CUDA COMPILER DRIVER NVCC

TRM-06721-001_v7.5 | September 2015

Reference Guide

CHANGES FROM PREVIOUS VERSION

- Major update to the document to reflect recent **nvcc** changes.

TABLE OF CONTENTS

Chapter 1. Introduction 1
1.1. Overview 1
1.1.1. CUDA Programming Model 1
1.1.2. CUDA Sources 1
1.1.3. Purpose of NVCC 2
1.2. Supported Host Compilers 2
Chapter 2. Compilation Phases 3
2.1. NVCC Identification Macro 3
2.2. NVCC Phases 3
2.3. Supported Input File Suffixes 4
2.4. Supported Phases 4
Chapter 3. NVCC Command Options 7
3.1. Command Option Types and Notation 7
3.2. Command Option Description 8
3.2.1. Options for Specifying the Compilation Phase 8
3.2.2. File and Path Specifications 9
3.2.3. Options for Specifying Behavior of Compiler/Linker 10
3.2.4. Options for Passing Specific Phase Options 11
3.2.5. Options for Guiding the Compiler Driver 12
3.2.6. Options for Steering CUDA Compilation 13
3.2.7. Options for Steering GPU Code Generation 13
3.2.8. Generic Tool Options 17
3.2.9. Phase Options 17
3.2.9.1. Ptxas Options 17
3.2.9.2. NVLINK Options 19
Chapter 4. The CUDA Compilation Trajectory 20
Chapter 5. GPU Compilation 22
5.1. GPU Generations 22
5.2. GPU Feature List 22
5.3. Application Compatibility 23
5.4. Virtual Architectures 23
5.5. Virtual Architecture Feature List 24
5.6. Further Mechanisms 25
5.6.1. Just-in-Time Compilation 25
5.6.2. Fatbinaries 26
5.7. NVCC Examples 26
5.7.1. Base Notation 26
5.7.2. Shorthand 26
5.7.2.1. Shorthand 1 26
5.7.2.2. Shorthand 2 26

5.7.2.3. Shorthand 3....27
5.7.3. Extended Notation....27
5.7.4. Virtual Architecture Identification Macro....28
Chapter 6. Using Separate Compilation in CUDA....29
6.1. Code Changes for Separate Compilation....29
6.2. NVCC Options for Separate Compilation....29
6.3. Libraries....30
6.4. Examples....31
6.5. Potential Separate Compilation Issues....32
6.5.1. Object Compatibility....32
6.5.2. JIT Linking Support....33
6.5.3. Implicit CUDA Host Code....33
6.5.4. Using __CUDA_ARCH__....33
Chapter 7. Miscellaneous NVCC Usage....35
7.1. Cross Compilation....35
7.2. Keeping Intermediate Phase Files....35
7.3. Cleaning Up Generated Files....35
7.4. Printing Code Generation Statistics....36

LIST OF FIGURES

Figure 1 CUDA Whole Program Compilation Trajectory 21

Figure 2 Two-Staged Compilation with Virtual and Real Architectures 24

Figure 3 Just-in-Time Compilation of Device Code 25

Figure 4 CUDA Separate Compilation Trajectory 30

Chapter 1. INTRODUCTION

1.1. Overview

1.1.1. 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.

1.1.2. 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.

1.1.3. 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.

1.2. 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 Mac OS X
- NVIDIA CUDA Installation Guide for Microsoft Windows

On all platforms, the default host compiler executable (**gcc** and **g++** on Linux, **clang** and **clang++** on Mac OS X, 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 2. COMPILATION PHASES

2.1. 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 sources files in relocatable device code mode (see NVCC Options for Separate Compilation).

__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**.

__CUDACC_VER__
Defined with the full version number of **nvcc**, represented as **__CUDACC_VER_MAJOR__ * 10000 + __CUDACC_VER_MINOR__ * 100 + __CUDACC_VER_BUILD__** .

2.2. 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 will list the recognized file name suffixes and the supported compilation phases. A full explanation of the **nvcc** command line options can be found in the next chapter.

2.3. 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
`.gpu`	GPU intermediate file (see Figure 1)
`.ptx`	PTX intermediate assembly file (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.

2.4. 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>*

Phase	nvcc Option		Default Output File Name
	Long Name	Short Name	
C/C++ compilation to object file	`--compile`	`-c`	Source file name with suffix replaced by `o` on Linux and Mac OS X, or `obj` on Windows
Cubin generation from CUDA source files	`--cubin`	`-cubin`	Source file name with suffix replaced by `cubin`
Cubin generation from `.gpu` intermediate 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`
PTX generation from `.gpu` intermediate 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`
GPU C code generation from CUDA source files	`--gpu`	`-gpu`	Source file name with suffix replaced by `gpu`
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

Phase	nvcc Option		Default Output File Name
	Long Name	Short Name	
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>*
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 3.
NVCC COMMAND OPTIONS

3.1. 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.

3.2. 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.

3.2.1. Options for Specifying the Compilation Phase

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

Long Name	Short Name	Description
`--cuda`	`-cuda`	Compile all `.cu` input files to `.cu.cpp.ii` output.
`--cubin`	`-cubin`	Compile all `.cu`/`.gpu`/`.ptx` input files to device-only `.cubin` files. This step discards the host code for each `.cu` input file.
`--fatbin`	`-fatbin`	Compile all `.cu`/`.gpu`/`.ptx`/`.cubin` input files to device-only `.fatbin` files. This step discards the host code for each `.cu` input file.
`--ptx`	`-ptx`	Compile all `.cu`/`.gpu` input files to device-only `.ptx` files. This step discards the host code for each `.cu` input file.
`--gpu`	`-gpu`	Compile all `.cu` input files to device-only `.gpu` files. This step discards the host code for each `.cu` input file.
`--preprocess`	`-E`	Preprocess all `.c`/`.cc`/`.cpp`/`.cxx`/`.cu` input files.
`--generate-dependencies`	`-M`	Generate a dependency file that can be included in a `make` file for the `.c`/`.cc`/`.cpp`/`.cxx`/`.cu` input file (more than one are not allowed in this mode).
`--compile`	`-c`	Compile each `.c`/`.cc`/`.cpp`/`.cxx`/`.cu` input file into an object file.
`--device-c`	`-dc`	Compile each `.c`/`.cc`/`.cpp`/`.cxx`/`.cu` input file into an object file that contains relocatable device code. It is equivalent to `--relocatable-device-code=true --compile`.
`--device-w`	`-dw`	Compile each `.c`/`.cc`/`.cpp`/`.cxx`/`.cu` input file into an object file that contains executable device code. It is

Long Name	Short Name	Description
		equivalent to `--relocatable-device-code=false --compile`.
`--device-link`	`-dlink`	Link object files with relocatable device code and `.ptx`/`.cubin`/`.fatbin` files into an object file with executable device code, which can be passed to the host linker.
`--link`	`-link`	This option specifies the default behavior: compile and link all inputs.
`--lib`	`-lib`	Compile all input files into object files (if necessary), and add the results to the specified library output file.
`--run`	`-run`	This option compiles and links all inputs into an executable, and executes it. Or, 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`.

3.2.2. File and Path Specifications

Long Name	Short Name	Description
`--output-file` *file*	`-o`	Specify name and location of the output file. Only a single input file is allowed when this option is present in `nvcc` non-linking/archiving mode.
`--pre-include` *file,...*	`-include`	Specify header files that must be preincluded during preprocessing or compilation.
`--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`	Specify macro definitions for use during preprocessing or compilation.
`--undefine-macro` *def,...*	`-U`	Undefine macro definitions 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

Long Name	Short Name	Description
		dependency generation step (see **--generate-dependencies**) generate a rule that defines the target object file in the proper directory.
--compiler-bindir ***directory***	**-ccbin**	Specify the directory in which the 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.
--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 for this option: **none**, **shared**, **static**. Default value: **static**
--libdevice-directory ***directory***	**-ldir**	Specify the directory that contains the libdevice library files when option **--dont-use-profile** is used. Libdevice library files are located in the **nvvm/libdevice** directory in the CUDA Toolkit.

3.2.3. Options for Specifying Behavior of Compiler/Linker

Long Name	Short Name	Description
--profile	**-pg**	Instrument generated code/executable for use by **gprof** (Linux only).
--debug	**-g**	Generate debug information for host code.
--device-debug	**-G**	Generate debug information for device code.
--generate-line-info	**-lineinfo**	Generate line-number information for device code.
--optimize ***level***	**-O**	Specify optimization level for host code.
--ftemplate-backtrace-limit ***limit***	**-ftemplate-backtrace-limit**	When compiling CUDA source code, 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. Note: This flag does not affect the host compiler used for compiling host part of the CUDA source code.

Long Name	Short Name	Description
`--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 for this option: `c`, `c++`, `cu`.
`--std {c++11}`	`-std`	Select a particular C++ dialect. The only value currently supported is `c++11`. Enabling C++11 mode also turns on C++11 mode for the host compiler. Allowed value for this option: `c++11`
`--no-host-device-initializer-list`	`-nohdinitlist`	Do not implicitly consider member functions of `std::initializer_list` as `__host__ __device__` functions.
`--no-host-device-move-forward`	`-nohdmoveforward`	Do not implicitly consider `std::move` and `std::forward` as `__host__ __device__` function templates.
`--expt-relaxed-constexpr`	`-expt-relaxed-constexpr`	Experimental flag: Allow host code to invoke `__device__ constexpr` functions, and device code to invoke `__host__ constexpr` functions.
`--expt-extended-lambda`	`-expt-extended-lambda`	Experimental flag: Allow `__host__`, `__device__` annotations in lambda declaration.
`--machine {32|64}`	`-m`	Specify 32-bit vs. 64-bit architecture. Allowed values for this option: `32`, `64`.

3.2.4. 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.

Long Name	Short Name	Description
`--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 library manager.
`--ptxas-options` *options,...*	`-Xptxas`	Specify options directly to `ptxas`, the PTX optimizing assembler.

Long Name	Short Name	Description
--nvlink-options *options,...*	**-Xnvlink**	Specify options directly to **nvlink**.

3.2.5. Options for Guiding the Compiler Driver

Long Name	Short Name	Description
--dont-use-profile	**-noprof**	**nvcc** uses the **nvcc.profiles** file for compilation. When specifying this option, the profile file is not used.
--dryrun	**-dryrun**	Do not execute the compilation commands generated by **nvcc**. Instead, list them.
--verbose	**-v**	List the compilation commands generated by this compiler driver, but do not suppress their execution.
--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**	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**	Used in combination with option **--run** to specify command line arguments for the executable.
--input-drive-prefix *prefix*	**-idp**	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 **/** the ***prefix*** for MinGW.
--dependency-drive-prefix *prefix*	**-ddp**	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

Long Name	Short Name	Description
		prefix for MinGW. Or leave these file names in the native Windows format by specifying nothing.
--drive-prefix ***prefix***	**-dp**	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		Specifies 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.

3.2.6. Options for Steering CUDA Compilation

Long Name	Short Name	Description
--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 for this option: **legacy** The CUDA legacy stream (per context, implicitly synchronizes with other streams) **per-thread** A normal CUDA stream (per thread, does not implicitly synchronize with other streams) **null** is a deprecated alias for **legacy**. Default value: **legacy**

3.2.7. Options for Steering GPU Code Generation

Long Name	Short Name	Description
--gpu-architecture ***arch***	**-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_20). Normally, this option alone does not trigger assembly of the generated PTX for a *real* architecture (that is the role of **nvcc** option **--gpu-**

Long Name	Short Name	Description
		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_20), in which case **nvcc** uses the specified *real* architecture and its closest *virtual* architecture as effective architecture values. For example, **nvcc --gpu-architecture=sm_20** is equivalent to **nvcc --gpu-architecture=compute_20 --gpu-code=sm_20,compute_20**. See Virtual Architecture Feature List for the list of supported *virtual* architectures and GPU Feature List for the list of supported *real* architectures.
--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_20), and PTX code for the *virtual* architecture (such as compute_20). 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_35** is not compatible with **--gpu-code=sm_30**, because the earlier compilation stages will assume the availability of **compute_35** features that are not present on **sm_30**.

Long Name	Short Name	Description
		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 (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 for this option: **true**, **false**. Default value: **false**
--entries ***entry,...***	**-e**	Specify the global entry functions for which code must be generated. By default, code will be generated for all entries.
--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.

Long Name	Short Name	Description
		If this option is not specified, then no maximum is assumed. Value less than the minimum registers required by ABI will be bumped up by the compiler to ABI minimum limit.
--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**	This option controls 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 for this option: **true**, **false**. Default value: **false**
--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 for this option: **true**, **false**. Default value: **true**
--prec-sqrt {**true**|**false**}	**-prec-sqrt**	This option controls single-precision floating-point squre 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 for this option: **true**, **false**. Default value: **true**
--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 for this option: **true**, **false**. Default value: **true**

3.2.8. Generic Tool Options

Long Name	Short Name	Description
`--disable-warnings`	`-w`	Inhibit all warning messages.
`--source-in-ptx`	`-src-in-ptx`	Interleave source in PTX.
`--restrict`	`-restrict`	Programmer assertion that all kernel pointer parameters are restrict pointers.
`--Wno-deprecated-gpu-targets`	`-Wno-deprecated-gpu-targets`	Suppress warnings about deprecated GPU target architectures.
`--Werror` *kind,...*	`-Werror`	Make warnings of the specified kinds into errors. The following is the list of warning kinds accepted by this option: `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.
`--resource-usage`	`-res-usage`	Show resource usage such as registers and memeory 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.

3.2.9. Phase Options

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

3.2.9.1. Ptxas Options

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

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

Long Name	Short Name	Description
		If unspecified, default behavior is to enable this feature for optimization level >= O2.
--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_20, compute_30, compute_35, compute_50, compute_52; and sm_20, sm_21, sm_30, sm_32, sm_35, sm_50 and sm_52. Default value: sm_20.
--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.
--output-file *file*	-o	Specify name of output file. Default value: elf.o.

Long Name	Short Name	Description
`--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.

3.2.9.2. NVLINK Options

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

Long Name	Short Name	Description
`--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.

Chapter 4.
THE CUDA COMPILATION TRAJECTORY

The CUDA phase converts a source file coded in the extended CUDA language into a regular ANSI C++ source file that can be handed over to a general purpose C++ host compiler for further compilation and linking. The exact steps that are followed to achieve this are displayed in Figure 1.

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 preprocesed 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 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.

The CUDA compilation trajectory is more complicated in the separate compilation mode. For more information, see Using Separate Compilation in CUDA.

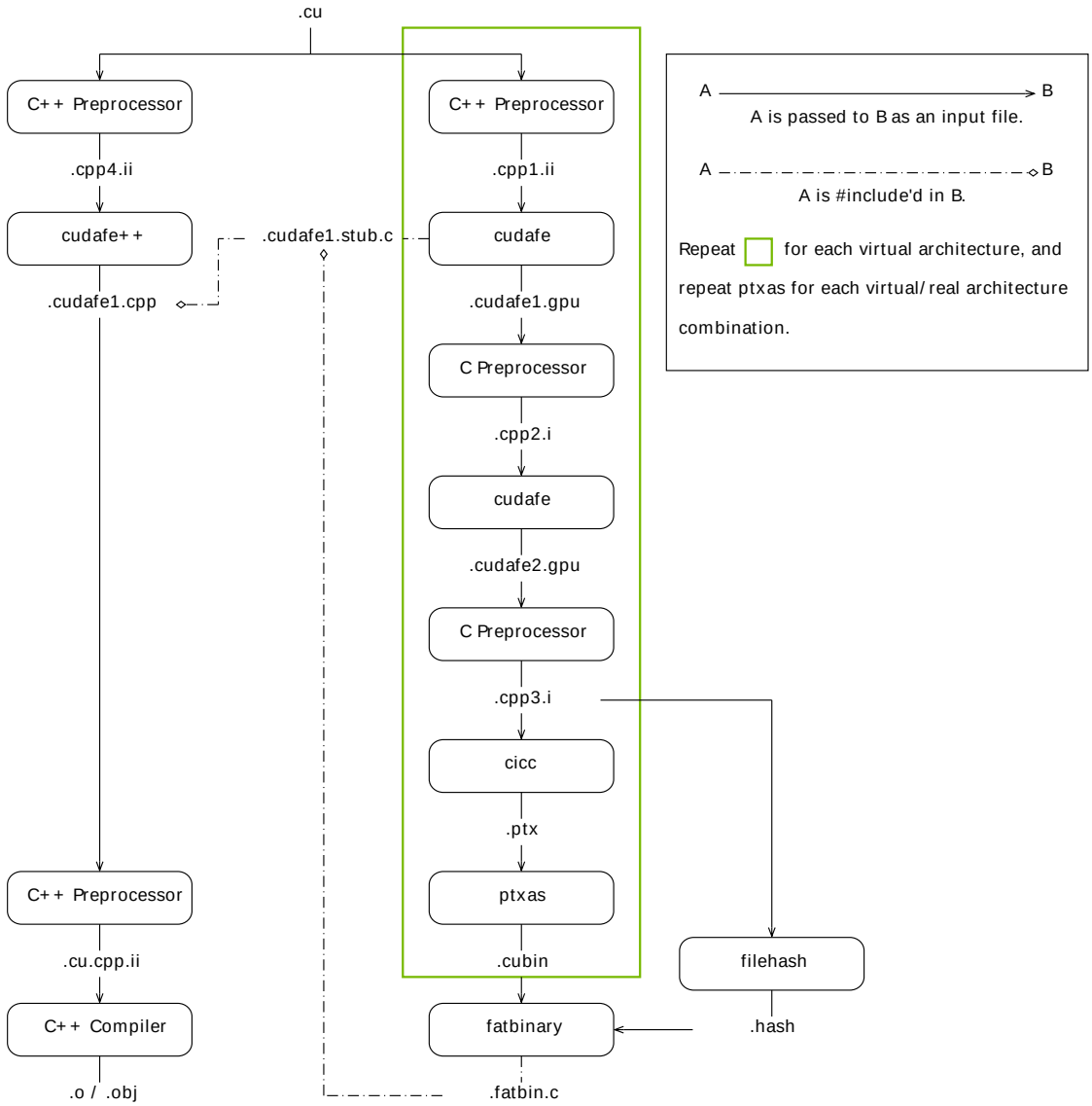

Figure 1 CUDA Whole Program Compilation Trajectory

Chapter 5. 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.

5.1. 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 geneartion 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* Kepler version **sm_30** whose functionality is a subset of all other Kepler versions: any code compiled for **sm_30** will run on all other Kepler GPUs.

5.2. 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 $\mathbf{x_1y_1} <= \mathbf{x_2y_2}$ then all non-ISA related capabilities of $\mathbf{sm\_x_1y_1}$ are included in those of

sm_x_2y_2. From this it indeed follows that **sm_30** is the *base* Kepler 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_30**'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_20	Basic features + Fermi support
sm_30 and **sm_32**	+ Kepler support + Unified memory programming
sm_35	+ Dynamic parallelism support
sm_50, **sm_52**, and **sm_53**	+ Maxwell support

5.3. 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.

5.4. 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.

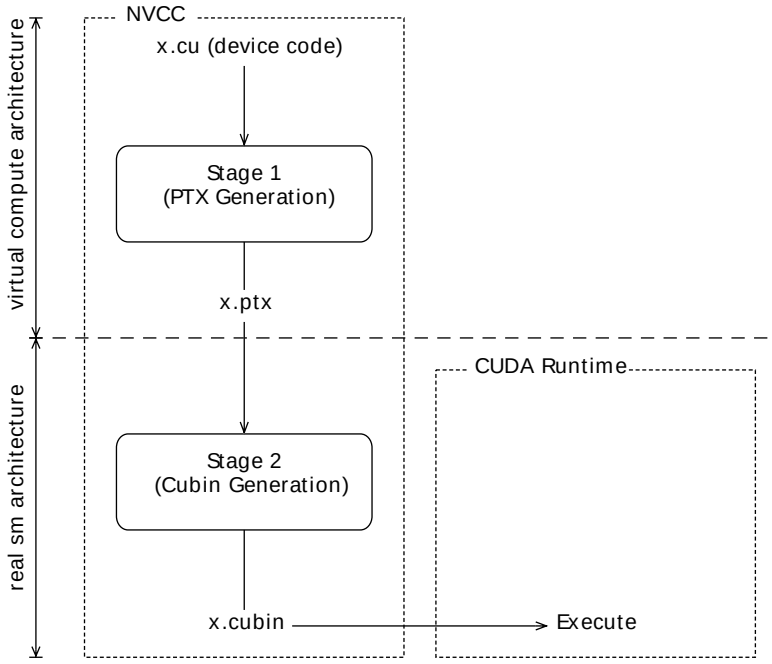

Figure 2 Two-Staged Compilation with Virtual and Real Architectures

5.5. Virtual Architecture Feature List

`compute_20`	Basic features + Fermi support
`compute_30` and `compute_32`	+ Kepler support + Unified memory programming
`compute_35`	+ Dynamic parallelism support
`compute_50`, `compute_52`, and `compute_53`	+ Maxwell 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.

5.6. 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.

5.6.1. 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_30`** allows the code to run on all Kepler-generation GPUs, but compiling to **`sm_35`** would probably yield better code if Kepler GK110 and later are the only targets.

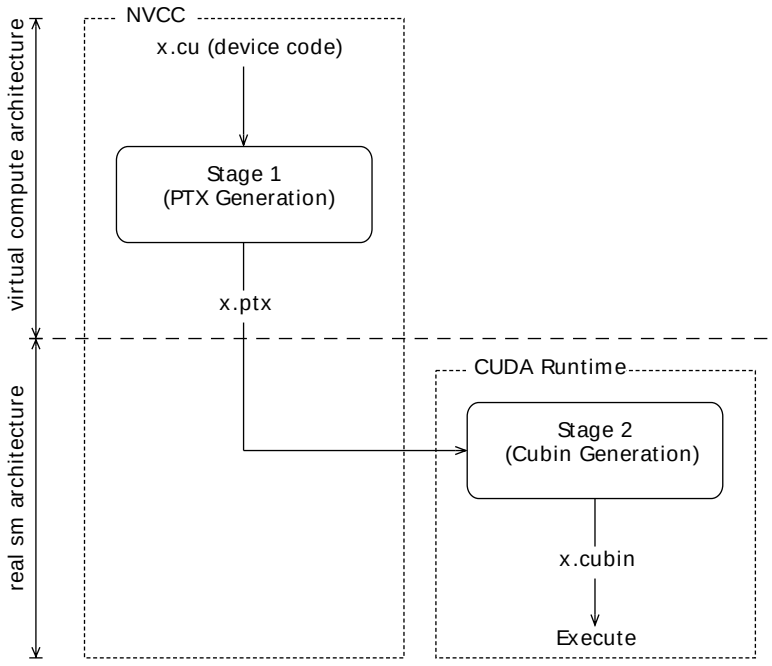

Figure 3 Just-in-Time Compilation of Device Code

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_20`** or later architecture.

```
nvcc x.cu --gpu-architecture=compute_20 --gpu-code=compute_20
```

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.

5.6.2. 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_30 --gpu-code=compute_30,sm_30,sm_35
```

This command generates exact code for two Kepler 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.

5.7. NVCC Examples

5.7.1. 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_30 --gpu-code=sm_30,sm_35
```

5.7.2. Shorthand

nvcc allows a number of shorthands for simple cases.

5.7.2.1. 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_30 --gpu-code=compute_30,sm_30,sm_35
```

5.7.2.2. 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_35
nvcc x.cu --gpu-architecture=compute_30
```

are equivalent to

```
nvcc x.cu --gpu-architecture=compute_35 --gpu-code=sm_35,compute_35
nvcc x.cu --gpu-architecture=compute_30 --gpu-code=compute_30
```

5.7.2.3. 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_20 --gpu-code=sm_20,compute_20
```

5.7.3. 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 warp shuffle support for both the **sm_20** code and the **sm_30** code:

```
nvcc x.cu --gpu-architecture=compute_20 --gpu-code=compute_20,sm_20,sm_30
```

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 warp shuffle support for the **sm_20** and **sm_21** code, but full support on **sm_3x**:

```
nvcc x.cu \
    --generate-code arch=compute_20,code=sm_20 \
    --generate-code arch=compute_20,code=sm_21 \
    --generate-code arch=compute_30,code=sm_30
```

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

```
nvcc x.cu \
    --generate-code arch=compute_20,code=compute_20 \
    --generate-code arch=compute_30,code=compute_30
```

The code sub-options can be combined, but for technical reasons must then be quoted, which causes a slightly more complex syntax:

```
nvcc x.cu \
    --generate-code arch=compute_20,code=\"sm_20,sm_21\" \
    --generate-code arch=compute_30,code=\"sm_30,sm_35\"
```

5.7.4. Virtual Architecture Identification Macro

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.

Chapter 6. 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.

6.1. 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.

6.2. 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 (**nvcc --device-c** has the same flow as Figure 1).

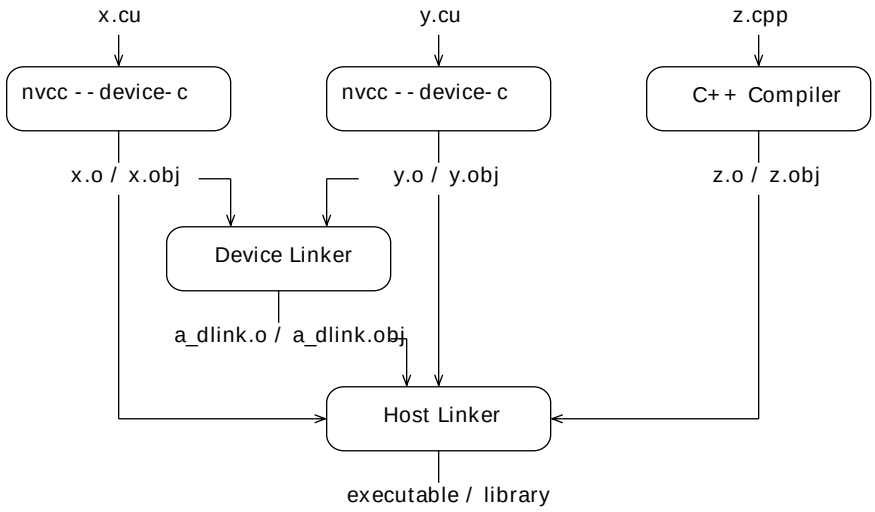

Figure 4 CUDA Separate Compilation Trajectory

6.3. 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_20 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_20 a.obj b.obj foo.lib --library-path=<path>
```

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

6.4. 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) {

  __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_20 --device-c a.cu b.cu
nvcc --gpu-architecture=sm_20 a.o b.o
```

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

```
nvcc --gpu-architecture=sm_20 --device-c a.cu b.cu
nvcc --gpu-architecture=sm_20 --device-link a.o b.o --output-file link.o
g++ a.o b.o link.o --library-path=<path> --library=cudart
```

Note that a target architecture must be passed to the device linker.

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

```
nvcc --gpu-architecture=sm_20 --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_20 --device-c a.cu b.cu
nvcc --lib a.o b.o --output-file test.a
nvcc --gpu-architecture=sm_20 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_20 --device-c a.ptx
```

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

```
nvcc --gpu-architecture=sm_35 --device-c a.cu b.cu
nvcc --gpu-architecture=sm_35 --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).

6.5. Potential Separate Compilation Issues

6.5.1. Object Compatibility

Only relocatable device code with the same ABI version, same SM target architecture, and same pointer size (32 or 64) can be linked together. Incompatible objects will produce a link error. 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 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.

6.5.2. JIT Linking Support

CUDA 5.0 does not support JIT linking, while CUDA 5.5 does. This means that to use JIT linking you must recompile your code with CUDA 5.5 or later. JIT linking means doing a relink of the code at startup time. The device linker (**nvlink**) links at the cubin level. 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.

6.5.3. 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***.

6.5.4. 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__ == 200
  return NULL; /* no address */
#else
  __shared__ T arr[256];
  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_20 --device-c a.cu
nvcc --gpu-architecture=compute_30 --device-c b.cu
nvcc --gpu-architecture=sm_30 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.

Chapter 7. MISCELLANEOUS NVCC USAGE

7.1. Cross Compilation

Cross compilation is controlled by using the following **nvcc** command line options:

- **--compiler-bindir** is used for cross compilation, where the underlying host compiler is capable of generating objects for the target platform.
- **--machine=32**. This option signals that the target platform is a 32-bit platform. Use this when the host platform is a 64-bit platform.

7.2. 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.

7.3. 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
```

7.4. 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
ptxas info    : 1536 bytes gmem, 8 bytes cmem[14]
ptxas info    : Compiling entry function 'acos_main' for 'sm_20'
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 amounts of statically allocated global memory (gmem) and constant memory in bank 14 (cmem) are 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.

Notice

Trademarks

Copyright