The CUDA Compiler Driver NVCC

Last modified on: <10-18-2011>

Introduction

Overview

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.

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.

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.

Supported host compilers

Nvcc will use the following compilers for host code compilation:

On Linux platforms:	The GNU compiler, gcc
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 page 13).

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 page 14).
Windows	MinGW shells, use nvcc's drive prefix options (see page 14).

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

Compilation Phases

Nvcc identification macro

Nvcc predefines the macro __CUDACC__. This macro can be used in sources to test whether they are currently being compiled by nvcc.

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.

Supported input file suffixes

The following table defines how nvcc interprets its input files

.cu	CUDA source file, containing host code and device functions
.cup	*Preprocessed* CUDA source file, containing host code and device functions
.c	'C' source file
.cc, .cxx, .cpp	C++ source file
.gpu	Gpu intermediate file (see 0)
.ptx	Ptx intermeditate assembly file (see 0)

.o, .obj	Object file
.a, .lib	Library file
.res	Resource file
.so	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.

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	-cuda	.c/.cpp appended to source file name, as in x.cu.c/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++ preprocesing	-E	< result on standard output >
C/C++ compilation to object file	-c	Source file name with suffix replaced by "o" on Linux, or "obj" on Windows
Cubin generation from CUDA source files	-cubin	Source file name with suffix replaced by "cubin"
Cubin generation from .gpu intermediate files	-cubin	Source file name with suffix replaced by "cubin"
Cubin generation from Ptx intermediate files.	-cubin	Source file name with suffix replaced by "cubin"
Ptx generation from CUDA source files	-ptx	Source file name with suffix replaced by "ptx"
Ptx generation from .gpu intermediate files	-ptx	Source file name with suffix replaced by "ptx"
Fatbin generation from source, ptx or cubin files	-fatbin	Source file name with suffix replaced by "fatbin"
Gpu generation from CUDA source files	-gpu	Source file name with suffix replaced by "gpu"
Linking an executable, or dll	< no phase option >	a.out on Linux, or a.exe on Windows
Constructing an object file archive, or library	-lib	a.a on Linux, or a.lib on Windows
'Make' dependency generation	-M	< result on standard output >

Running an executable	-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. Running using nvcc will automatically set the environment variables that are specified in nvcc.profile (see page 8) 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.

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.

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 the table on page 5.

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.

Example:

```
nvcc acos.cu –keep
nvcc acos.cu –keep –clean
```

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 above example, omitting –keep, or adding –c will have different cleanup effects.

Use of platform compiler

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

1. During non-CUDA phases (except the run phase), because these phases will be forwarded by nvcc to this compiler
2. During CUDA phases, for several preprocessing stages (see also chapter "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.

'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 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.

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.

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.

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 next section) and assigned ('='), or conditionally assigned ('?='), or prepended ('+='), or appended ('=+').

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

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.

Variables interpreted by nvcc itself

The following variables are used by nvcc itself:

compiler-bindir	The default value of the directory in which the host compiler resides (see Section 0). This value can still be overridden by command line *option --compiler-bindir*
INCLUDES	This string extends the value of nvcc command option –Xcompiler. It is intended for defining additional include paths. It is in actual compiler option syntax, i.e. gcc syntax on Linux and cl syntax on Windows.
LIBRARIES	This string extends the value of nvcc command option –Xlinker. It is intended for defining additional libraries and library search paths. It is in actual compiler option syntax, i.e. gcc syntax on Linux and cl syntax on Windows.
PTXAS_FLAGS	This string extends the value of nvcc command option –Xptxas. It is intended for passing optimization options to the CUDA internal tool ptxas.
OPENCC_FLAGS	This string extends the value of nvcc command line option –Xopencc. It is intended to pass optimization options to the CUDA internal tool nvopencc.

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      +=
```

Nvcc Command Options

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 of 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

-Idir1,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 can be used interchangedly. 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.

Command option description

Options for specifying the compilation phase

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

--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.
--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.
--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.
--preprocess	-E	Preprocess all .c/.cc/.cpp/.cxx/.cu input files.
--generate-dependencies	-M	Generate for the one .c/.cc/.cpp/.cxx/.cu input file (more than one are not allowed in this step) a dependency file that can be included in a make file.
--compile	-c	Compile each .c/.cc/.cpp/.cxx/.cu input file into an object file.
--link	-link	This option specifies the default behavior: compile and link all inputs.
--lib	-lib	Compile all input files into object files (if necessesary), 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. 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 nvcc.profile).

File and path specifications

--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 *include-file,…*	-include	Specify header files that must be preincluded during preprocessing or compilation.
--library *library-file,…*	-l	Specify libraries to be used in the linking stage. The libraries are searched for on the library search paths that have been specified using option '-L'.
--define-macro *macrodef,…*	-D	Specify macro definitions for use during preprocessing or compilation
--undefine-macro *macrodef,…*	-U	Undefine a macro definition
--include-path *include-path,…*	-I	Specify include search paths.
--system-include *include-path,…*	-isystem	Specify system include search paths.
--library-path *library-path,…*	-L	Specify library search paths.
--output-directory *directory*	-odir	Specify the directory of the output file. This option is intended for letting the dependency generation step (--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 host compiler executable (Microsoft Visual Studio cl, or a gcc derivative) resides. By default, this executable is expected in the current executable search path.

Options altering compiler/linker behavior

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

Options for passing specific phase options

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

--compiler-options *options,…*	-Xcompiler	Specify options directly to the compiler/preprocessor.
--linker-options *options,…*	-Xlinker	Specify options directly to the linker.

--cudafe-options	-Xcudafe	Specify options directly to cudafe.
--opencc-options *options,…*	-Xopencc	Specify options directly to nvopencc, typically for steering nvopencc optimization.
--ptxas-options *options,…*	-Xptxas	Specify options directly to the ptx optimizing assembler.

Options for guiding the compiler driver

--dry run	-dry run	Do not execute the compilation commands generated by nvcc. Instead, list them.
--v erbose	-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.
--sav e-temps	-sav e-temps	This option is an alias of –keep.
--dont-use-profile	-noprof	Don't use the nvcc.profile file to guide the compilation.
--clean-targets	-clean	This option reverses the behaviour 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 -R, to specify command line arguments for the executable.
--input-driv e-prefix *prefix*	-idp	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 'current' development environment represents absolute paths. Use '-idp /cygwin/' for CygWin build environments, and '-idp /' for Mingw.
--dependency-drive-prefix *prefix*	-ddp	On Windows platforms, when generating dependency files (option -M), all file names must be converted to whatever the used instance of 'make' will recognize. Some instances of 'make' have trouble with the colon in absolute paths in native Windows format, which depends on the environment in which this 'make' instance has been compiled. Use '-ddp /cygwin/' for a CygWin make, and '-ddp /' for Mingw. Or leave these file names in native Windows format by specifying nothing.
--driv e-prefix *prefix*	-dp	Specifies <prefix> as both input-drive-prefix and dependency-drive-prefix.

Options for steering CUDA compilation

--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	-ftz	The –ftz option controls single precision denormals support. When –ftz=false, denormals are supported and with -ftz=true, denormals are flushed to 0.

--prec-div	-prec-div	The –prec-div option controls single precision division.. With –prec-div=true, the division is IEEE compliant, with –prec-div=false, the division is approximate
--prec-sqrt	-prec-sqrt	The –prec-sqrt option controls single precision square root. With –prec-sqrt=true, the square root is IEEE compliant, with –prec-sqrt=false, the square root is approximate
--entries *entry,…*	-e	In case of compilation of ptx or gpu files to cubin: specify the global entry functions for which code must be generated. By default, code will be generated for all entries.
--fmad	-fmad	Enables (disables) the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA). The default is -fmad=true.

Options for steering GPU code generation

--gpu-architecture *gpuarch*	-arch	Specify the name of the NVIDIA GPU to compile for. This can either be a 'real' GPU, or a 'virtual' 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 . 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 option are assumed by the last, potentially runtime, compilation stage. Currently supported compilation architectures are: virtual architectures compute_10, compute_11, compute_12, compute_13 plus GPU architectures sm_10, sm_11, sm_12 and sm_13 that implement these.
--gpu-code *gpuarch,…*	-code	Specify the name of the NVIDIA GPU to generate code for. Unless option –export-dir is specified (see below), nvcc embeds a compiled code image in the executable for each specified 'code' architecture, which is a true binary load image for each 'real' architecture, and ptx code for each virtual architecture. During runtime, such embedded ptx code will be dynamically compiled by the CUDA runtime system if no binary load image is found for the 'current' GPU. Architectures specified for options –arch and –code may be virtual as well as real, but the 'code' architectures must be compatible with the 'arch' architecture. When the code option is used, the value for the –arch option must b a virtual ptx architecture. For instance, 'arch'=compute_13 is not compatible with 'code'=sm_10, because the earlier compilation stages will assume the availability of compute_13 features that are not present on sm_10. This option defaults to the value of option '-arch'. Currently supported GPU architectures: sm_10, sm_11, sm_12 and sm_13.

--generate-code	-gencode	This option provides a generalization of the '-arch=<arch> -code=code,...' option combination for specifying nvcc behavior with respect to code generation. Where use of the previous options generates different code for a fixed virtual architecture, option '--generate-code' allows multiple nvopencc invocations, iterating over different virtual architectures. In fact, -arch=<arch> -code=<code>,...' is equivalent to '--generate-code. arch=<arch>,code=<code>,...'. '--generate-code' options may be repeated for different virtual architectures. Allowed keywords for this option: 'arch','code'.
--export-dir *file*	-dir	Specify the name of a directory to which all device code images will be copied, intended as a device code repository that can be inspected by the CUDA driver at application runtime when it occurs in the appropriate device code search paths ('dir' should be in CUDA_DEVCODE_PATH). This repository can either be a directory, or a zip file. In either case, nvcc will maintain a directory structure in order to facilitate code lookup by the CUDA driver. When this option is specified with the name of a nonexisting file, then this file will be created as a directory.
--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. If this option is not specified, then no maximum is assumed. Otherwise the specified value will be rounded to the next multiple of 4 registers until the GPU specific maximum of 128 registers.

Generic tool options

--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.

Phase options

The following table lists some useful options to lower level compilation tools:

-Xcompiler, -Xlinker	See host compiler documentation	
-Xptxas	-v	Print code generation statistics.
-Xopencc	-LIST:source=on	Include source code in generated ptx

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.ii phase is listed. All other CUDA compilations are variants in some form of another of the .cu to C++ transformation.

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 csh each have slightly different notations for assigning values to environment variables.

The command list contains the following categories, which in the example below are alternately shown in normal print and boldface (see also sections 0 and 0):

1. Definition of standard variables _HERE_ and _SPACE_
2. Environment assignments resulting from executing nvcc.profile
3. Definition of Visual Studio installation macros (derived from –compiler-bindir)
4. Environment assignments resulting from executing vsvars32.bat
5. 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\binn\;c:\Program
Files\Perforce;C:\cygwin\lib\lapack
#$
PATH=c:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/win32_debu
g/open64/bin;c:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/wi
n32_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\X
11R6\bin;c:\WINDOWS\system32;c:\WINDOWS;c:\WINDOWS\System32\Wbem;c:\Progra
m Files\Microsoft SQL Server\90\Tools\binn\;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/in
clude"                    "-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=
#$ OPENCC_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\Microso
ft.NET\Framework\v2.0.50727;c:\MVS\VC\VCPackages;c:\sw\gpgpu\bin\win32_deb
ug/../../../compiler/gpgpu/export/win32_debug/open64/bin;c:\sw\gpgpu\bin\w
in32_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\binn\;c:\Program Files\Perforce;C:\cygwin\lib\lapack
#$
INCLUDE=c:\MVS\VC\ATLMFC\INCLUDE;c:\MVS\VC\INCLUDE;c:\MVS\VC\PlatformSDK\i
nclude;c:\MVS\SDK\v2.0\include;
#$
LIB=c:\MVS\VC\ATLMFC\LIB;c:\MVS\VC\LIB;c:\MVS\VC\PlatformSDK\lib;c:\MVS\SD
K\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:\W
INDOWS\Microsoft.NET\Framework\v2.0.50727;c:\MVS\VC\VCPackages;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:\WIND
OWS\system32;c:\WINDOWS;c:\WINDOWS\System32\Wbem;c:\Program
Files\Microsoft SQL Server\90\Tools\binn\;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/in
clude"                    "-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.cpp1.ii" "x.cu"
#$ cudafe "-
Ic:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/win32_debug/in
clude"                    "-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/inc"
"-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/tools/cudart"   -I. --
gen_c_file_name "x.cudafe1.c" --gen_device_file_name "x.cudafe1.gpu" --
include_file_name x.fatbin.c --no_exceptions -tused  "x.cpp1.ii"
#$ cudafe -E --c -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/in
clude"                    "-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  -o "x.cpp2.i" "x.cudafe1.gpu"
#$ cudafe --c "-
Ic:\sw\gpgpu\bin\win32_debug/../../../compiler/gpgpu/export/win32_debug/in
clude"                    "-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/inc"
"-Ic:\sw\gpgpu\bin\win32_debug/../../cuda/tools/cudart"   -I. --
gen_c_file_name "x.cudafe2.c" --gen_device_file_name "x.cudafe2.gpu" --
include_file_name x.fatbin.c  "x.cpp2.i"
#$ cudafe -E --c -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/in
clude"                    "-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__GNUC__ -
D__CUDABE__  -o "x.cpp3.i" "x.cudafe2.gpu"
#$ nvopencc  -Werror   "x.cpp3.i"  -o "x.ptx"
#$ ptxas  -arch=sm_10 "x.ptx"  -o "x.cubin"
#$ filehash --skip-cpp-directives -s "" "x.cpp3.i" > "x.cpp3.i.hash"
#$ fatbin --key="x@xxxxxxxxx" --source-name="x.cu" --usage-mode="" --
embedded-fatbin="x.fatbin.c" --image=profile=sm_10,file=x.cubin
#$ cudafe -E --c -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/in
clude"                    "-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"  -o "x.cu.c"
"x.cudafe1.c"
```

Full CUDA compilation trajectory

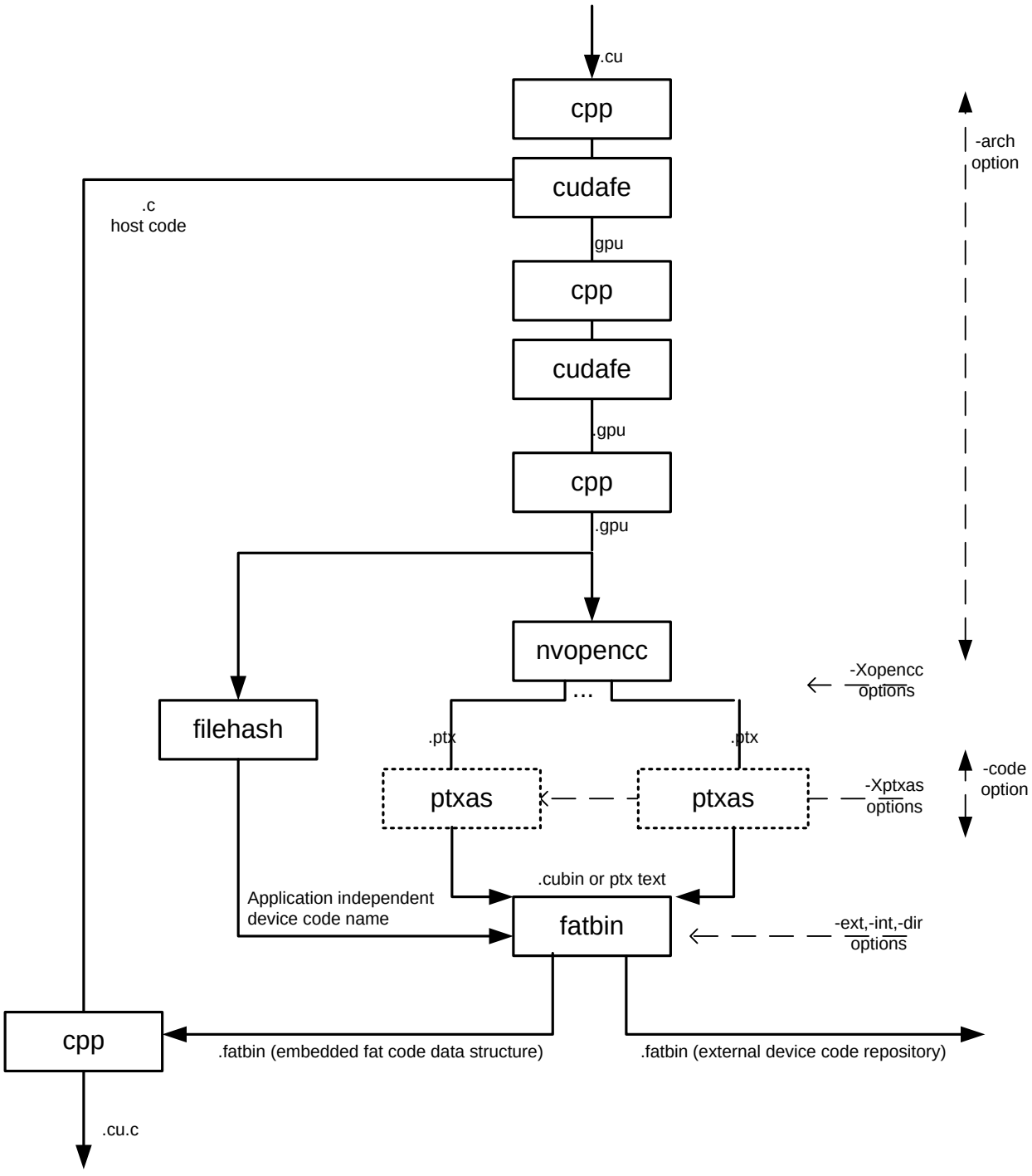


Figure 2: CUDA compilation from .cu to .cu.cpp.ii

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.

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.

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.

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.

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 $@


#
# Cleanup:
#
clean :
        $(RM) $(OBJECTS) *.dep
```

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.

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 Tesla GPU will very likely not run on a next generation graphics card (and vice versa). This is because the Tesla instruction set and instruction encodings is different from FERMI, which in turn will probably be substantially different from those of the next generation GPU.

V1 (Tesla)	***V2 (Next generation)***
- sm_10	*- sm_20*
- sm_11	*- .. ?? ..*
- sm_12	
- sm_13	
V3 (??)	***V4 (??)***
- .. ?? ..	*- .. ?? ..*

Because they share the basic instruction set, binary compatibility within one GPU generation, however, can under certain conditions 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.

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 $x_1y_1 <= x_2y_2$ then all non- ISA related capabilities of $sm\_x_1y_1$ are included in those of $sm\_x_2y_2$. 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.

sm_10	ISA_1 Basic features
sm_11	+ atomic memory operations on global memory
sm_12	+ atomic memory operations on shared memory + vote instructions
sm_13	+ double precision floating point support
sm_20	+ FERMI support

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.

Virtual architectures

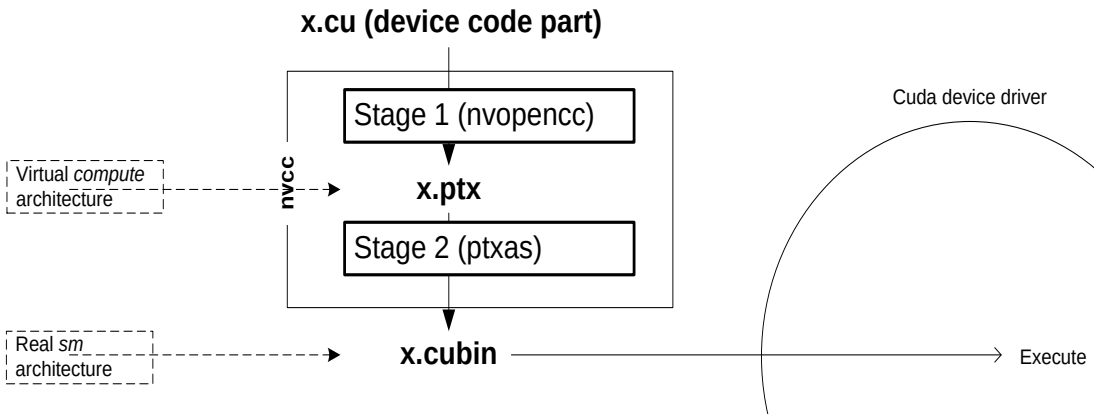
GPU compilation is performed via an intermediate representation, PTX ([…]), which can be considered as assembly for a virtual GPU architecture. Contrary to an actual graphics processor, such a virtual GPU is defined entirely by the set of

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 (someway 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.



Virtual architecture feature list

compute_10	Basic features
compute_11	+ atomic memory operations on global memory
compute_12	+ atomic memory operations on shared memory + vote instructions

compute_13	+ double precision floating point support
Compute_20	+ FERMI 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.

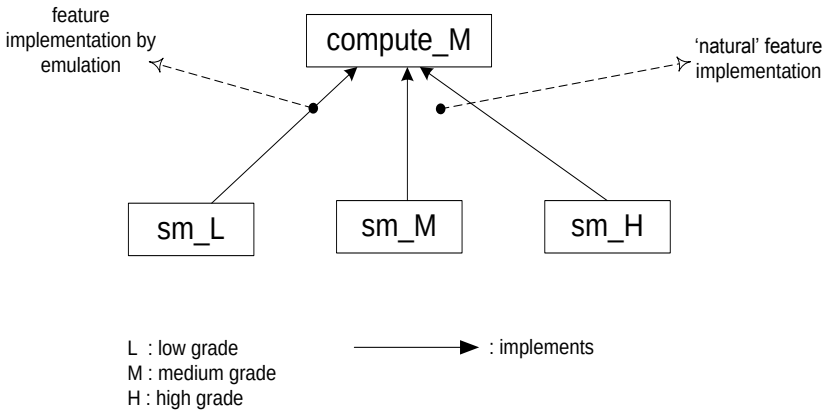
First, a next generation architecture might not provide any functional improvements, in which case the list of 'real' architectures will be extended (because we must be able to generate code for this architecture), but no new compute architecture is necessary.

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:

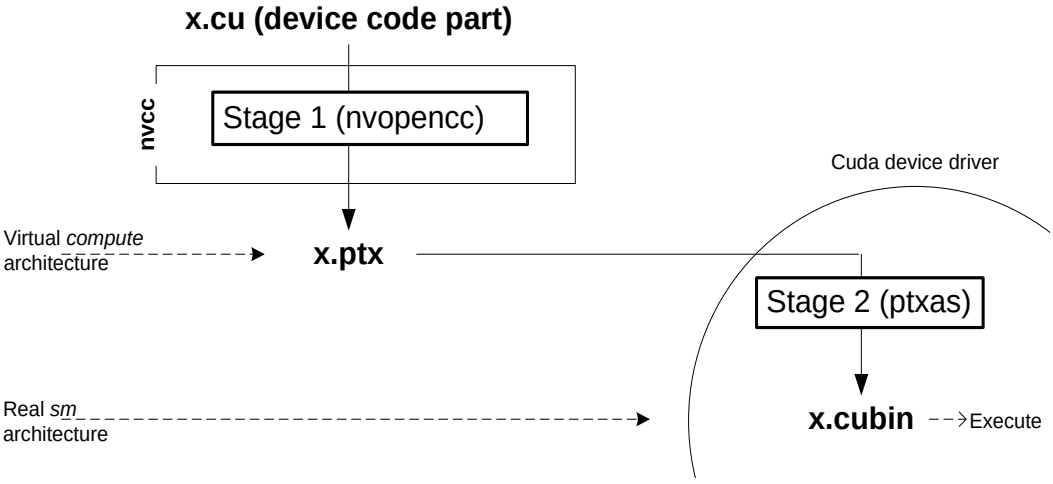


Further mechanisms

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 the CUDA SDK: just in time compilation (JIT) and device code repositories.

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 an *sm_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 (see next chapter) which is persistent over multiple runs of the applications.

Device code repositories

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 device code repositories, 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. Device code repositories are explained in more detail in the next chapter.

Nvcc examples

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 will perform 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
```

Nvcc allows a number of short hands for simple cases:

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
```

Shorthand 2

The *-arch* option can be omitted, in which case it defaults to the 'closest' virtual architecture that is implemented by the GPU that is specified with the *–code* option. In case multiple GPUs are specified as code architecture, this default virtual architecture must be unique over all these GPUs.

Example:

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

are short hands for

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

Shorthand 3

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
```

Shorthand 4

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
```

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_:

```
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\'
```

Virtual architecture identification macro

The architecture identification macro __CUDA_ARCH__ will be 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.

Device code repositories

With the existence of multiple NVIDIA GPU architectures, it is not always predictable at compile time on what type of GPU the application will run. The importance of this issue is directly proportional to the number of different GPUs.

Nvcc, together with the CUDA runtime system, provides the following mechanisms for dealing with this:

1. Storing more than one generated code instance embedded in the executable.
2. Allowing ptx intermediate representations as generated code
3. Maintaining device code repositories external to the executable, in directory trees, or in zip files.
4. More than one compiled code instance for the same device code occurring in the CUDA source allows the CUDA runtime system to select an instance that is compatible with the current GPU, which is the GPU on which the runtime system is about to launch the the code. If more than one compatible code instances are found, then the runtime system can select the 'most appropriate', and in case the most appropriate code instance is still ptx intermediate code, the runtime system may decide to compile it for the current GPU. Ptx intermediate code is especially useful for distributed libraries, such as cublas.
5. External code repositories allow finetuning as more of the compilation environment becomes known: because such repositories are directory trees in an open format (normal directory or zip format), any ptx code that it contains can be 'hand- compiled' after distribution. One particular way of such finetuning is to use runtime compilation while enabling a device code translation cache: this will result in a new code repository, or it will extend an existing one.

External device code structure

The structure of device code repositories will be automatically created and maintained by nvcc, during static compilation, and by the CUDA runtime system whenever it is storing compiled ptx code into a translation cache.

Generating code in an external repository

External code repositories can be populated using nvcc option –dir. By using this options, either all device code can be embedded in the produced object files and executables, or all device code can be stored in an external repository.

For example, consider the following nvcc compilation commands:

1. nvcc acos.cu –o acos.out
2. nvcc acos.cu –o a.out –arch compute_10
3. nvcc acos.cu –o a.out –arch compute_10 –code compute_10,sm_10,sm_13
4. nvcc acos.cu –o a.out –arch compute_10 –code compute_10,sm_10,sm_13 -dir=a.out.devcode

These commands have the following effects:

1. Generate sm_10 code binary code embedded in the executable (default value for option –arch)
2. Generate compute_10 intermediate ptx code, embedded in the executable. Unless the CUDA driver finds matching binary code at runtime in a code repository file, this code will be compiled at application startup.
3. Generate a mix of compiled code alternatives for the CUDA driver to choose from, still embedded in the executable.
4. Generate the same mix of code alternatives, but this time store all of the generated code in the external repository file, called *a.out.devcode*. No device code is embedded in the executable itself. When starting executable *a.out*, the CUDA driver will automatically search for a device code repository in the environment variable CUDA_DEVCODE_PATH. So if *a.out.devcode* is in CUDA_DEVCODE_PATH, then the device code will be found.

Using code repositories by the CUDA runtime system

While running an executable E, the CUDA runtime system can be instructed to search device code repositories, in the following ways:

By defining environment variable CUDA_DEVCODE_PATH to a colon- (Linux), or semicolon- (Windows) separated list of repository file names. For each name R in this list, the CUDA runtime will search in R for the device code.

By defining the device code translation cache (see next).

Enabling the device code translation cache

By default, the result of any runtime compiled ptx code will be used for the lifetime of the process that compiles it, and then discarded. Runtime compilation is intended

to be an escape situation, but in case it occurs, it might be desirable to keep the result for later invocations of the executable.

This can be achieved by defining the environment variable CUDA_DEVCODE_CACHE to the name of a selected code repository. When defined, the CUDA runtime system will add the result of runtime compiled code to this repository, after creating it as a directory when it did not exist before.

Additionally, CUDA_DEVCODE_CACHE will be placed on the repository search list.

Miscellaneous nvcc usage

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.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com