



CUDA COMPILER DRIVER NVCC

TRM-06721-001_v6.0 | February 2014

Reference Guide



CHANGES FROM PREVIOUS VERSION

- ▶ New support for separate compilation.
- ▶ Replaced Device Code Repositories with Using Separate Compilation in CUDA

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
1.3. Supported Build Environments.....	2
Chapter 2. Compilation Phases.....	4
2.1. NVCC Identification Macro.....	4
2.2. NVCC Phases.....	4
2.3. Supported Input File Suffixes.....	4
2.4. Supported Phases.....	5
2.5. Supported Phase Combinations.....	6
2.6. Keeping Intermediate Phase Files.....	7
2.7. Cleaning Up Generated Files.....	7
2.8. Use of Platform Compiler.....	7
2.8.1. Proper Compiler Installations.....	8
2.8.2. Non Proper Compiler Installations.....	8
2.9. cross compiling from x86 to ARMv7.....	8
2.10. nvcc.profile.....	8
2.10.1. Syntax.....	9
2.10.2. Environment Variable Expansion.....	9
2.10.3. HERE_, _SPACE_.....	9
2.10.4. Variables Interpreted by NVCC Itself.....	9
2.10.5. Example of profile.....	10
Chapter 3. NVCC Command Options.....	11
3.1. Command Option Types and Notation.....	11
3.2. Command Option Description.....	12
3.2.1. Options for Specifying the Compilation Phase.....	12
3.2.2. File and Path Specifications.....	13
3.2.3. Options for Altering Compiler/Linker Behavior.....	14
3.2.4. Options for Passing Specific Phase Options.....	15
3.2.5. Options for Guiding the Compiler Driver.....	15
3.2.6. Options for Steering CUDA Compilation.....	16
3.2.7. Options for Steering GPU Code Generation.....	17
3.2.8. Generic Tool Options.....	18
3.2.9. Phase Options.....	19
3.2.9.1. Ptxas Options.....	19
3.2.9.2. Nvlink Options.....	21
Chapter 4. The CUDA Compilation Trajectory.....	22

4.1. Listing and Rerunning NVCC Steps.....	22
4.2. Full CUDA Compilation Trajectory.....	25
4.2.1. Compilation Flow.....	26
4.2.2. CUDA Frontend.....	26
4.2.3. Preprocessing.....	26
Chapter 5. Sample NVCC Usage.....	27
Chapter 6. GPU Compilation.....	30
6.1. GPU Generations.....	30
6.2. GPU Feature List.....	31
6.3. Application Compatibility.....	31
6.4. Virtual Architectures.....	32
6.5. Virtual Architecture Feature List.....	33
6.6. Further Mechanisms.....	34
6.6.1. Just in Time Compilation.....	34
6.6.2. Fatbinaries.....	35
6.7. NVCC Examples.....	36
6.7.1. Base Notation.....	36
6.7.2. Shorthand.....	36
6.7.2.1. Shorthand 1.....	36
6.7.2.2. Shorthand 2.....	36
6.7.2.3. Shorthand 3.....	37
6.7.3. Extended Notation.....	37
6.7.4. Virtual Architecture Identification Macro.....	38
Chapter 7. Using Separate Compilation in CUDA.....	39
7.1. Code Changes for Separate Compilation.....	39
7.2. Default Behavior and sm_1x.....	39
7.3. NVCC Options for Separate Compilation.....	39
7.4. Libraries.....	41
7.5. Examples.....	41
7.6. Potential Separate Compilation Issues.....	43
7.6.1. Object Compatibility.....	43
7.6.2. JIT Linking Support.....	43
7.6.3. Implicit CUDA Host Code.....	43
Chapter 8. Miscellaneous NVCC Usage.....	45
8.1. Printing Code Generation Statistics.....	45

LIST OF FIGURES

Figure 1 Example of CUDA Source File	3
Figure 2 CUDA Compilation from .cu to .cu.cpp.o	25

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 computer (Linux, Windows), and which use one or more NVIDIA GPUs as coprocessors for accelerating SIMD 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.

Dispatching GPU jobs by the host process is supported by the CUDA Toolkit in the form of remote procedure calling. 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. See [Figure 1](#).

1.1.2. CUDA Sources

Hence, source files for CUDA applications consist of a mixture of conventional C++ host code, plus GPU device (i.e., GPU-) functions. The CUDA compilation trajectory separates the device functions from the host code, compiles the device functions using proprietary NVIDIA compilers/assemblers, compiles the host code using a general purpose C/C++ compiler that is available on the host platform, and afterwards embeds the compiled GPU functions as load images in the host object file. In the linking stage, specific CUDA runtime libraries are added for supporting remote SIMD 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

This compilation trajectory involves several splitting, compilation, preprocessing, and merging steps for each CUDA source file, and several of these steps are subtly different for different modes of CUDA compilation (such as compilation for device emulation, or the generation of device code repositories). It is the purpose of the CUDA compiler driver **nvcc** to hide the intricate details of CUDA compilation from developers. Additionally, instead of being a specific CUDA compilation driver, **nvcc** mimics the behavior of the GNU compiler **gcc**: 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 general purpose C compiler that is supported by **nvcc**, and on Windows platforms, where this compiler is an instance of the Microsoft Visual Studio compiler, **nvcc** will translate its options into appropriate **cl** command syntax. This extended behavior plus **cl** option translation is intended for support of portable application build and make scripts across Linux and Windows platforms.

1.2. Supported Host Compilers

nvcc uses the following compilers for host code compilation:

On Linux platforms

The GNU compiler, **gcc**, and **arm-linux-gnueabi-g++** for cross compilation to the ARMv7 architecture

On Windows platforms

The Microsoft Visual Studio compiler, **cl**

On both platforms, the compiler found on the current execution search path will be used, unless **nvcc** option **-compiler-bindir** is specified (see [File and Path Specifications](#)).

1.3. Supported Build Environments

nvcc can be used in the following build environments:

Linux

Any shell

Windows

DOS shell

Windows

CygWin shells, use **nvcc**'s drive prefix options (see [Options for Guiding the Compiler Driver](#)).

Windows:

MinGW shells, use **nvcc**'s drive prefix options (see [Options for Guiding the Compiler Driver](#)).

Although a variety of POSIX style shells is supported on Windows, **nvcc** will still assume the Microsoft Visual Studio compiler for host compilation. Use of **gcc** is not supported on Windows.

```

#define ACOS_TESTS      (5)
#define ACOS_THREAD_CNT (128)
#define ACOS_CTA_CNT    (96)

struct acosParams {
    float *arg;
    float *res;
    int n;
};

__global__ void acos_main (struct acosParams parms)
{
    int i;
    int totalThreads = gridDim.x * blockDim.x;
    int ctaStart = blockDim.x * blockIdx.x;
    for (i = ctaStart + threadIdx.x; i < parms.n; i += totalThreads) {
        parms.res[i] = acosf(parms.arg[i]);
    }
}

int main (int argc, char *argv[])
{
    volatile float acosRef;
    float* acosRes = 0;
    float* acosArg = 0;
    float* arg = 0;
    float* res = 0;
    float t;
    struct acosParams funcParams;
    int errors;
    int i;

    cudaMalloc ((void **)&acosArg, ACOS_TESTS * sizeof(float));
    cudaMalloc ((void **)&acosRes, ACOS_TESTS * sizeof(float));

    arg = (float *) malloc (ACOS_TESTS * sizeof(arg[0]));
    res = (float *) malloc (ACOS_TESTS * sizeof(res[0]));

    cudaMemcpy (acosArg, arg, ACOS_TESTS * sizeof(arg[0]),
                cudaMemcpyHostToDevice);

    funcParams.res = acosRes;
    funcParams.arg = acosArg;
    funcParams.n = opts.n;

    acos_main<<<ACOS_CTA_CNT,ACOS_THREAD_CNT>>>(funcParams);

    cudaMemcpy (res, acosRes, ACOS_TESTS * sizeof(res[0]),
                cudaMemcpyDeviceToHost);
}

```

Figure 1 Example of CUDA Source File

Chapter 2.

COMPILATION PHASES

2.1. NVCC Identification Macro

nvcc predefines the macro `__NVCC__`. This macro can be used in C/C++/CUDA source files to test whether they are currently being compiled by **nvcc**. In addition, **nvcc** predefines the macro `__CUDACC__`, which can be used in source files to test whether they are being treated as CUDA source files. The `__CUDACC__` macro can be particularly useful when writing header files.

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:

<code>.cu</code>	CUDA source file, containing host code and device functions
<code>.cup</code>	<i>Preprocessed</i> CUDA source file, containing host code and device functions
<code>.c</code>	C source file
<code>.cc</code> , <code>.cxx</code> , <code>.cpp</code>	C++ source file
<code>.gpu</code>	GPU intermediate file (see Figure 2)
<code>.ptx</code>	PTX intermediate assembly file (see Figure 2)
<code>.o</code> , <code>.obj</code>	Object file
<code>.a</code> , <code>.lib</code>	Library file
<code>.res</code>	Resource file
<code>.so</code>	Shared object file

Notes:

- ▶ **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.
- ▶ **nvcc** deviates from gcc behavior with respect to files whose suffixes are *unknown* (i.e., that do not occur in the above table): instead of assuming that these files must be linker input, **nvcc** will generate an error.

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 `-o`:

CUDA compilation to C/C++ source file	<code>-cuda</code>	<code>.cpp.i</code> appended to source file name, as in <code>x.cu.cpp.i</code> . This output file can be compiled by the host compiler that was used by nvcc to preprocess the <code>.cu</code> file
C/C++ preprocessing	<code>-E</code>	< result on standard output >
C/C++ compilation to object file	<code>-c</code>	Source file name with suffix replaced by <code>o</code> on Linux, or <code>obj</code> on Windows
Cubin generation from CUDA source files	<code>-cubin</code>	Source file name with suffix replaced by <code>cubin</code>

Cubin generation from <code>.gpu</code> intermediate files	<code>-cubin</code>	Source file name with suffix replaced by <code>cubin</code>
Cubin generation from <code>ptx</code> intermediate files.	<code>-cubin</code>	Source file name with suffix replaced by <code>cubin</code>
PTX generation from CUDA source files	<code>-ptx</code>	Source file name with suffix replaced by <code>ptx</code>
PTX generation from <code>.gpu</code> intermediate files	<code>-ptx</code>	Source file name with suffix replaced by <code>ptx</code>
Fatbin generation from source, <code>ptx</code> or <code>cubin</code> files	<code>-fatbin</code>	Source file name with suffix replaced by <code>fatbin</code>
GPU generation from CUDA source files	<code>-gpu</code>	Source file name with suffix replaced by <code>gpu</code>
Linking an executable, or <code>dll</code>	< no phase option >	<code>a.out</code> on Linux, or <code>a.exe</code> on Windows
Constructing an object file archive, or library	<code>-lib</code>	<code>a.a</code> on Linux, or <code>a.lib</code> on Windows
<code>make</code> dependency generation	<code>-M</code>	< result on standard output >
Running an executable	<code>-run</code>	-

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. Running using `nvcc` will automatically set the environment variables that are specified in `nvcc.profile` (see [Environment Variable Expansion](#)) prior to starting the executable.
- ▶ Files with extension `.cup` are assumed to be the result of preprocessing CUDA source files, by `nvcc` commands as `nvcc -E x.cu -o x.cup`, or `nvcc -E x.cu > x.cup`. Similar to regular compiler distributions, such as Microsoft Visual Studio or `gcc`, preprocessed source files are the best format to include in compiler bug reports. They are most likely to contain all information necessary for reproducing the bug.

2.5. Supported Phase Combinations

The following phase combinations are supported by `nvcc`:

- ▶ CUDA compilation to object file. This is a combination of CUDA Compilation and C compilation, and invoked by option `-c`.
- ▶ Preprocessing is usually implicitly performed as first step in compilation phases
- ▶ Unless a phase option is specified, `nvcc` will compile and link all its input files

- ▶ When `-lib` is specified, `nvcc` will compile all its input files, and store the resulting object files into the specified `archive/library`.

2.6. Keeping Intermediate Phase Files

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

Windows temp directory

Value of environment variable `TEMP`, or `c:/Windows/temp`

Linux temp directory

`/tmp`

Options `-keep` or `-save-temps` (these options are equivalent) will instead store these intermediate files in the current directory, with names as described in [Supported Phases](#).

2.7. 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`. This option is particularly useful after using `-keep`, because the keep option usually leaves quite an amount of intermediate files around.

Because using `-clean` 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 `-c` will have different cleanup effects.

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

2.8. Use of Platform Compiler

A general purpose C 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 (see also [The CUDA Compilation Trajectory](#)).

On Linux platforms, the compiler is assumed to be `gcc`, or `g++` for linking. On Windows platforms, the compiler is assumed to be `cl`. The compiler executables are expected to be in the current executable search path, unless option `--compiler-bindir` is specified, in which case the value of this option must be the name of the directory in which these compiler executables reside. This option is used for cross compilation to the ARMv7 architecture as well, where the underlying host compiler is required to be a gcc compiler, capable of generating ARMv7 code.

2.8.1. Proper Compiler Installations

On both Linux and Windows, *properly* installed compilers have some form of *internal knowledge* that enables them to locate system include files, system libraries and dlls, include files and libraries related to the compiler installation itself, and include files and libraries that implement `libc` and `libc++`.

A properly installed `gcc` compiler has this knowledge built in, while a properly installed Microsoft Visual Studio compiler has this knowledge available in a batch script `vsvars.bat`, at a known place in its installation tree. This script must be executed prior to running the `cl` compiler, in order to place the correct settings into specific environment variables that the `cl` compiler recognizes.

On Windows platforms, `nvcc` will locate `vsvars.bat` via the specified `--compiler-bindir` and execute it so that these environment variables become available.

On Linux platforms, `nvcc` will always assume that the compiler is properly installed.

2.8.2. Non Proper Compiler Installations

The platform compiler can still be *improperly* used, but in this case the user of `nvcc` is responsible for explicitly providing the correct include and library paths on the `nvcc` command line. Especially using `gcc` compilers, this requires intimate knowledge of `gcc` and Linux system issues, and these may vary over different `gcc` distributions. Therefore, this practice is not recommended

2.9. cross compiling from x86 to ARMv7

Cross compiling to the ARMv7 architecture is controlled by using the following `nvcc` command line options:

- ▶ `-target-cpu-arch ARM`. This option signals cross compilation to ARM.
- ▶ `-ccbin <arm-cross-compiler>`. This sets the host compiler with which `nvcc` cross compiles the host.
- ▶ `-m32`. This option signals that the target platform is a 32-bit platform. Use this when the host platform is a 64-bit x86 platform.

2.10. nvcc.profile

`nvcc` expects a configuration file `nvcc.profile` in the directory where the `nvcc` executable itself resides. This profile contains a sequence of assignments to environment variables which are necessary for correct execution of executables that `nvcc` invokes. Typical is extending the variables `PATH`, `LD_LIBRARY_PATH` with the bin and lib directories in the CUDA Toolkit installation.

The single purpose of `nvcc.profile` is to define the directory structure of the CUDA release tree to `nvcc`. It is not intended as a configuration file for `nvcc` users.

2.10.1. Syntax

Lines containing all spaces, or lines that start with zero or more spaces followed by a **#** character are considered comment lines. All other lines in `nvcc.profile` must have settings of either of the following forms:

```
name = <text>
name ?= <text>
name += <text>
name =+ <text>
```

Each of these three forms will cause an assignment to environment variable **name**: the specified text string will be macro- expanded (see [Environment Variable Expansion](#)) and assigned (=), or conditionally assigned (=?), or prepended (+=), or appended (=+)

2.10.2. Environment Variable Expansion

The assigned text strings may refer to the current value of environment variables by either of the following syntax:

```
%name%
  DOS style
$(name)
  make style
```

2.10.3. `HERE_`, `_SPACE_`

Prior to evaluating `nvcc.profile`, `nvcc` defines `_HERE_` to be directory path in which the profile file was found. Depending on how `nvcc` was invoked, this may be an absolute path or a relative path.

Similarly, `nvcc` will assign a single space string to `_SPACE_`. This variable can be used to enforce separation in profile lines such as:

```
INCLUDES += -I../common $_SPACE_
```

Omitting the `_SPACE_` could cause *glueing* effects such as `-I../common-Iapps` with previous values of `INCLUDES`.

2.10.4. Variables Interpreted by NVCC Itself

The following variables are used by `nvcc` itself:

<code>compiler-bindir</code>	The default value of the directory in which the host compiler resides (see Supported Host Compilers). This value can still be overridden by command line option <code>--compiler-bindir</code>
<code>INCLUDES</code>	This string extends the value of <code>nvcc</code> command option <code>-xcompiler</code> . It is intended for defining additional include paths. It is in actual compiler option syntax, i.e., <code>gcc</code> syntax on Linux and <code>cl</code> syntax on Windows.

LIBRARIES	This string extends the value of <code>nvcc</code> command option <code>-Xlinker</code> . It is intended for defining additional libraries and library search paths. It is in actual compiler option syntax, i.e., <code>gcc</code> syntax on Linux and <code>cl</code> syntax on Windows.
PTXAS_FLAGS	This string extends the value of <code>nvcc</code> command option <code>-Xptxas</code> . It is intended for passing optimization options to the CUDA internal tool <code>ptxas</code> .
OPENCC_FLAGS	This string extends the value of <code>nvcc</code> command line option <code>-Xopencc</code> . It is intended to pass optimization options to the CUDA internal tool <code>nvopencc</code> .

2.10.5. Example of profile

```
#
# nvcc and nvcc.profile are in the bin directory of the
# cuda installation tree. Hence, this installation tree
# is 'one up':
#
TOP          = $_HERE_/..

#
# Define the cuda include directories:
#
INCLUDES += -I$(TOP)/include -I$(TOP)/include/cudart ${_SPACE_}

#
# Extend dll search path to find cudart.dll and cuda.dll
# and add these two libraries to the link line
#
PATH        += $(TOP)/lib;
LIBRARIES   += ${_SPACE_} -L$(TOP)/lib -lcuda -lcudart
#
# Extend the executable search path to find the
# cuda internal tools:
#
PATH        += $(TOP)/open64/bin:$(TOP)/bin:

#
# Location of Microsoft Visual Studio compiler
#
compiler-bindir = c:/mvs/bin

#
# No special optimization flags for device code compilation:
#
PTXAS_FLAGS +=
```


Chapter 3.

NVCC COMMAND OPTIONS

3.1. Command Option Types and Notation

nvcc recognizes three types of command options: boolean (flag-) options, single value options, and list (multivalued-) 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 (multivalued-) options may be repeated. Examples of each of these option types are, respectively: `-v` (switch to verbose mode), `-o` (specify output file), and `-I` (specify include path).

Single value options and list options must have arguments, which must follow the name of the option itself by either one or more spaces or an equals character. In some cases of compatibility with `gcc` (such as `-I`, `-l`, and `-L`), the value of the option may also immediately follow the option itself, without being separated by spaces. The individual values of multivalued 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
-I dir1,dir2 -I=dir3 -I dir4,dir5
```

The option type in the tables in the remainder of this section 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.

Each 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. An example of this is the long alias of `-I`, which is `--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. For **nvcc**, this distinction may be of dubious value, because many of its options are well known compiler driver options, and the names of many other single-hyphen options were already chosen before **nvcc** was developed (and not especially short). However, the distinction is a useful convention, and the *short* options names may be shortened in future releases of the CUDA Toolkit.

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

3.2. Command Option Description

3.2.1. Options for Specifying the Compilation Phase

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

<code>--cuda</code>	<code>-cuda</code>	Compile all <code>.cu</code> input files to <code>.cu.cpp.i</code> output.
<code>--cubin</code>	<code>-cubin</code>	Compile all <code>.cu/.gpu/.ptx</code> input files to device-only <code>.cubin</code> files. This step discards the host code for each <code>.cu</code> input file.
<code>--ptx</code>	<code>-ptx</code>	Compile all <code>.cu/.gpu</code> input files to device-only <code>.ptx</code> files. This step discards the host code for each <code>.cu</code> input file.
<code>--gpu</code>	<code>-gpu</code>	Compile all <code>.cu</code> input files to device-only <code>.gpu</code> files. This step discards the host code for each <code>.cu</code> input file.
<code>--fatbin</code>	<code>-fatbin</code>	Compile all <code>.cu/.gpu/.ptx/.cubin</code> input files to device-only <code>.fatbin</code> files. This step discards the host code for each <code>.cu</code> input file.
<code>--preprocess</code>	<code>-E</code>	Preprocess all <code>.c/.cc/.cpp/.cxx/.cu</code> input files.
<code>--generate-dependencies</code>	<code>-M</code>	Generate for the one <code>.c/.cc/.cpp/.cxx/.cu</code> input file (more than one are not allowed in this step) a dependency file that can be included in a <code>make</code> file.
<code>--compile</code>	<code>-c</code>	Compile each <code>.c/.cc/.cpp/.cxx/.cu</code> input file into an object file.

<code>--link</code>	<code>-link</code>	This option specifies the default behavior: compile and link all inputs.
<code>--lib</code>	<code>-lib</code>	Compile all input files into object files (if necessary), and add the results to the specified library output file.
<code>--run</code>	<code>-run</code>	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. This step is intended for developers who do not want to be bothered with setting the necessary CUDA dll search paths (these will be set temporarily by nvcc according to the definitions in <code>nvcc.profile</code>).

3.2.2. File and Path Specifications

<code>--x language</code>	<code>-x</code>	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: <code>c,c++,cu</code> .
<code>--output-file file</code>	<code>-o</code>	Specify name and location of the output file. Only a single input file is allowed when this option is present in <code>nvcc</code> non-linking/archiving mode.
<code>--pre-include include-file,...</code>	<code>-include</code>	Specify header files that must be preincluded during preprocessing or compilation.
<code>--library library-file,...</code>	<code>-l</code>	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 <code>-I</code> (see Libraries).
<code>--define-macro macrodef,...</code>	<code>-D</code>	Specify macro definitions for use during preprocessing or compilation.
<code>--undefine-macro macrodef,...</code>	<code>-U</code>	Undefine a macro definition.

<code>--include-path <i>include-path</i>,...</code>	<code>-I</code>	Specify include search paths.
<code>--system-include <i>include-path</i>,...</code>	<code>-isystem</code>	Specify system include search paths.
<code>--library-path <i>library-path</i>,...</code>	<code>-L</code>	Specify library search paths (see Libraries).
<code>--output-directory <i>directory</i></code>	<code>-odir</code>	Specify the directory of the output file. This option is intended for letting the dependency generation step (<code>--generate-dependencies</code>) generate a rule that defines the target object file in the proper directory.
<code>--compiler-bindir <i>directory</i></code>	<code>-ccbin</code>	Specify the directory in which the host compiler executable (Microsoft Visual Studio <code>cl</code> , or a <code>gcc</code> derivative) resides. By default, this executable is expected in the current executable search path.
<code>--libdevice-directory <i>directory</i></code>	<code>-ldir</code>	Specify the directory that contains the libdevice library files when option <code>--dont-use-profile</code> is used. Libdevice library files are located in the <code>nvvm/libdevice</code> directory in the CUDA toolkit.

3.2.3. Options for Altering Compiler/Linker Behavior

<code>--profile</code>	<code>-pg</code>	Instrument generated code/executable for use by <code>gprof</code> (Linux only).
<code>--debug <i>level</i></code>	<code>-g</code>	Generate debug-able code.
<code>--device-debug</code>	<code>-G</code>	Generate debug-able device code.
<code>--generate-line-info</code>	<code>-lineinfo</code>	Generate line-number information for device code.
<code>--optimize <i>level</i></code>	<code>-O</code>	Generate optimized code.
<code>--shared</code>	<code>-shared</code>	Generate a shared library during linking. Note: when other linker options are required for controlling <code>dll</code> generation, use option <code>-xlinker</code> .
<code>--machine</code>	<code>-m</code>	Specify 32-bit vs. 64-bit architecture.

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.

<code>--compiler-options options,...</code>	<code>-Xcompiler</code>	Specify options directly to the compiler/preprocessor.
<code>--linker-options options,...</code>	<code>-Xlinker</code>	Specify options directly to the host linker.
<code>--opencc-options options,...</code>	<code>-Xopencc</code>	Specify options directly to <code>nvopencc</code> , typically for steering <code>nvopencc</code> optimization.
<code>--ptxas-options options,...</code>	<code>-Xptxas</code>	Specify options directly to the <code>ptx</code> optimizing assembler.
<code>--nvlink-options options,...</code>	<code>-Xnvlink</code>	Specify options directly to <code>nvlink</code> .

3.2.5. Options for Guiding the Compiler Driver

<code>--dryrun</code>	<code>-dryrun</code>	Do not execute the compilation commands generated by <code>nvcc</code> . Instead, list them.
<code>--verbose</code>	<code>-v</code>	List the compilation commands generated by this compiler driver, but do not suppress their execution.
<code>--keep</code>	<code>-keep</code>	Keep all intermediate files that are generated during internal compilation steps.
<code>--save-temps</code>	<code>-save-temps</code>	This option is an alias of <code>--keep</code> .
<code>--dont-use-profile</code>	<code>-noprof</code>	Do not use the <code>nvcc.profile</code> file to guide the compilation.
<code>--clean-targets</code>	<code>-clean</code>	This option reverses the behaviour of <code>nvcc</code> . When specified, none of the compilation phases will be executed. Instead, all of the non-temporary files that <code>nvcc</code> would otherwise create will be deleted.
<code>--run-args arguments,...</code>	<code>-run-args</code>	Used in combination with option <code>-R</code> , to specify command line arguments for the executable.
<code>--input-drive-prefix prefix</code>	<code>-idp</code>	On Windows platforms, all command line arguments that refer to file names must be converted to Windows native format before they are passed to pure Windows

		executables. This option specifies how the <i>current</i> development environment represents absolute paths. Use <code>-idp / cygwin/</code> for CygWin build environments, and <code>-idp /</code> for Mingw.
<code>--dependency-drive-prefix <i>prefix</i></code>	<code>-ddp</code>	On Windows platforms, when generating dependency files (option <code>-M</code>), all file names must be converted to whatever the used instance of <code>make</code> will recognize. Some instances of <code>make</code> have trouble with the colon in absolute paths in native Windows format, which depends on the environment in which this <code>make</code> instance has been compiled. Use <code>-ddp /cygwin/</code> for a CygWin <code>make</code> , and <code>-ddp /</code> for Mingw. Or leave these file names in native Windows format by specifying nothing.
<code>--drive-prefix <i>prefix</i></code>	<code>-dp</code>	Specifies <i>prefix</i> as both <code>input-drive-prefix</code> and <code>dependency-drive-prefix</code> .

3.2.6. Options for Steering CUDA Compilation

<code>--target-cpu-architecture</code>	<code>-target-cpu-arch</code>	Specify the name of the class of CPU architecture for which the input files must be compiled.
<code>--use_fast_math</code>	<code>-use_fast_math</code>	Make use of fast math library. <code>-use_fast_math</code> implies <code>-ftz=true -prec-div=false -prec-sqrt=false -fmad=true</code> .
<code>--ftz</code>	<code>-ftz</code>	The <code>-ftz</code> option controls single precision denormals support. When <code>-ftz=false</code> , denormals are supported and with <code>-ftz=true</code> , denormals are flushed to 0.
<code>--prec-div</code>	<code>-prec-div</code>	The <code>-prec-div</code> option controls single precision division. With <code>-prec-div=true</code> , the division is IEEE compliant, with <code>-prec-div=false</code> , the division is approximate.
<code>--prec-sqrt</code>	<code>-prec-sqrt</code>	The <code>-prec-sqrt</code> option controls single precision square root. With <code>-prec-sqrt=true</code> , the square root is IEEE compliant, with <code>-prec-sqrt=false</code> , the square root is approximate.
<code>--entries <i>entry,...</i></code>	<code>-e</code>	In case of compilation of <code>ptx</code> or <code>gpu</code> files to cubin: specify the global entry functions for which code must be generated. By default, code will be generated for all entries.
<code>--fmad</code>	<code>-fmad</code>	Enables (disables) the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add

		operations (FMAD, FFMA, or DFMA). The default is <code>-fmad=true</code> .
--	--	--

3.2.7. Options for Steering GPU Code Generation

<code>--gpu-architecture <i>gpuarch</i></code>	<code>-arch</code>	<p>Specify the name of the NVIDIA GPU to compile for. This can either be a <i>real</i> GPU, or a <i>virtual</i> PTX architecture. PTX code represents an intermediate format that can still be further compiled and optimized for, depending on the ptx version, a specific class of actual GPUs .</p> <p>The architecture specified by this option is the architecture that is assumed by the compilation chain up to the PTX stage, while the architecture(s) specified with the <code>-code</code> option are assumed by the last, potentially runtime, compilation stage.</p> <p>Currently supported compilation architectures are: virtual architectures <code>compute_10</code>, <code>compute_11</code>, <code>compute_12</code>, <code>compute_13</code>, <code>compute_20</code>, <code>compute_30</code>, <code>compute_32</code>, <code>compute_35</code>; and GPU architectures <code>sm_10</code>, <code>sm_11</code>, <code>sm_12</code>, <code>sm_13</code>, <code>sm_20</code>, <code>sm_21</code>, <code>sm_30</code>, <code>sm_32</code>, <code>sm_35</code>.</p>
<code>--gpu-code <i>gpuarch</i>,...</code>	<code>-code</code>	<p>Specify the name of the NVIDIA GPU to generate code for.</p> <p><code>nvcc</code> embeds a compiled code image in the executable for each specified <i>code</i> architecture, which is a true binary load image for each <i>real</i> architecture, and PTX code for each virtual architecture.</p> <p>During runtime, such embedded PTX code will be dynamically compiled by the CUDA runtime system if no binary load image is found for the <i>current</i> GPU.</p> <p>Architectures specified for options <code>-arch</code> and <code>-code</code> may be virtual as well as real, but the <i>code</i> architectures must be compatible with the <i>arch</i> architecture. When the <code>code</code> option is used, the value for the <code>-arch</code> option must be a virtual PTX architecture.</p> <p>For instance, <code>arch=compute_13</code> is not compatible with <code>code=sm_10</code>, because the earlier compilation stages will assume the availability of <code>compute_13</code> features that are not present on <code>sm_10</code>.</p> <p>This option defaults to the value of option <code>-arch</code>. Currently supported GPU architectures: <code>sm_10</code>, <code>sm_11</code>, <code>sm_12</code>,</p>

		<code>sm_13, sm_20, sm_21, sm_30, sm_32</code> and <code>sm_35</code> .
<code>--generate-code</code>	<code>-gencode</code>	<p>This option provides a generalization of the <code>-arch=<arch> -code=code,...</code> option combination for specifying <code>nvcc</code> behavior with respect to code generation. Where use of the previous options generates different code for a fixed virtual architecture, option <code>--generate-code</code> allows multiple <code>nvopencc</code> invocations, iterating over different virtual architectures. In fact, <code>-arch=<arch> -code=<code>,...</code> is equivalent to <code>--generate-code arch=<arch>,code=<code>,...</code></p> <p><code>--generate-code</code> options may be repeated for different virtual architectures.</p> <p>Allowed keywords for this option: <code>arch,code</code>.</p>
<code>--maxrregcount amount</code>	<code>-maxrregcount</code>	<p>Specify the maximum amount of registers that GPU functions can use.</p> <p>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 <code>maxrregcount</code> value is the result of a trade-off.</p> <p>If this option is not specified, then no maximum is assumed.</p> <p>Value less than the minimum registers required by ABI will be bumped up by the compiler to ABI minimum limit.</p>

3.2.8. Generic Tool Options

<code>--source-in-ptx</code>	<code>-src-in-ptx</code>	Interleave source in PTX.
<code>--Werror kind,...</code>	<code>-Werror</code>	<p>Make warnings of the specified kinds into errors. The following is the list of warning kinds accepted by this option:</p> <ul style="list-style-type: none"> ▶ <code>cross-execution-space-call</code> Be more strict about unsupported cross execution space calls. The compiler will generate an error

		instead of a warning for a call from a <code>__host__ __device__</code> to a <code>__host__</code> function.
<code>--help</code>	<code>-h</code>	Print help information on this tool.
<code>--version</code>	<code>-v</code>	Print version information on this tool.
<code>--options-file file,...</code>	<code>-optf</code>	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`.

<code>--allow-expensive-optimizations</code>	<code>-allow-expensive-optimizations</code>	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 \geq <code>o2</code> .
<code>--compile-only</code>	<code>-c</code>	Generate relocatable object.
<code>--def-load-cache</code>	<code>-dlcm</code>	Default cache modifier on global/generic load. Default value: <code>ca</code> .
<code>--def-store-cache</code>	<code>-dscm</code>	Default cache modifier on global/generic store.
<code>--gpu-name <i>gpuname</i></code>	<code>-arch</code>	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: <code>compute_10</code> , <code>compute_11</code> , <code>compute_12</code> , <code>compute_13</code> , <code>compute_20</code> , <code>compute_30</code> , <code>compute_35</code> ; and <code>sm_10</code> , <code>sm_11</code> , <code>sm_12</code> , <code>sm_13</code> , <code>sm_20</code> , <code>sm_21</code> , <code>sm_30</code> , <code>sm_32</code> and <code>sm_35</code> .

		Default value: <code>sm_10</code> .
<code>--opt-level N</code>	<code>-O</code>	Specify optimization level. Default value: 3.
<code>--output-file file</code>	<code>-o</code>	Specify name of output file. Default value: <code>elf.o</code> .
<code>--preserve-relocs</code>	<code>-preserve-relocs</code>	This option will make <code>ptxas</code> to generate relocatable references for variables and preserve relocations generated for them in linked executable.
<code>--sp-bound-check</code>	<code>-sp-bound-check</code>	Generate stack-pointer bounds-checking code sequence. This option is turned on automatically when <code>device-debug (-g)</code> or <code>opt-level (-O) 0</code> is specified.
<code>--disable-optimizer-constants</code>	<code>-disable-optimizer-consts</code>	Disable use of optimizer constant bank.
<code>--verbose</code>	<code>-v</code>	Enable verbose mode which prints code generation statistics.
<code>--warning-as-error</code>	<code>-Werror</code>	Make all warnings into errors.
<code>--device-debug</code>	<code>-g</code>	Semantics same as <code>nvcc</code> option <code>--device-debug</code> .
<code>--entry entry,...</code>	<code>-e</code>	Semantics same as <code>nvcc</code> option <code>--entries</code> .
<code>--fmad</code>	<code>-fmad</code>	Semantics same as <code>nvcc</code> option <code>--fmad</code> .
<code>--generate-line-info</code>	<code>-lineinfo</code>	Semantics same as <code>nvcc</code> option <code>--generate-line-info</code> .
<code>--machine</code>	<code>-m</code>	Semantics same as <code>nvcc</code> option <code>--machine</code> .
<code>--maxrregcount amount</code>	<code>-maxrregcount</code>	Semantics same as <code>nvcc</code> option <code>--maxrregcount</code> .
<code>--help</code>	<code>-h</code>	Semantics same as <code>nvcc</code> option <code>--help</code> .
<code>--options-file file,...</code>	<code>-optf</code>	Semantics same as <code>nvcc</code> option <code>--options-file</code> .
<code>--version</code>	<code>-V</code>	Semantics same as <code>nvcc</code> option <code>--version</code> .

3.2.9.2. Nvlink Options

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

<code>--preserve-relocs</code>	<code>-preserve-relocs</code>	Preserve resolved relocations in linked executable.
--------------------------------	-------------------------------	---

Chapter 4.

THE CUDA COMPILATION TRAJECTORY

This chapter explains the internal structure of the various CUDA compilation phases. These internals can usually be ignored unless one wants to understand, or *manually* rerun, the compilation steps corresponding to phases. Such command replay is useful during debugging of CUDA applications, when intermediate files need be inspected or modified. It is important to note that this structure reflects the current way in which **nvcc** implements its phases; it may significantly change with new releases of the CUDA Toolkit.

The following section illustrates how internal steps can be made visible by **nvcc**, and rerun. After that, a translation diagram of the **.cu** to **.cu.cpp.i** phase is listed. All other CUDA compilations are variants in some form of another of the **.cu** to C++ transformation.

4.1. Listing and Rerunning NVCC Steps

Intermediate steps can be made visible by options **-v** and **-dryrun**. In addition, option **-keep** might be specified to retain temporary files, and also to give them slightly more meaningful names. The following sample command lists the intermediate steps for a CUDA compilation:

```
nvcc -cuda x.cu --compiler-bindir=c:/mvs/vc/bin -keep -dryrun
```

This command results in a listing as the one shown at the end of this section.

Depending on the actual command shell that is used, the displayed commands are *almost* executable: the DOS command shell, and the Linux shells **sh** and **cs** each have slightly different notations for assigning values to environment variables.

The command list contains the following:

- ▶ Definition of standard variables **_HERE_** and **_SPACE_** (see [HERE_ _SPACE_](#))
- ▶ Environment assignments resulting from executing **nvcc.profile** (see [nvcc.profile](#))
- ▶ Definition of Visual Studio installation macros, derived from **-compiler-bindir** (see [Variables Interpreted by NVCC Itself](#))
- ▶ Environment assignments resulting from executing **vsvars32.bat**

- ▶ Commands generated by **nvcc**

```

#$ _SPACE =
#$ _HERE =c:\sw\gpgpu\bin\win32_debug
#$ TOP=c:\sw\gpgpu\bin\win32_debug/../../
#$ BINDIR=c:\sw\gpgpu\bin\win32_debug
#$
COMPILER_EXPORT=c:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/
win32_debug
#$
PATH=c:\sw\gpgpu\bin\win32_debug/open64/bin;c:\sw\gpgpu\bin\win32_debug;C:
\cygwin\usr\local\bin;C:\cygwin\bin;C:\cygwin\bin;C:\cygwin\usr\X11R6\bin;c:
\WINDOWS\system32;c:\WINDOWS;c:\WINDOWS\System32\Wbem;c:\Program Files\Microsoft
SQL Server\90\Tools\bin;c:\Program Files\Perforce;C:\cygwin\lib\lapack
#$
PATH=c:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/win32_debug/
open64/bin;c:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/
win32_debug/bin;c:\sw\gpgpu\bin\win32_debug/open64/bin;c:\sw\gpgpu\bin
\win32_debug;C:\cygwin\usr\local\bin;C:\cygwin\bin;C:\cygwin\bin;C:\cygwin\usr
\X11R6\bin;c:\WINDOWS\system32;c:\WINDOWS;c:\WINDOWS\System32\Wbem;c:\Program
Files\Microsoft SQL Server\90\Tools\bin;c:\Program Files\Perforce;C:\cygwin
\lib\lapack
#$ INCLUDES="-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/inc" "-Ic:\sw\gpgpu\bin
\win32_debug/../../cuda/tools/cudart"
#$ INCLUDES="-Ic:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/
win32_debug/include" "-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/inc" "-Ic:\sw
\gpgpu\bin\win32_debug/../../cuda/tools/cudart"
#$ LIBRARIES= "c:\sw\gpgpu\bin\win32_debug/cuda.lib" "c:\sw\gpgpu\bin
\win32_debug/cudart.lib"
#$ PTXAS_FLAGS=
#$ OPENC_C_FLAGS=-Werror
#$ VSINSTALLDIR=c:/mvs/vc/bin/..
#$ VCINSTALLDIR=c:/mvs/vc/bin/..
#$ FrameworkDir=c:\WINDOWS\Microsoft.NET\Framework
#$ FrameworkVersion=v2.0.50727
#$ FrameworkSDKDir=c:\MVS\SDK\v2.0
#$ DevEnvDir=c:\MVS\Common7\IDE
#$
PATH=c:\MVS\Common7\IDE;c:\MVS\VC\BIN;c:\MVS\Common7\Tools;c:\MVS
\Common7\Tools\bin;c:\MVS\VC\PlatformSDK\bin;c:\MVS\SDK\v2.0\bin;c:\WINDOWS
\Microsoft.NET\Framework\v2.0.50727;c:\MVS\VC\VCackages;c:\sw\gpgpu\bin
\win32_debug/../../../compiler/gpgpu/export/win32_debug/open64/bin;c:\sw\gpgpu
\bin\win32_debug/../../../compiler/gpgpu/export/win32_debug/bin;c:\sw\gpgpu
\bin\win32_debug/open64/bin;c:\sw\gpgpu\bin\win32_debug;C:\cygwin\usr\local
\bin;C:\cygwin\bin;C:\cygwin\bin;C:\cygwin\usr\X11R6\bin;c:\WINDOWS\system32;c:
\WINDOWS;c:\WINDOWS\System32\Wbem;c:\Program Files\Microsoft SQL Server\90\Tools
\bin;c:\Program Files\Perforce;C:\cygwin\lib\lapack
#$
INCLUDE=c:\MVS\VC\ATLMFC\INCLUDE;c:\MVS\VC\INCLUDE;c:\MVS\VC\PlatformSDK
\include;c:\MVS\SDK\v2.0\include;
#$
LIB=c:\MVS\VC\ATLMFC\LIB;c:\MVS\VC\LIB;c:\MVS\VC\PlatformSDK\lib;c:\MVS\SDK
\v2.0\lib;
#$
LIBPATH=c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727;c:\MVS\VC\ATLMFC\LIB
#$
PATH=c:/mvs/vc/bin;c:\MVS\Common7\IDE;c:\MVS\VC\BIN;c:\MVS\Common7\Tools;c:\MVS
\Common7\Tools\bin;c:\MVS\VC\PlatformSDK\bin;c:\MVS\SDK\v2.0\bin;c:\WINDOWS
\Microsoft.NET\Framework\v2.0.50727;c:\MVS\VC\VCackages;c:\sw\gpgpu\bin
\win32_debug/../../../compiler/gpgpu/export/win32_debug/open64/bin;c:\sw\gpgpu
\bin\win32_debug/../../../compiler/gpgpu/export/win32_debug/bin;c:\sw\gpgpu
\bin\win32_debug/open64/bin;c:\sw\gpgpu\bin\win32_debug;C:\cygwin\usr\local
\bin;C:\cygwin\bin;C:\cygwin\bin;C:\cygwin\usr\X11R6\bin;c:\WINDOWS\system32;c:
\WINDOWS;c:\WINDOWS\System32\Wbem;c:\Program Files\Microsoft SQL Server\90\Tools
\bin;c:\Program Files\Perforce;C:\cygwin\lib\lapack
#$ cudafe -E -DCUDA_NO_SM_12_ATOMIC_INTRINSICS -DCUDA_NO_SM_13_DOUBLE_INTRINSICS
-DCUDA_FLOAT_MATH_FUNCTIONS "-Ic:\sw\gpgpu\bin\win32_debug/../../../compiler/
gpgpu/export/win32_debug/include" "-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/inc"
"-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/tools/cudart" -I. "-Ic:\MVS\VC\ATLMFC
\INCLUDE" "-Ic:\MVS\VC\INCLUDE" "-Ic:\MVS\VC\PlatformSDK\include" "-Ic:\MVS\SDK
\v2.0\include" -D__CUDACC__ -C --preinclude "cuda_runtime.h" -o "x.cppl.i"
"x.cu"
#$ cudafe "-Ic:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/
win32_debug/include" "-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/inc" "-Ic:
\sw\gpgpu\bin\win32_debug/../../cuda/tools/cudart" -I. --gen_c_file_name
"x.cudafel.c" --gen_device_file_name "x.cudafel.gpu" --include_file_name
x.fatbin.c --no_exceptions -tused "x.cppl.i"
#$ cudafe -E --C -DCUDA_NO_SM_12_ATOMIC_INTRINSICS -

```

4.2. Full CUDA Compilation Trajectory

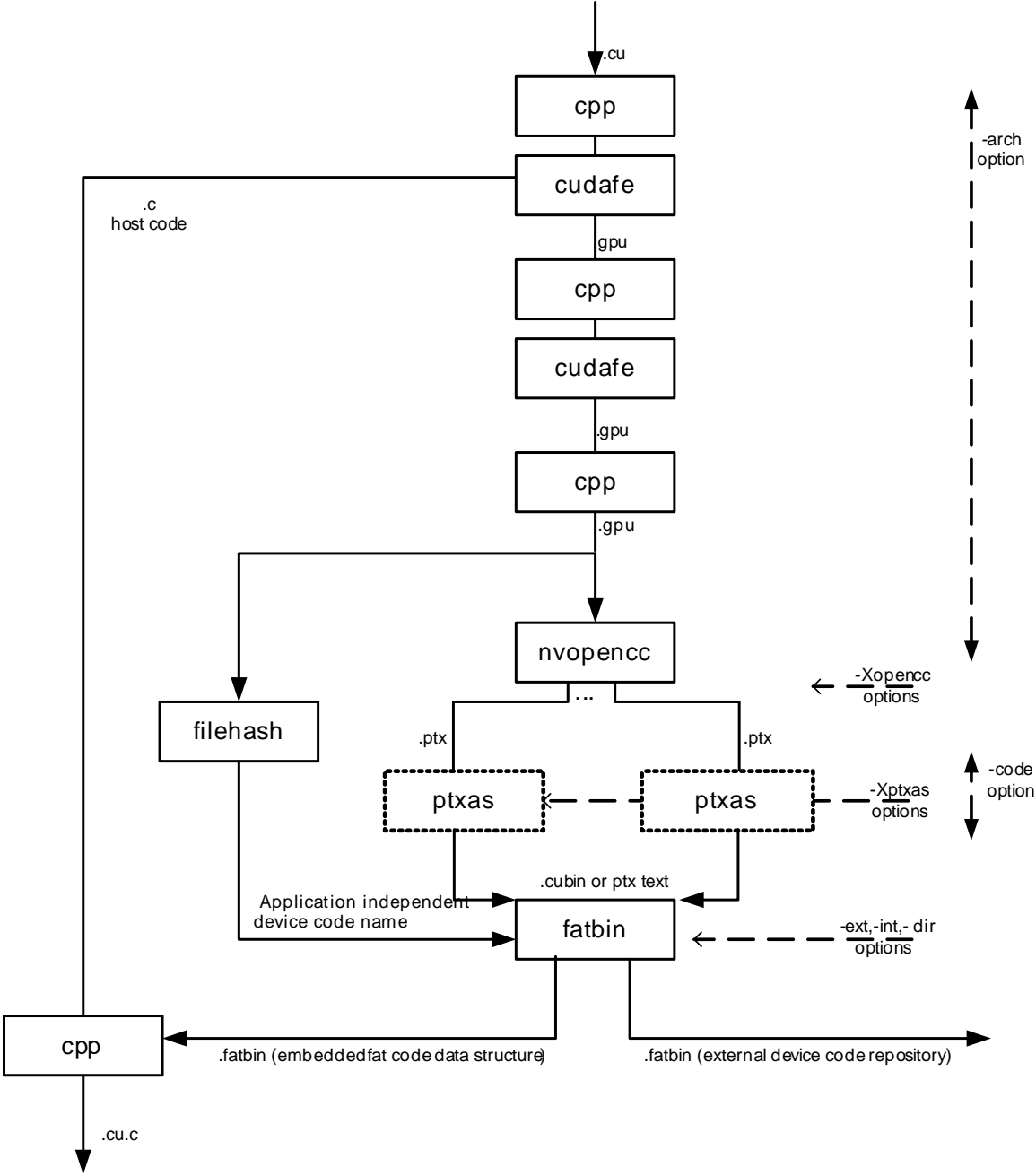


Figure 2 CUDA Compilation from .cu to .cu.cpp.ii

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++ compiler for further compilation and linking. The exact steps that are followed to achieve this are displayed in Figure 2.

4.2.1. Compilation Flow

In short, CUDA compilation works as follows: the input program is separated by the CUDA front end (**cudafe**), into C/C++ host code and the **.gpu** device code. Depending on the value(s) of the **-code** option to **nvcc**, this device code is further translated by the CUDA compilers/assemblers into CUDA binary (**cubin**) and/or into intermediate PTX code. This code is merged into a device code descriptor which is included by the previously separated host code. This descriptor will be inspected by the CUDA runtime system whenever the device code is invoked (*called*) by the host program, in order to obtain an appropriate load image for the current GPU.

4.2.2. CUDA Frontend

In the current CUDA compilation scheme, the CUDA front end is invoked twice. The first step is for the actual splitup of the **.cu** input into host and device code. The second step is a technical detail (it performs dead code analysis on the **.gpu** generated by the first step), and it might disappear in future releases.

4.2.3. Preprocessing

The trajectory contains a number of preprocessing steps. The first of these, on the **.cu** input, has the usual purpose of expanding include files and macro invocations that are present in the source file. The remaining preprocessing steps expand CUDA system macros in (C-) code that has been generated by preceding CUDA compilation steps. The last preprocessing step also merges the results of the previously diverged compilation flow.

Chapter 5.

SAMPLE NVCC USAGE

The following lists a sample **makefile** that uses **nvcc** for portability across Windows and Linux.

```
#
# On windows, store location of Visual Studio compiler
# into the environment. This will be picked up by nvcc,
# even without explicitly being passed.
# On Linux, use whatever gcc is in the current path
# (so leave compiler-bindir undefined):
#
ifdef ON_WINDOWS
    export compiler-bindir := c:/mvs/bin
endif

#
# Similar for OPENCC_FLAGS and PTXAS_FLAGS.
# These are simply passed via the environment:
#
export OPENCC_FLAGS :=
export PTXAS_FLAGS := -fastimul

#
# cuda and C/C++ compilation rules, with
# dependency generation:
#
%.o : %.cpp
$(NVCC) -c %^ $(CFLAGS) -o $@
$(NVCC) -M %^ $(CFLAGS) > $@.dep

%.o : %.c
$(NVCC) -c %^ $(CFLAGS) -o $@
$(NVCC) -M %^ $(CFLAGS) > $@.dep

%.o : %.cu
$(NVCC) -c %^ $(CFLAGS) -o $@
$(NVCC) -M %^ $(CFLAGS) > $@.dep

#
# Pick up generated dependency files, and
# add /dev/null because gmake does not consider
# an empty list to be a list:
#
include $(wildcard *.dep) /dev/null

#
# Define the application;
# for each object file, there must be a
# corresponding .c or .cpp or .cu file:
#
OBJECTS = a.o    b.o    c.o
APP      = app

$(APP) : $(OBJECTS)
$(NVCC) $(OBJECTS) $(LDFLAGS) -o $@
```


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 next generation graphics card (and vice versa). This is because the Fermi instruction set and instruction encodings is different from Kepler, which in turn will probably be substantially different from those of the next generation GPU.

Tesla

- ▶ `sm_10`
- ▶ `sm_11`
- ▶ `sm_12`
- ▶ `sm_13`

Fermi

- ▶ `sm_20`
- ▶ `sm_21`

Kepler

- ▶ `sm_30`
- ▶ `sm_35`
- ▶ `..??..`

Next generation

- ▶ ..??..

Because they share the basic instruction set, binary compatibility within one GPU generation, however, can under certain conditions be guaranteed. 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* Tesla version `sm_10` whose functionality is a subset of all other Tesla versions: any code compiled for `sm_10` will run on all other Tesla 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_10` is the *base* Tesla 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_10`'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`.

<code>sm_10</code>	ISA_1 Basic features
<code>sm_11</code>	+ atomic memory operations on global memory
<code>sm_12</code>	+ atomic memory operations on shared memory + vote instructions
<code>sm_13</code>	+ double precision floating point support
<code>sm_20</code>	+ Fermi support
<code>sm_30</code>	+ Kepler support
<code>sm_35</code>	+ dynamic parallelism 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.

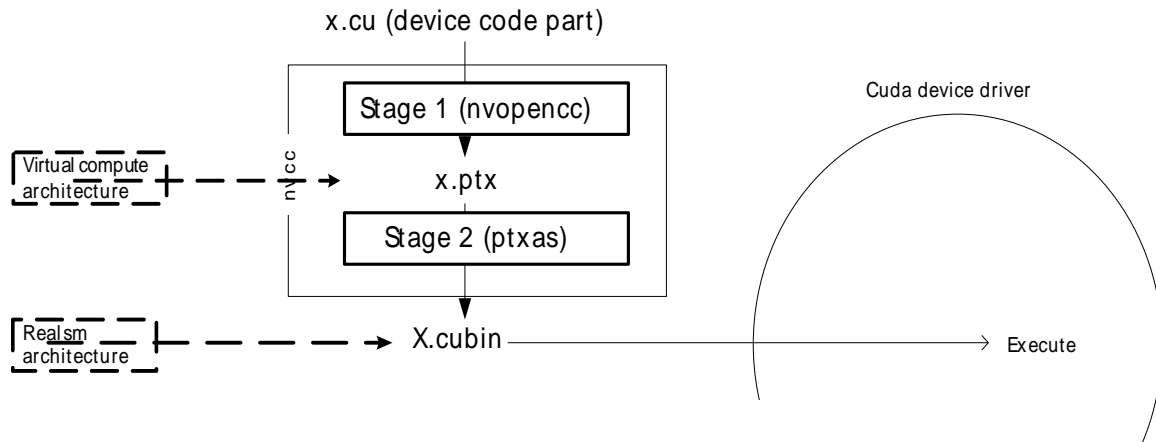
6.4. Virtual Architectures

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 *compute* architecture to specify the 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 (somehow or another) 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 *compute* architecture should always be chosen as *low* as possible, thereby maximizing the actual GPUs to run on. The *real* sm 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.



6.5. Virtual Architecture Feature List

<code>compute_10</code>	Basic features
<code>compute_11</code>	+ atomic memory operations on global memory
<code>compute_12</code>	+ atomic memory operations on shared memory + vote instructions
<code>compute_13</code>	+ double precision floating point support
<code>compute_20</code>	+ Fermi support
<code>compute_30</code>	+ Kepler support

The above table lists the currently defined virtual architectures. As it appears, this table shows a 1-1 correspondence to the table of actual GPUs listed earlier in this chapter. The only difference except for the architecture names is that the ISA specification is missing for the compute architectures.

However, this correspondence is misleading, and might degrade when new GPU architectures are introduced and also due to development of the CUDA compiler.

First, a next generation architecture might not provide any functional improvements, in which case the list of *real* architectures will be extended (because we must be able to generate code for this architecture), but no new compute architecture is necessary.

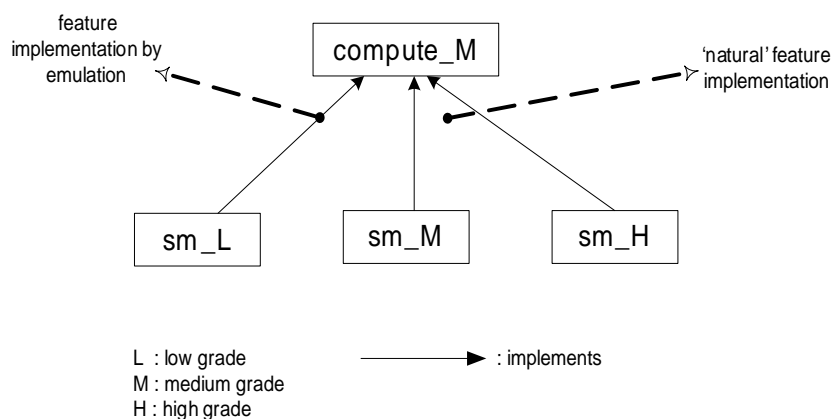
Second, it may be decided to let the compiler emulate certain *higher* grade features on *lower* grade GPUs. For example, this might be done for double precision floating

point support. In this case double precision based applications will run on all *real* GPU architectures, though with considerably lower performance on the models that do not provide native double support. Such double precision emulation is here used merely as an example (it currently is not actually considered), but the CUDA compiler already does emulation for features that are considered *basic* though not natively supported: integer division and 64-bit integer arithmetic. Because integer division and 64-bit integer support are part of the basic feature set, they will not explicitly show up in the features tables.

Feature emulation might have two different consequences for the virtual architecture table: the feature might be silently added to a lower grade virtual architecture (as has happened for integer division and 64-bit arithmetic), or it could be kept in a separate virtual architecture. For instance if we were to emulate double precision floating point on an `sm_10`, then keeping the virtual architecture `compute_13` would make sense because of the drastic performance consequences: applications would then have to explicitly *enable* it during `nvcc` compilation and there would therefore be no danger of unwittingly using it on lower grade GPUs. Either way, the following `nvcc` command would become valid (which currently is not the case):

```
nvcc x.cu -arch=compute_13 -code=sm_10
```

The two cases of feature implementation are further illustrated below:



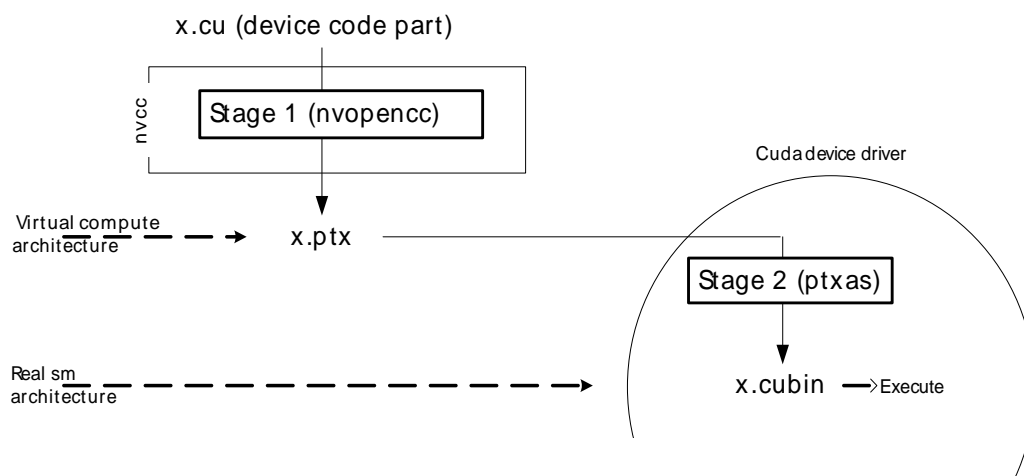
6.6. Further Mechanisms

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, for Tesla, compiling to **sm_10** allows the code to run on all Tesla versions, but compiling to **sm_13** would probably yield better code.



By specifying a virtual code architecture instead of a *real* GPU, `nvcc` postpones the second compilation stage 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 **ansm_10**, an **sm_13**, and even a later architecture

```
nvcc x.cu -arch=compute_10 -code=compute_10
```

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.

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 -arch=compute_10 -code=compute_10,sm_10,sm_13
```

This command generates exact code for two Tesla 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 **-arch** and **-code** for specifying the target architectures for both translation stages. Except for allowed short hands described below, the **-arch** option takes a single value, which must be the name of a virtual compute architecture, while option **-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 -arch=compute_10 -code=sm_10,sm_13
```

6.7.2. Shorthand

nvcc allows a number of shorthands for simple cases.

6.7.2.1. Shorthand 1

-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 -arch=compute_10 -code=compute_10,sm_10,sm_13
```

6.7.2.2. Shorthand 2

The **-code** option can be omitted. Only in this case, the **-arch** value can be a non-virtual architecture. The **-code** values default to the closest virtual architecture that is implemented by the GPU specified with **-arch**, plus the **-arch** value itself (in case the **-arch** value is a virtual architecture then these two are the same, resulting in a single-**code** default). After that, the effective **-arch** value will be the *closest* virtual architecture:

Example

```
nvcc x.cu -arch=sm_13
nvcc x.cu -arch=compute_10
```

are short hands for

```
nvcc x.cu -arch=compute_13 -code=sm_13,compute_13
nvcc x.cu -arch=compute_10 -code=compute_10
```

6.7.2.3. Shorthand 3

Both `-arch` and `-code` options can be omitted.

Example

```
nvcc x.cu
```

is short hand for

```
nvcc x.cu -arch=compute_10 -code=sm_10,compute_10
```

6.7.3. Extended Notation

The options `-arch` and `-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 double precision floating point support for both the `sm_10` code and the `sm_13` code:

```
nvcc x.cu -arch=compute_10 -code=compute_10,sm_10,sm_13
```

Sometimes it is necessary to perform different GPU code generation steps, partitioned over different architectures. This is possible using `nvcc` option `-gencode`, which then must be used instead of a `-arch/-code` combination.

Unlike option `-arch`, option `-gencode` 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 double precision support for the `sm_10` and `sm_11` code, but full support for `sm_13`:

```
nvcc x.cu \
-gencode arch=compute_10,code=sm_10 \
-gencode arch=compute_10,code=sm_11 \
-gencode arch=compute_13,code=sm_13
```

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

```
nvcc x.cu \
-gencode arch=compute_10,code=compute_10 \
-gencode arch=compute_13,code=compute_13
```

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

```
nvcc x.cu \  
-gencode arch=compute_10,code=\'sm_10,sm_11\' \  
-gencode arch=compute_12,code=\'sm_12,sm_13\'
```

6.7.4. 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 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. Default Behavior and `sm_1x`

Separate compilation only works for `sm_20` and above, because there is no ABI for `sm_1x`. `sm_10` is also the default target architecture, so a default compile will be whole program compilation. Because multiple architectures can be specified on the same command line, we have left the default as whole program compilation for all architectures. So a user must explicitly opt-in to separate compilation by specifying new commands, which are described in the next section.

7.3. 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 the device linker (`nvlink`) 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={true,false}` option, which can be shortened to `-rdc={true,false}`.

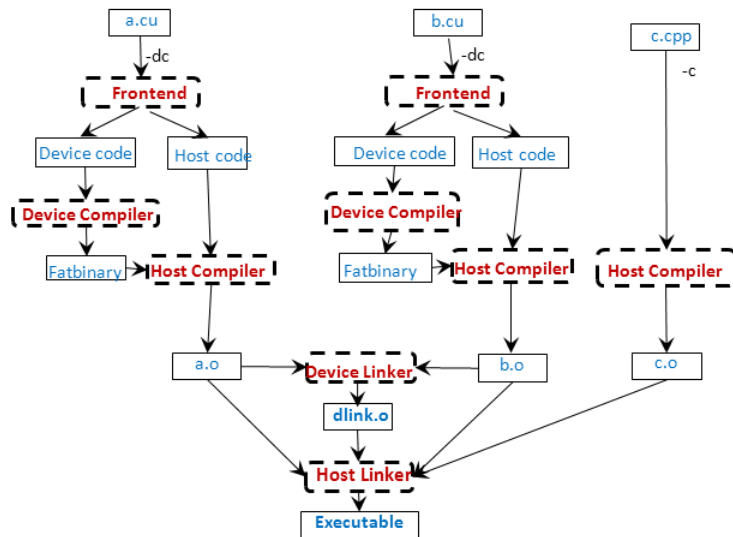
The `-c` option is already used to control stopping a compile at a host object, so a new option `--device-c` (or `-dc`) is added that simply does `-c --relocatable-device-code=true`.

To invoke just the device linker, the `--device-link` (`-dlink`) 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 as long as the architecture is `> sm_20`. This works because if the device linker does not see any relocatable code it does not do anything.

A diagram of the flow is as follows:



7.4. Libraries

The device linker has the ability to read the static host library formats (`.a` on Linux and Mac, `.lib` on Windows). It ignores any dynamic (`.so` or `.dll`) libraries. The `-l` and `-L` 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 `-l` option is used.

```
nvcc -arch=sm_20 a.o b.o -L<path> -lfoo
```

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

```
nvcc -arch=sm_20 a.obj b.obj foo.lib -L<path>
```

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

7.5. 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 -arch=sm_20 -dc a.cu b.cu
nvcc -arch=sm_20 a.o b.o

```

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

```

nvcc -arch=sm_20 -dc a.cu b.cu
nvcc -arch=sm_20 -dlink a.o b.o -o link.o
g++ a.o b.o link.o -L<path> -lcudart

```

Note that a target architecture must be passed to the device linker. If you invoke the device linker without a target arch, e.g.,

```

nvcc -dlink a.o b.o

```

You will get an error because that defaults to **sm_10**.

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


```
nvcc -arch=sm_20 -dc a.cu b.cu
nvcc -lib a.o b.o -o test.a
nvcc -arch=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 -arch=sm_20 -dc a.ptx
```

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

```
nvcc -arch=sm_35 -dc a.cu b.cu
nvcc -arch=sm_35 -dlink a.o b.o -lcudadevrt -o link.o
g++ a.o b.o link.o -lcudadevrt -L<path> -lcudart
```

It is possible to do multiple device links within a single host executable, as long as each device link is independent of the other (they cannot share code across device *executables*).

7.6. Potential Separate Compilation Issues

7.6.1. 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.

7.6.2. 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.

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).

Chapter 8.

MISCELLANEOUS NVCC USAGE

8.1. 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 **-v** to **ptxas**:

```
nvcc -Xptxas -v acos.cu
ptxas info    : Compiling entry function 'acos_main'
ptxas info    : Used 4 registers, 60+56 bytes lmem, 44+40 bytes smem,
                20 bytes cmem[1], 12 bytes cmem[14]
```

As shown in the above example, the amounts of local and shared memory are listed by two numbers each. First number represents the total size of all the variables declared in that memory segment and the second number represents the amount of system allocated data. The amount and location of system allocated data as well as the allocation of constant variables to constant banks is profile specific. For constant memory, the total space allocated in that bank is shown.

If separate compilation is used, some of this info must come from the device linker, so should use **nvcc -Xnvlink -v**.

Notice

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.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

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.

Copyright

© 2007-2014 NVIDIA Corporation. All rights reserved.