs for more information.

This option currently is effective only when used with -hls=gen-lcs; in all other cases, this option is ignored currently.

5.2.4. Options for Passing Specific Phase Options

These allow for passing specific options directly to the internal compilation tools that nvcc encapsulates, without burdening nvcc with too-detailed knowledge on these tools. A table of useful sub-tool options can be found at the end of this chapter.

--compiler-options options,... (-Xcompiler)

Specify options directly to the compiler/preprocessor.

--linker-options options,... (-Xlinker)

Specify options directly to the host linker.

`--archive-options options,...` (`-Xarchive`)

Specify options directly to the library manager.

`--ptxas-options options,...` (`-Xptxas`)

Specify options directly to `ptxas`, the PTX optimizing assembler.

`--nvlink-options options,...` (`-Xnvlink`)

Specify options directly to `nvlink`, the device linker.

5.2.5. Options for Guiding the Compiler Driver

`--forward-unknown-to-host-compiler` (`-forward-unknown-to-host-compiler`)

Forward unknown options to the host compiler. An 'unknown option' is a command line argument that starts with `-` followed by another character, and is not a recognized nvcc flag or an argument for a recognized nvcc flag.

If the unknown option is followed by a separate command line argument, the argument will not be forwarded, unless it begins with the `-` character.

For example:

- `nvcc -forward-unknown-to-host-compiler -foo=bar a.cu` will forward `-foo=bar` to host compiler.
- `nvcc -forward-unknown-to-host-compiler -foo bar a.cu` will report an error for `bar` argument.
- `nvcc -forward-unknown-to-host-compiler -foo -bar a.cu` will forward `-foo` and `-bar` to host compiler.

`--forward-unknown-to-host-linker` (`-forward-unknown-to-host-linker`)

Forward unknown options to the host linker. An 'unknown option' is a command line argument that starts with `-` followed by another character, and is not a recognized nvcc flag or an argument for a recognized nvcc flag.

If the unknown option is followed by a separate command line argument, the argument will not be forwarded, unless it begins with the `-` character.

For example:

- `nvcc -forward-unknown-to-host-linker -foo=bar a.cu` will forward `-foo=bar` to host linker.
- `nvcc -forward-unknown-to-host-linker -foo bar a.cu` will report an error for `bar` argument.
- `nvcc -forward-unknown-to-host-linker -foo -bar a.cu` will forward `-foo` and `-bar` to host linker.

`--dont-use-profile` (`-noprof`)

Do not use configurations from the `nvcc.profile` file for compilation.

`--threads number` (`-t`)

Specify the maximum number of threads to be used to execute the compilation steps in parallel.

This option can be used to improve the compilation speed when compiling for multiple architectures. The compiler creates *number* threads to execute the compilation steps in parallel. If *number* is 1, this option is ignored. If *number* is 0, the number of threads used is the number of CPUs on the machine.

`--dryrun` (`-dryrun`)

List the compilation sub-commands without executing them.

`--verbose` (`-v`)

List the compilation sub-commands while executing them.

`--keep` (`-keep`)

Keep all intermediate files that are generated during internal compilation steps.

`--keep-dir directory` (`-keep-dir`)

Keep all intermediate files that are generated during internal compilation steps in this directory.

`--save-temps` (`-save-temps`)

This option is an alias of `--keep`.

`--clean-targets` (`-clean`)

Delete all the non-temporary files that the same `nvcc` command would generate without this option.

This option reverses the behavior of `nvcc`. When specified, none of the compilation phases will be executed. Instead, all of the non-temporary files that `nvcc` would otherwise create will be deleted.

--run-args arguments,... (-run-args)

Specify command line arguments for the executable when used in conjunction with --run.

--use-local-env (-use-local-env)

Skip MSVC environment initialization.

By default nvcc assumes that the MSVC environment needs to be initialized. This is done by executing the appropriate command file available for the MSVC installation detected or specified. Initializing the environment for each nvcc invocation can add noticeable overheads. If the environment used to invoke nvcc has already been configured, this option can be used to skip this step.

--input-drive-prefix prefix (-idp)

Specify the input drive prefix.

On Windows, all command line arguments that refer to file names must be converted to the Windows native format before they are passed to pure Windows executables. This option specifies how the current development environment represents absolute paths. Use /cygwin/ as prefix for Cygwin build environments and / as prefix for MinGW.

--dependency-drive-prefix prefix (-ddp)

Specify the dependency drive prefix.

On Windows, when generating dependency files (see --generate-dependencies), all file names must be converted appropriately for the instance of make that is used. Some instances of make have trouble with the colon in absolute paths in the native Windows format, which depends on the environment in which the make instance has been compiled. Use /cygwin/ as prefix for a Cygwin make, and / as prefix for MinGW. Or leave these file names in the native Windows format by specifying nothing.

--drive-prefix prefix (-dp)

Specify the drive prefix.

This option specifies prefix as both --input-drive-prefix and --dependency-drive-prefix.

--dependency-target-name target (-MT)

Specify the target name of the generated rule when generating a dependency file (see --generate-dependencies).

--no-align-double

Specify that -malign-double should not be passed as a compiler argument on 32-bit platforms.

WARNING: this makes the ABI incompatible with the CUDA's kernel ABI for certain 64-bit types.

--no-device-link (-nodlink)

Skip the device link step when linking object files.

--allow-unsupported-compiler (-allow-unsupported-compiler)

Disable nvcc check for supported host compiler versions.

Using an unsupported host compiler may cause compilation failure or incorrect run time execution. Use at your own risk. This option has no effect on MacOS.

5.2.6. Options for Steering CUDA Compilation

--default-stream {legacy|null|per-thread} (-default-stream)

Specify the stream that CUDA commands from the compiled program will be sent to by default.

Allowed Values

legacy

The CUDA legacy stream (per context, implicitly synchronizes with other streams)

per-thread

Normal CUDA stream (per thread, does not implicitly synchronize with other streams)

null

Deprecated alias for legacy

Default

legacy is used as the default stream.

5.2.7. Options for Steering GPU Code Generation

--gpu-architecture {arch|native|all|all-major} (-arch)

Specify the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

With the exception as described for the shorthand below, the architecture specified with this option must be a *virtual* architecture (such as compute_50). Normally, this option alone does not trigger assembly of the generated PTX for a *real* architecture (that is the role of nvcc option --gpu-code, see below); rather, its purpose is to control preprocessing and compilation of the input to PTX.

For convenience, in case of simple `nvcc` compilations, the following shorthand is supported. If no value for option `--gpu-code` is specified, then the value of this option defaults to the value of `--gpu-architecture`. In this situation, as only exception to the description above, the value specified for `--gpu-architecture` may be a *real* architecture (such as a sm_50), in which case `nvcc` uses the specified *real* architecture and its closest *virtual* architecture as effective architecture values. For example, `nvcc --gpu-architecture=sm_50` is equivalent to `nvcc --gpu-architecture=compute_50 --gpu-code=sm_50,compute_50`.

When `-arch=native` is specified, `nvcc` detects the visible GPUs on the system and generates codes for them, no PTX program will be generated for this option. It is a warning if no visible supported GPU on the system, and the default architecture will be used.

If `-arch=all` is specified, `nvcc` embeds a compiled code image for all supported architectures `(sm_*)`, and a PTX program for the highest major virtual architecture. For `-arch=all-major`, `nvcc` embeds a compiled code image for all supported major versions `(sm_*0)`, plus the earliest supported, and adds a PTX program for the highest major virtual architecture.

See Virtual Architecture Feature List for the list of supported *virtual* architectures and GPU Feature List for the list of supported *real* architectures.

Default

`sm_52` is used as the default value; PTX is generated for `compute_52` then assembled and optimized for `sm_52`.

`--gpu-code code,...` (`-code`)

Specify the name of the NVIDIA GPU to assemble and optimize PTX for.

`nvcc` embeds a compiled code image in the resulting executable for each specified *code* architecture, which is a true binary load image for each *real* architecture (such as sm_50), and PTX code for the *virtual* architecture (such as compute_50).

During runtime, such embedded PTX code is dynamically compiled by the CUDA runtime system if no binary load image is found for the *current* GPU.

Architectures specified for options `--gpu-architecture` and `--gpu-code` may be *virtual* as well as *real*, but the `code` architectures must be compatible with the `arch` architecture. When the `--gpu-code` option is used, the value for the `--gpu-architecture` option must be a *virtual* PTX architecture.

For instance, `--gpu-architecture=compute_60` is not compatible with `--gpu-code=sm_52`, because the earlier compilation stages will assume the availability of `compute_60` features that are not present on `sm_52`.

See Virtual Architecture Feature List for the list of supported *virtual* architectures and GPU Feature List for the list of supported *real* architectures.

`--generate-code specification` (`-gencode`)

This option provides a generalization of the `--gpu-architecture=arch --gpu-code=code,...` option combination for specifying `nvcc` behavior with respect to code generation.

Where use of the previous options generates code for different *real* architectures with the PTX for the same *virtual* architecture, option `--generate-code` allows multiple PTX generations for different *virtual* architectures. In fact, `--gpu-architecture=arch --gpu-code=code,...` is equivalent to `--generate-code=arch=arch,code=code,...`.

`--generate-code` options may be repeated for different virtual architectures.

See Virtual Architecture Feature List for the list of supported *virtual* architectures and GPU Feature List for the list of supported *real* architectures.

`--relocatable-device-code {true|false}` (`-rdc`)

Enable or disable the generation of relocatable device code.

If disabled, executable device code is generated. Relocatable device code must be linked before it can be executed.

Allowed Values

- `true`
- `false`

Default

The generation of relocatable device code is disabled.

`--entries entry,...` (`-e`)

Specify the global entry functions for which code must be generated.

PTX generated for all entry functions, but only the selected entry functions are assembled. Entry function names for this option must be specified in the mangled name.

Default

`nvcc` generates code for all entry functions.

`--maxrregcount amount` (`-maxrregcount`)

Specify the maximum amount of registers that GPU functions can use.

Until a function-specific limit, a higher value will generally increase the performance of individual GPU threads that execute this function. However, because thread registers are allocated from a global register pool on each GPU, a higher value of this option will also reduce the maximum thread block size, thereby reducing the amount of thread parallelism. Hence, a good `maxrregcount` value is the result of a trade-off.

Value less than the minimum registers required by ABI will be bumped up by the compiler to ABI minimum limit.

User program may not be able to make use of all registers as some registers are reserved by compiler.

Default

No maximum is assumed.

--use_fast_math (-use_fast_math)

Make use of fast math library.

--use_fast_math implies --ftz=true --prec-div=false --prec-sqrt=false --fmad=true.

--ftz {true|false} (-ftz)

Control single-precision denormals support.

--ftz=true flushes denormal values to zero and --ftz=false preserves denormal values.

--use_fast_math implies --ftz=true.

Allowed Values

- true
- false

Default

This option is set to false and nvcc preserves denormal values.

--prec-div {true|false} (-prec-div)

This option controls single-precision floating-point division and reciprocals.

--prec-div=true enables the IEEE round-to-nearest mode and --prec-div=false enables the fast approximation mode.

--use_fast_math implies --prec-div=false.

Allowed Values

- true
- false

Default

This option is set to true and nvcc enables the IEEE round-to-nearest mode.

--prec-sqrt {true|false} (-prec-sqrt)

This option controls single-precision floating-point square root.

--prec-sqrt=true enables the IEEE round-to-nearest mode and --prec-sqrt=false enables the fast approximation mode.

--use_fast_math implies --prec-sqrt=false.

Allowed Values

- true
- false

Default

This option is set to `true` and `nvcc` enables the IEEE round-to-nearest mode.

`--fmad {true|false}` (`-fmad`)

This option enables (disables) the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA).

`--use_fast_math` implies `--fmad=true`.

Allowed Values

- `true`
- `false`

Default

This option is set to `true` and `nvcc` enables the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA).

`--extra-device-vectorization` (`-extra-device-vectorization`)

This option enables more aggressive device code vectorization.

`--compile-as-tools-patch` (`-astoolspatch`)

Compile patch code for CUDA tools. Implies –keep-device-functions.

May only be used in conjunction with `--ptx` or `--cubin` or `--fatbin`.

Shall not be used in conjunction with `-rdc=true` or `-ewp`.

Some PTX ISA features may not be usable in this compilation mode.

`--keep-device-functions` (`-keep-device-functions`)

In whole program compilation mode, preserve user defined external linkage `__device__` function definitions in generated PTX.

5.2.8. Generic Tool Options

`--disable-warnings` (`-w`)

Inhibit all warning messages.

`--source-in-ptx` (`-src-in-ptx`)

Interleave source in PTX.

May only be used in conjunction with `--device-debug` or `--generate-line-info`.

`--restrict` (`-restrict`)

Assert that all kernel pointer parameters are restrict pointers.

`--Wno-deprecated-gpu-targets` (`-Wno-deprecated-gpu-targets`)

Suppress warnings about deprecated GPU target architectures.

`--Wno-deprecated-declarations` (`-Wno-deprecated-declarations`)

Suppress warning on use of a deprecated entity.

`--Wreorder` (`-Wreorder`)

Generate warnings when member initializers are reordered.

`--Wdefault-stream-launch` (`-Wdefault-stream-launch`)

Generate warning when an explicit stream argument is not provided in the `<<<...>>>` kernel launch syntax.

`--Wmissing-launch-bounds` (`-Wmissing-launch-bounds`)

Generate warning when a `__global__` function does not have an explicit `__launch_bounds__` annotation.

`--Wext-lambda-captures-this` (`-Wext-lambda-captures-this`)

Generate warning when an extended lambda implicitly captures `this`.

`--Werror kind,...` (`-Werror`)

Make warnings of the specified kinds into errors.

The following is the list of warning kinds accepted by this option:

`all-warnings`

Treat all warnings as errors.

`cross-execution-space-call`

Be more strict about unsupported cross execution space calls. The compiler will generate an error instead of a warning for a call from a `__host____device__` to a `__host__` function.

`reorder`

Generate errors when member initializers are reordered.

`default-stream-launch`

Generate error when an explicit stream argument is not provided in the `<<<...>>>` kernel launch syntax.

`missing-launch-bounds`

Generate warning when a `__global__` function does not have an explicit `__launch_bounds__` annotation.

`ext-lambda-captures-this`

Generate error when an extended lambda implicitly captures `this`.

`deprecated-declarations`

Generate error on use of a deprecated entity.

`--display-error-number` (`-err-no`)

This option displays a diagnostic number for any message generated by the CUDA frontend compiler (note: not the host compiler).

`--no-display-error-number` (`-no-err-no`)

This option disables the display of a diagnostic number for any message generated by the CUDA frontend compiler (note: not the host compiler).

`--diag-error errNum,...` (`-diag-error`)

Emit error for specified diagnostic message(s) generated by the CUDA frontend compiler (note: does not affect diagnostics generated by the host compiler/preprocessor).

`--diag-suppress errNum,...` (`-diag-suppress`)

Suppress specified diagnostic message(s) generated by the CUDA frontend compiler (note: does not affect diagnostics generated by the host compiler/preprocessor).

`--diag-warn errNum,...` (`-diag-warn`)

Emit warning for specified diagnostic message(s) generated by the CUDA frontend compiler (note: does not affect diagnostics generated by the host compiler/preprocessor).

`--resource-usage` (`-res-usage`)

Show resource usage such as registers and memory of the GPU code.

This option implies `--nvlink-options=--verbose` when `--relocatable-device-code=true` is set. Otherwise, it implies `--ptxas-options=--verbose`.

`--help` (`-h`)

Print help information on this tool.

`--version` (`-V`)

Print version information on this tool.

`--options-file file,...` (`-optf`)

Include command line options from specified file.

`--time filename` (`-time`)

Generate a comma separated value table with the time taken by each compilation phase, and append it at the end of the file given as the option argument. If the file is empty, the column headings are generated in the first row of the table.

If the file name is `-`, the timing data is generated in stdout.

`--qpp-config config` (`-qpp-config`)

Specify the configuration ([[compiler/]version,][target]) when using q++ host compiler. The argument will be forwarded to the q++ compiler with its -V flag.

`--list-gpu-code` (`-code-ls`)

List the gpu architectures (sm_XX) supported by the tool and exit.

If both –list-gpu-code and –list-gpu-arch are set, the list is displayed using the same format as the –generate-code value.

`--list-gpu-arch` (`-arch-ls`)

List the virtual device architectures (compute_XX) supported by the tool and exit.

If both –list-gpu-arch and –list-gpu-code are set, the list is displayed using the same format as the –generate-code value.

5.2.9. Phase Options

The following sections lists some useful options to lower level compilation tools.

Ptxas Options

The following table lists some useful `ptxas` options which can be specified with `nvcc` option `-Xptxas`.

`--allow-expensive-optimizations` (`-allow-expensive-optimizations`)

Enable (disable) to allow compiler to perform expensive optimizations using maximum available resources (memory and compile-time).

If unspecified, default behavior is to enable this feature for optimization level >= 02.

`--compile-only` (`-c`)

Generate relocatable object.

`--def-load-cache` (`-dlcm`)

Default cache modifier on global/generic load.

Default value: `ca`.

`--def-store-cache` (`-dscm`)

Default cache modifier on global/generic store.

`--device-debug` (`-g`)

Semantics same as `nvcc` option `--device-debug`.

--disable-optimizer-constants (-disable-optimizer-consts)

Disable use of optimizer constant bank.

--entry entry,... (-e)

Semantics same as nvcc option --entries.

--fmad (-fmad)

Semantics same as nvcc option --fmad.

--force-load-cache (-flcm)

Force specified cache modifier on global/generic load.

--force-store-cache (-fscm)

Force specified cache modifier on global/generic store.

--generate-line-info (-lineinfo)

Semantics same as nvcc option --generate-line-info.

--gpu-name gpuname (-arch)

Specify name of NVIDIA GPU to generate code for.

This option also takes virtual compute architectures, in which case code generation is suppressed. This can be used for parsing only.

Allowed values for this option: compute_50, compute_52, compute_53, compute_60, compute_61, compute_62, compute_70, compute_72, compute_75, compute_80, compute_86, compute_87, compute_89,compute_90,lto_50, lto_52, lto_53, lto_60, lto_61, lto_62, lto_70, lto_72, lto_75, lto_80, lto_86, lto_87, lto_89,lto_90, sm_50, sm_52, sm_53, sm_60, sm_61, sm_62, sm_70, sm_72, sm_75, sm_80, sm_86, sm_87, sm_89, sm_90

Default value: sm_52.

`--help` (`-h`)

Semantics same as `nvcc` option `--help`.

`--machine` (`-m`)

Semantics same as `nvcc` option `--machine`.

`--maxrregcount amount` (`-maxrregcount`)

Semantics same as `nvcc` option `--maxrregcount`.

`--opt-level N` (`-O`)

Specify optimization level.

Default value: 3.

`--options-file file,...` (`-optf`)

Semantics same as `nvcc` option `--options-file`.

`--position-independent-code` (`-pic`)

Generate position-independent code.

Default value: `false`

`--preserve-relocs` (`-preserve-relocs`)

This option will make `ptxas` to generate relocatable references for variables and preserve relocations generated for them in linked executable.

`--sp-bound-check` (`-sp-bound-check`)

Generate stack-pointer bounds-checking code sequence.

This option is turned on automatically when `--device-debug` or `--opt-level=0` is specified.

`--verbose` (`-v`)

Enable verbose mode which prints code generation statistics.

`--version` (`-V`)

Semantics same as `nvcc` option `--version`.

`--warning-as-error` (`-Werror`)

Make all warnings into errors.

`--warn-on-double-precision-use` (`-warn-double-usage`)

Warning if double(s) are used in an instruction.

`--warn-on-local-memory-usage` (`-warn-lmem-usage`)

Warning if local memory is used.

`--warn-on-spills` (`-warn-spills`)

Warning if registers are spilled to local memory.

`--compile-as-tools-patch` (`-astoolspatch`)

Compile patch code for CUDA tools.

Shall not be used in conjunction with `-Xptxas -c` or `-ewp`.

Some PTX ISA features may not be usable in this compilation mode.

NVLINK Options

The following table lists some useful `nvlink` options which can be specified with `nvcc` option `--nvlink-options`.

--disable-warnings (-w)

Inhibit all warning messages.

--preserve-relocs (-preserve-relocs)

Preserve resolved relocations in linked executable.

--verbose (-v)

Enable verbose mode which prints code generation statistics.

--warning-as-error (-Werror)

Make all warnings into errors.

--suppress-arch-warning (-suppress-arch-warning)

Suppress the warning that otherwise is printed when object does not contain code for target arch.

--suppress-stack-size-warning (-suppress-stack-size-warning)

Suppress the warning that otherwise is printed when stack size cannot be determined.

--dump-callgraph (-dump-callgraph)

Dump information about the callgraph and register usage.

5.3. NVCC Environment Variables

The nvcc command line flags can be augmented using the following environment variables, if set:

NVCC_PREPEND_FLAGS

Flags to be injected before the normal nvcc command line.

NVCC_APPEND_FLAGS

Flags to be injected after the normal nvcc command line.

For example, after setting:

```
export NVCC_PREPEND_FLAGS='-G -keep -arch=sm_60'

export NVCC_APPEND_FLAGS='-DNAME=" foo "'
```

The following invocation:

```
nvcc foo.cu -o foo
```

Becomes equivalent to:

```
nvcc -G -keep -arch=sm_60 foo.cu -o foo -DNAME=" foo "
```

These environment variables can be useful for injecting `nvcc` flags globally without modifying build scripts.

The additional flags coming from either NVCC_PREPEND_FLAGS or NVCC_APPEND_FLAGS will be listed in the verbose log (`--verbose`).

Chapter 6. GPU Compilation

This chapter describes the GPU compilation model that is maintained by nvcc, in cooperation with the CUDA driver. It goes through some technical sections, with concrete examples at the end.

6.1. GPU Generations

In order to allow for architectural evolution, NVIDIA GPUs are released in different generations. New generations introduce major improvements in functionality and/or chip architecture, while GPU models within the same generation show minor configuration differences that *moderately* affect functionality, performance, or both.

Binary compatibility of GPU applications is not guaranteed across different generations. For example, a CUDA application that has been compiled for a Fermi GPU will very likely not run on a Kepler GPU (and vice versa). This is the instruction set and instruction encodings of a generation is different from those of of other generations.

Binary compatibility within one GPU generation can be guaranteed under certain conditions because they share the basic instruction set. This is the case between two GPU versions that do not show functional differences at all (for instance when one version is a scaled down version of the other), or when one version is functionally included in the other. An example of the latter is the *base* Maxwell version sm_52 whose functionality is a subset of all other Maxwell versions: any code compiled for sm_52 will run on all other Maxwell GPUs.

6.2. GPU Feature List

The following table lists the names of the current GPU architectures, annotated with the functional capabilities that they provide. There are other differences, such as amounts of register and processor clusters, that only affect execution performance.

In the CUDA naming scheme, GPUs are named sm_xy, where x denotes the GPU generation number, and y the version in that generation. Additionally, to facilitate comparing GPU capabilities, CUDA attempts to choose its GPU names such that if x1y1 <= x2y2 then all non-ISA related capabilities of sm_x1y1 are included in those of sm_x2y2. From this it indeed follows that sm_52 is the *base* Maxwell model, and it also explains why higher entries in the tables are always functional extensions to the lower entries. This is denoted by the plus sign in the table. Moreover, if we abstract from the instruction encoding, it implies that sm_52's functionality will continue to be included in all later GPU generations. As we will see next, this property will be the foundation for application compatibility support by nvcc.

`sm_50`, `sm_52` and `sm_53`	Maxwell support
`sm_60`, `sm_61`, and `sm_62`	Pascal support
`sm_70` and `sm_72`	Volta support
`sm_75`	Turing support
`sm_80`, `sm_86` and `sm_87`	NVIDIA Ampere GPU architecture support
`sm_89`	Ada support
`sm_90`, `sm_90a`	Hopper support

6.3. Application Compatibility

Binary code compatibility over CPU generations, together with a published instruction set architecture is the usual mechanism for ensuring that distributed applications *out there in the field* will continue to run on newer versions of the CPU when these become mainstream.

This situation is different for GPUs, because NVIDIA cannot guarantee binary compatibility without sacrificing regular opportunities for GPU improvements. Rather, as is already conventional in the graphics programming domain, `nvcc` relies on a two stage compilation model for ensuring application compatibility with future GPU generations.

6.4. Virtual Architectures

GPU compilation is performed via an intermediate representation, PTX, which can be considered as assembly for a virtual GPU architecture. Contrary to an actual graphics processor, such a virtual GPU is defined entirely by the set of capabilities, or features, that it provides to the application. In particular, a virtual GPU architecture provides a (largely) generic instruction set, and binary instruction encoding is a non-issue because PTX programs are always represented in text format.

Hence, a `nvcc` compilation command always uses two architectures: a *virtual* intermediate architecture, plus a *real* GPU architecture to specify the intended processor to execute on. For such an `nvcc` command to be valid, the *real* architecture must be an implementation of the *virtual* architecture. This is further explained below.

The chosen virtual architecture is more of a statement on the GPU capabilities that the application requires: using a *smallest* virtual architecture still allows a *widest* range of actual architectures for the second `nvcc` stage. Conversely, specifying a virtual architecture that provides features unused by the application unnecessarily restricts the set of possible GPUs that can be specified in the second `nvcc` stage.

From this it follows that the virtual architecture should always be chosen as *low* as possible, thereby maximizing the actual GPUs to run on. The *real* architecture should be chosen as *high* as possible (assuming that this always generates better code), but this is only possible with knowledge of the actual GPUs on which the application is expected to run. As we will see later, in the situation of just in time compilation, where the driver has this exact knowledge: the runtime GPU is the one on which the program is about to be launched/executed.

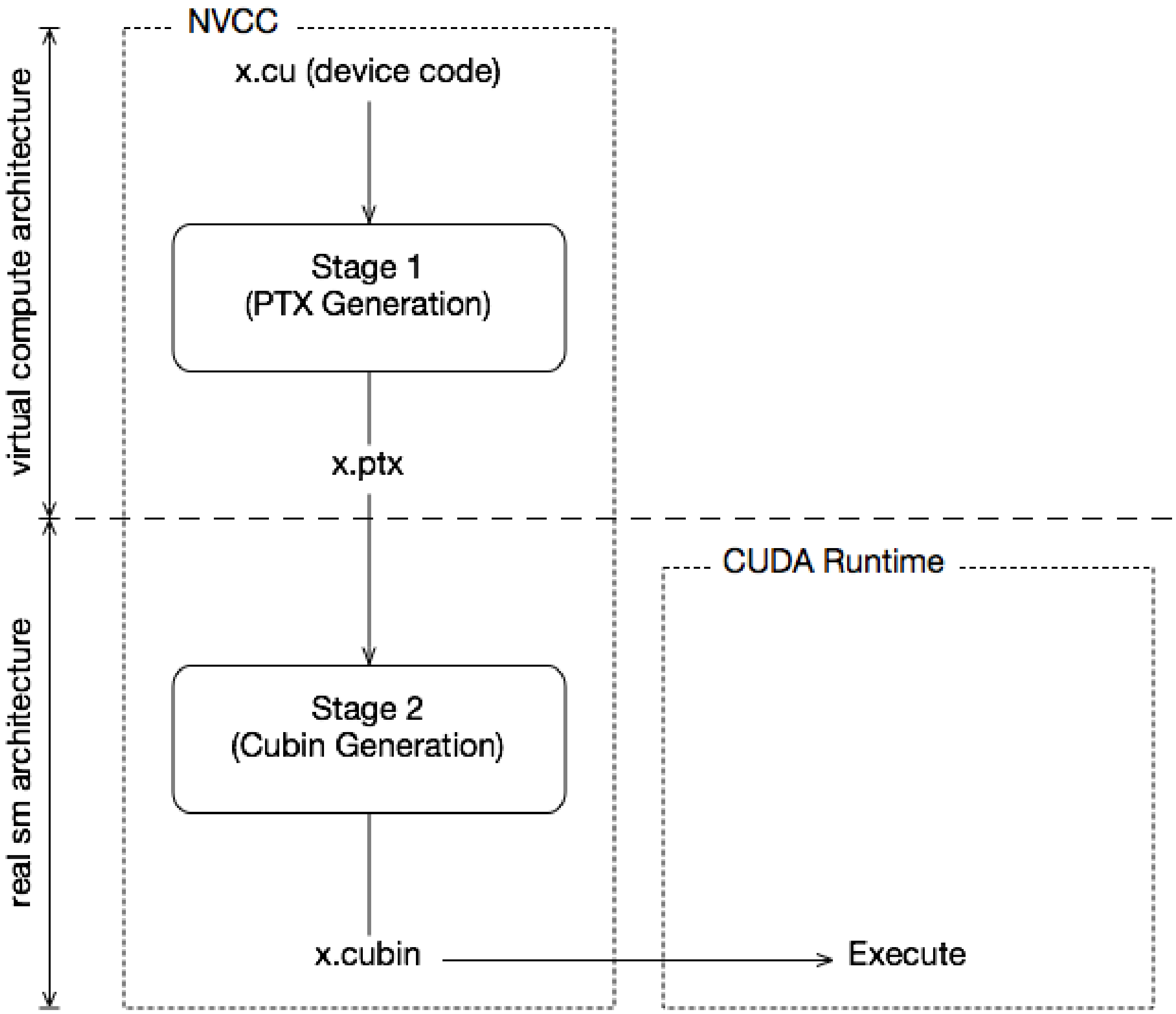

Fig. 1: Two-Staged Compilation with Virtual and Real Architectures

6.5. Virtual Architecture Feature List

compute_50, compute_52, and compute_53	Maxwell support
compute_60, compute_61, and compute_62	Pascal support
compute_70 and compute_72	Volta support
compute_75	Turing support
compute_80, compute_86 and compute_87	NVIDIA Ampere GPU architecture support
compute_89	Ada support
compute_90, compute_90a	Hopper support

The above table lists the currently defined virtual architectures. The virtual architecture naming scheme is the same as the real architecture naming scheme shown in Section GPU Feature List.

6.6. Further Mechanisms

Clearly, compilation staging in itself does not help towards the goal of application compatibility with future GPUs. For this we need the two other mechanisms by CUDA Samples: just in time compilation (JIT) and fatbinaries.

6.6.1. Just-in-Time Compilation

The compilation step to an actual GPU binds the code to one generation of GPUs. Within that generation, it involves a choice between GPU *coverage* and possible performance. For example, compiling to sm_52 allows the code to run on all Maxwell-generation GPUs, but compiling to sm_53 would probably yield better code if Maxwell GM206 and later are the only targets.

By specifying a virtual code architecture instead of a *real* GPU, nvcc postpones the assembly of PTX code until application runtime, at which the target GPU is exactly known. For instance, the command below allows generation of exactly matching GPU binary code, when the application is launched on an sm_50 or later architecture.

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50
```

The disadvantage of just in time compilation is increased application startup delay, but this can be alleviated by letting the CUDA driver use a compilation cache (refer to “Section 3.1.1.2. Just-in-Time Compilation” of CUDA C++ Programming Guide) which is persistent over multiple runs of the applications.

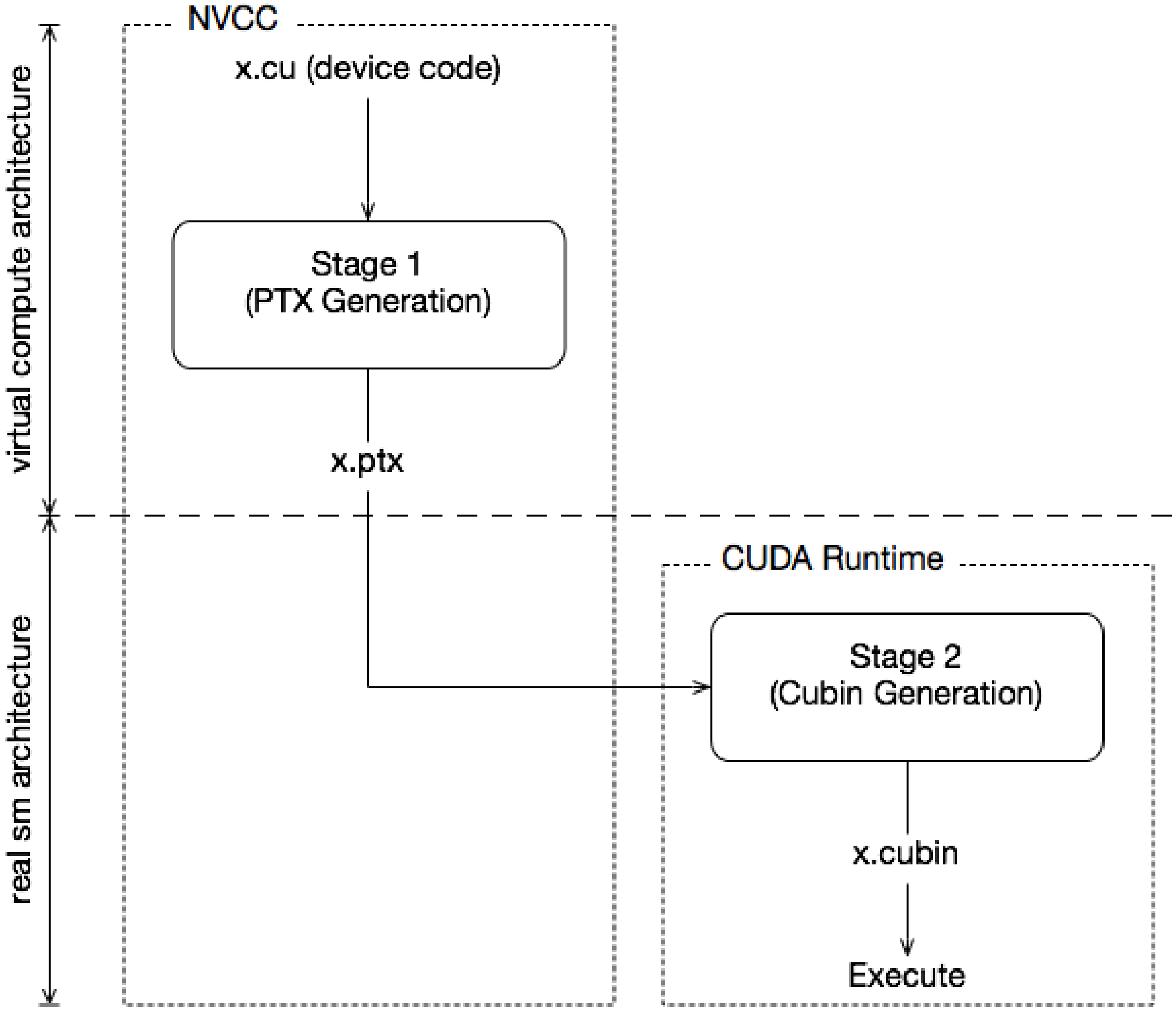

Fig. 2: Just-in-Time Compilation of Device Code

6.6.2. Fatbinaries

A different solution to overcome startup delay by JIT while still allowing execution on newer GPUs is to specify multiple code instances, as in

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50,sm_50,sm_52
```

This command generates exact code for two Maxwell variants, plus PTX code for use by JIT in case a next-generation GPU is encountered. nvcc organizes its device code in fatbinaries, which are able to hold multiple translations of the same GPU source code. At runtime, the CUDA driver will select the most appropriate translation when the device function is launched.

6.7. NVCC Examples

6.7.1. Base Notation

nvcc provides the options --gpu-architecture and --gpu-code for specifying the target architectures for both translation stages. Except for allowed short hands described below, the --gpu-architecture option takes a single value, which must be the name of a virtual compute architecture, while option --gpu-code takes a list of values which must all be the names of actual GPUs. nvcc performs a stage 2 translation for each of these GPUs, and will embed the result in the result of compilation (which usually is a host object file or executable).

Example

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=sm_50,sm_52
```

6.7.2. Shorthand

nvcc allows a number of shorthands for simple cases.

Shorthand 1

--gpu-code arguments can be virtual architectures. In this case the stage 2 translation will be omitted for such virtual architecture, and the stage 1 PTX result will be embedded instead. At application launch, and in case the driver does not find a better alternative, the stage 2 compilation will be invoked by the driver with the PTX as input.

Example

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50,sm_50,sm_52
```

Shorthand 2

The --gpu-code option can be omitted. Only in this case, the --gpu-architecture value can be a non-virtual architecture. The --gpu-code values default to the *closest* virtual architecture that is implemented by the GPU specified with --gpu-architecture, plus the --gpu-architecture, value itself. The *closest* virtual architecture is used as the effective --gpu-architecture, value. If the --gpu-architecture value is a virtual architecture, it is also used as the effective --gpu-code value.

Example

```
nvcc x.cu --gpu-architecture=sm_52
nvcc x.cu --gpu-architecture=compute_50
```

are equivalent to

```
nvcc x.cu --gpu-architecture=compute_52 --gpu-code=sm_52,compute_52
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50
```

Shorthand 3

Both --gpu-architecture and --gpu-code options can be omitted.

Example

```
nvcc x.cu
```

is equivalent to

```
nvcc x.cu --gpu-architecture=compute_52 --gpu-code=sm_52,compute_52
```

6.7.3. Extended Notation

The options --gpu-architecture and --gpu-code can be used in all cases where code is to be generated for one or more GPUs using a common virtual architecture. This will cause a single invocation of nvcc stage 1 (that is, preprocessing and generation of virtual PTX assembly code), followed by a compilation stage 2 (binary code generation) repeated for each specified GPU.

Using a common virtual architecture means that all assumed GPU features are fixed for the entire nvcc compilation. For instance, the following nvcc command assumes no half-precision floating-point operation support for both the sm_50 code and the sm_53 code:

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50,sm_50,sm_53
```

Sometimes it is necessary to perform different GPU code generation steps, partitioned over different architectures. This is possible using nvcc option --generate-code, which then must be used instead of a --gpu-architecture and --gpu-code combination.

Unlike option --gpu-architecture option --generate-code, may be repeated on the nvcc command line. It takes sub-options arch and code, which must not be confused with their main option equivalents, but behave similarly. If repeated architecture compilation is used, then the device code must use conditional compilation based on the value of the architecture identification macro __CUDA_ARCH__, which is described in the next section.

For example, the following assumes absence of half-precision floating-point operation support for the `sm_50` and `sm_52` code, but full support on `sm_53`:

```
nvcc x.cu \
    --generate-code arch=compute_50,code=sm_50 \
    --generate-code arch=compute_50,code=sm_52 \
    --generate-code arch=compute_53,code=sm_53
```

Or, leaving actual GPU code generation to the JIT compiler in the CUDA driver:

```
nvcc x.cu \
    --generate-code arch=compute_50,code=compute_50 \
    --generate-code arch=compute_53,code=compute_53
```

The code sub-options can be combined with a slightly more complex syntax:

```
nvcc x.cu \
    --generate-code arch=compute_50,code=[sm_50,sm_52] \
    --generate-code arch=compute_53,code=sm_53
```

6.7.4. Virtual Architecture Macros

The architecture identification macro `__CUDA_ARCH__` is assigned a three-digit value string `xy0` (ending in a literal `0`) during each `nvcc` compilation stage 1 that compiles for `compute_xy`.

This macro can be used in the implementation of GPU functions for determining the virtual architecture for which it is currently being compiled. The host code (the non-GPU code) must *not* depend on it.

The architecture list macro `__CUDA_ARCH_LIST__` is a list of comma-separated `__CUDA_ARCH__` values for each of the virtual architectures specified in the compiler invocation. The list is sorted in numerically ascending order.

The macro `__CUDA_ARCH_LIST__` is defined when compiling C, C++ and CUDA source files.

For example, the following nvcc compilation command line will define `__CUDA_ARCH_LIST__` as `500,530,800` :

```
nvcc x.cu \
--generate-code arch=compute_80,code=sm_80 \
--generate-code arch=compute_50,code=sm_52 \
--generate-code arch=compute_50,code=sm_50 \
--generate-code arch=compute_53,code=sm_53
```

Chapter 7. Using Separate Compilation in CUDA

Prior to the 5.0 release, CUDA did not support separate compilation, so CUDA code could not call device functions or access variables across files. Such compilation is referred to as *whole program compilation*. We have always supported the separate compilation of host code, it was just the device CUDA code that needed to all be within one file. Starting with CUDA 5.0, separate compilation of device code is supported, but the old whole program mode is still the default, so there are new options to invoke separate compilation.

7.1. Code Changes for Separate Compilation

The code changes required for separate compilation of device code are the same as what you already do for host code, namely using `extern` and `static` to control the visibility of symbols. Note that previously `extern` was ignored in CUDA code; now it will be honored. With the use of `static` it is possible to have multiple device symbols with the same name in different files. For this reason, the CUDA API calls that referred to symbols by their string name are deprecated; instead the symbol should be referenced by its address.

7.2. NVCC Options for Separate Compilation

CUDA works by embedding device code into host objects. In whole program compilation, it embeds executable device code into the host object. In separate compilation, we embed relocatable device code into the host object, and run `nvlink`, the device linker, to link all the device code together. The output of nvlink is then linked together with all the host objects by the host linker to form the final executable.

The generation of relocatable vs executable device code is controlled by the `--relocatable-device-code` option.

The `--compile` option is already used to control stopping a compile at a host object, so a new option `--device-c` is added that simply does `--relocatable-device-code=true --compile`.

To invoke just the device linker, the `--device-link` option can be used, which emits a host object containing the embedded executable device code. The output of that must then be passed to the host linker. Or:

```
nvcc <objects>
```

can be used to implicitly call both the device and host linkers. This works because if the device linker does not see any relocatable code it does not do anything.

Figure 4 shows the flow.

7.3. Libraries

The device linker has the ability to read the static host library formats (.a on Linux and Mac OS X, .lib on Windows). It ignores any dynamic (.so or .dll) libraries. The --library and --library-path options can be used to pass libraries to both the device and host linker. The library name is specified without the library file extension when the --library option is used.

```
nvcc --gpu-architecture=sm_50 a.o b.o --library-path=<path> --library=foo
```

Alternatively, the library name, including the library file extension, can be used without the --library option on Windows.

```
nvcc --gpu-architecture=sm_50 a.obj b.obj foo.lib --library-path=<path>
```

Note that the device linker ignores any objects that do not have relocatable device code.

7.4. Examples

Suppose we have the following files:

```
//---------- b.h ----------
#define N 8

extern __device__ int g[N];

extern __device__ void bar(void);
```

```
//---------- b.cu ----------
#include "b.h"

__device__ int g[N];

__device__ void bar (void)
{
  g[threadIdx.x]++;
}
```

```
//---------- a.cu ----------
#include <stdio.h>
#include "b.h"

__global__ void foo (void) {
```

(continues on next page)

(continued from previous page)

```
  __shared__ int a[N];
  a[threadIdx.x] = threadIdx.x;

  __syncthreads();

  g[threadIdx.x] = a[blockDim.x - threadIdx.x - 1];

  bar();
}

int main (void) {
  unsigned int i;
  int *dg, hg[N];
  int sum = 0;

  foo<<<1, N>>>();

  if(cudaGetSymbolAddress((void**)&dg, g)){
      printf("couldn't get the symbol addr\n");
      return 1;
  }
  if(cudaMemcpy(hg, dg, N * sizeof(int), cudaMemcpyDeviceToHost)){
      printf("couldn't memcpy\n");
      return 1;
  }

  for (i = 0; i < N; i++) {
    sum += hg[i];
  }
  if (sum == 36) {
    printf("PASSED\n");
  } else {
    printf("FAILED (%d)\n", sum);
  }

  return 0;
}
```

These can be compiled with the following commands (these examples are for Linux):

```
nvcc --gpu-architecture=sm_50 --device-c a.cu b.cu
nvcc --gpu-architecture=sm_50 a.o b.o
```

If you want to invoke the device and host linker separately, you can do:

```
nvcc --gpu-architecture=sm_50 --device-c a.cu b.cu
nvcc --gpu-architecture=sm_50 --device-link a.o b.o --output-file link.o
g++ a.o b.o link.o --library-path=<path> --library=cudart
```

Note that all desired target architectures must be passed to the device linker, as that specifies what will be in the final executable (some objects or libraries may contain device code for multiple architectures, and the link step can then choose what to put in the final executable).

If you want to use the driver API to load a linked cubin, you can request just the cubin:

```
nvcc --gpu-architecture=sm_50 --device-link a.o b.o \
    --cubin --output-file link.cubin
```

The objects could be put into a library and used with:

```
nvcc --gpu-architecture=sm_50 --device-c a.cu b.cu
nvcc --lib a.o b.o --output-file test.a
nvcc --gpu-architecture=sm_50 test.a
```

Note that only static libraries are supported by the device linker.

A PTX file can be compiled to a host object file and then linked by using:

```
nvcc --gpu-architecture=sm_50 --device-c a.ptx
```

An example that uses libraries, host linker, and dynamic parallelism would be:

```
nvcc --gpu-architecture=sm_50 --device-c a.cu b.cu
nvcc --gpu-architecture=sm_50 --device-link a.o b.o --output-file link.o
nvcc --lib --output-file libgpu.a a.o b.o link.o
g++ host.o --library=gpu --library-path=<path> \
    --library=cudadevrt --library=cudart
```

It is possible to do multiple device links within a single host executable, as long as each device link is independent of the other. This requirement of independence means that they cannot share code across device executables, nor can they share addresses (e.g., a device function address can be passed from host to device for a callback only if the device link sees both the caller and potential callback callee; you cannot pass an address from one device executable to another, as those are separate address spaces).

7.5. Optimization Of Separate Compilation

Separately compiled code may not have as high of performance as whole program code because of the inability to inline code across files. A way to still get optimal performance is to use link-time optimization, which stores intermediate code which is then linked together to perform high level optimizations. This can be done with the `--dlink-time-opt` or `-dlto` option. This option must be specified at both compile and link time. If only some of the files are compiled with `-dlto`, then those will be linked and optimized together while the rest uses the normal separate compilation. A side effect is that this shifts some of the compile time to the link phase, and there may be some scalability issues with really large codes. If you want to compile using `-gencode` to build for multiple arch, use `-dc -gencode arch=compute_NN,code=lto_NN` to specify the intermediate IR to be stored (where `NN` is the SM architecture version). Then use `-dlto` option to link for a specific architecture.

As of CUDA 12.0 there is support for runtime LTO via the `nvJitLink` library.

7.6. Potential Separate Compilation Issues

7.6.1. Object Compatibility

Only relocatable device code with the same ABI version, link-compatible SM target architecture, and same pointer size (32 or 64) can be linked together. The toolkit version of the linker must be >= the toolkit version of the objects. Incompatible objects will produce a link error. Link-compatible SM architectures are ones that have compatible SASS binaries that can combine without translating, e.g.

sm_52 and sm_50. An object could have been compiled for a different architecture but also have PTX available, in which case the device linker will JIT the PTX to cubin for the desired architecture and then link. Relocatable device code requires CUDA 5.0 or later Toolkit.

If Link Time Optimization is used with `-dlto`, the intermediate LTOIR is only guaranteed to be compatible within a major release (e.g. can link together 12.0 and 12.1 LTO intermediates, but not 12.1 and 11.6).

If a kernel is limited to a certain number of registers with the `launch_bounds` attribute or the `--maxrregcount` option, then all functions that the kernel calls must not use more than that number of registers; if they exceed the limit, then a link error will be given.

7.6.2. JIT Linking Support

JIT linking means doing an implicit relink of the code at load time. If the cubin does not match the target architecture at load time, the driver re-invokes the device linker to generate cubin for the target architecture, by first JIT'ing the PTX for each object to the appropriate cubin, and then linking together the new cubin. If PTX or cubin for the target architecture is not found for an object, then the link will fail. Implicit JIT linking of the LTO intermediates is not supported at this time, although they can be explicitly linked with the `nvJitLink` library.

7.6.3. Implicit CUDA Host Code

A file like `b.cu` above only contains CUDA device code, so one might think that the b.o object doesn't need to be passed to the host linker. But actually there is implicit host code generated whenever a device symbol can be accessed from the host side, either via a launch or an API call like `cudaGetSymbolAddress()`. This implicit host code is put into `b.o`, and needs to be passed to the host linker. Plus, for JIT linking to work all device code must be passed to the host linker, else the host executable will not contain device code needed for the JIT link. So a general rule is that the device linker and host linker must see the same host object files (if the object files have any device references in them—if a file is pure host then the device linker doesn't need to see it). If an object file containing device code is not passed to the host linker, then you will see an error message about the function `__cudaRegisterLinkedBinary_name` calling an undefined or unresolved symbol `__fatbinwrap_name`.

7.6.4. Using __CUDA_ARCH__

In separate compilation, `__CUDA_ARCH__` must not be used in headers such that different objects could contain different behavior. Or, it must be guaranteed that all objects will compile for the same compute_arch. If a weak function or template function is defined in a header and its behavior depends on `__CUDA_ARCH__`, then the instances of that function in the objects could conflict if the objects are compiled for different compute arch. For example, if an a.h contains:

```
template<typename T>
__device__ T* getptr(void)
{
#if __CUDA_ARCH__ == 500
  return NULL; /* no address */
#else
  __shared__ T arr[256];
```

(continues on next page)

(continued from previous page)

```
  return arr;
#endif
}
```

Then if a.cu and b.cu both include a.h and instantiate `getptr` for the same type, and b.cu expects a non-NULL address, and compile with:

```
nvcc --gpu-architecture=compute_50 --device-c a.cu
nvcc --gpu-architecture=compute_52 --device-c b.cu
nvcc --gpu-architecture=sm_52 a.o b.o
```

At link time only one version of the getptr is used, so the behavior would depend on which version is picked. To avoid this, either a.cu and b.cu must be compiled for the same compute arch, or `__CUDA_ARCH__` should not be used in the shared header function.

7.6.5. Device Code in Libraries

If a device function with non-weak external linkage is defined in a library as well as a non-library object (or another library), the device linker will complain about the multiple definitions (this differs from traditional host linkers that may ignore the function definition from the library object, if it was already found in an earlier object).

Chapter 8. Miscellaneous NVCC Usage

8.1. Cross Compilation

Cross compilation is controlled by using the following `nvcc` command line option:

- `--compiler-bindir` is used for cross compilation, where the underlying host compiler is capable of generating objects for the target platform.

On an x86 system, if a CUDA toolkit installation has been configured to support cross compilation to both Tegra and non-Tegra ARM targets, then `nvcc` will use the non-Tegra configuration by default, when an ARM host cross compiler has been specified. To use the Tegra configuration instead, pass "`-target-dir aarch64-linux`" to nvcc.

8.2. Keeping Intermediate Phase Files

`nvcc` stores intermediate results by default into temporary files that are deleted immediately before it completes. The location of the temporary file directories used are, depending on the current platform, as follows:

Windows Value of environment variable `TEMP` is used. If it is not set, `C:\Windows\temp` is used instead.

Other Platforms Value of environment variable `TMPDIR` is used. If it is not set, `/tmp` is used instead.

Option `--keep` makes `nvcc` store these intermediate files in the current directory or in the directory specified by `--keep-dir` instead, with names as described in Supported Phases.

8.3. Cleaning Up Generated Files

All files generated by a particular `nvcc` command can be cleaned up by repeating the command, but with additional option `--clean-targets`. This option is particularly useful after using `--keep`, because the `--keep` option usually leaves quite an amount of intermediate files around.

Because using `--clean-targets` will remove exactly what the original `nvcc` command created, it is important to exactly repeat all of the options in the original command. For instance, in the following example, omitting `--keep`, or adding `--compile` will have different cleanup effects.

```
nvcc acos.cu --keep
nvcc acos.cu --keep --clean-targets
```

8.4. Printing Code Generation Statistics

A summary on the amount of used registers and the amount of memory needed per compiled device function can be printed by passing option `--resource-usage` to nvcc:

```
$ nvcc --resource-usage acos.cu -arch sm_80
ptxas info    : 1536 bytes gmem
ptxas info    : Compiling entry function 'acos_main' for 'sm_80'
ptxas info    : Function properties for acos_main
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 6 registers, 1536 bytes smem, 32 bytes cmem[0]
```

As shown in the above example, the amount of statically allocated global memory (gmem) is listed.

Global memory and some of the constant banks are module scoped resources and not per kernel resources. Allocation of constant variables to constant banks is profile specific.

Followed by this, per kernel resource information is printed.

Stack frame is per thread stack usage used by this function. Spill stores and loads represent stores and loads done on stack memory which are being used for storing variables that couldn’t be allocated to physical registers.

Similarly number of registers, amount of shared memory and total space in constant bank allocated is shown.

Chapter 9. Notices

9.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

9.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

9.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

9.4. Copyright