The CUDA Compiler Driver NVCC

Last modified on: 11/5/2007

Document Change History

Version	Date	Responsible	Reason for Change
beta	01-15-2007	Juul VanderSpek	Initial release
0.1	05-25-2007	Juul VanderSpek	CUDA 0.1 release
1.0	06-13-2007	Juul VanderSpek	CUDA 1.0 release
1.1	10-12-2007	Juul VanderSpek	CUDA 1.1 release

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 an NVIDIA GPU as coprocessor 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, ref […].

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 proprietory NVIDIA compilers/assemblers, compiles the host code using any 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 'fat device code binaries'). 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 general purpose compiler drivers (such as gcc), in that

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 C compiler that is available on the current platform, and in case 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 application build and make scripts when these must be portable across Linux and Windows platforms.

```
/* ---------------------------- target code -------------------------------*/

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

__global__ void acos_main (struct acosParams parms)
{
    int i;
    for (i = threadIdx.x; i < parms.n; i += ACOS_THREAD_CNT) {
        parms.res[i] = acosf(parms.arg[i]);
    }
}

/* ---------------------------- host code -------------------------------*/

int main (int argc, char *argv[])
{
    cudaError_t         cudaStat;
    float*              acosRes = 0;
    float*              acosArg = 0;
    float*              arg = malloc(N*sizeof(arg[0]));
    float*              res = malloc(N*sizeof(res[0]));
    struct acosParams   funcParams;

    ... fill arguments array 'arg' ....

    cudaStat = cudaMalloc ((void **)&acosArg, N * sizeof(acosArg[0]));

    cudaStat = cudaMemcpy (acosArg, arg, N * sizeof(arg[0]),
                           cudaMemcpyHostToDevice);

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

    acos_main<<<1,ACOS_THREAD_CNT>>>(funcParams);

    cudaStat = cudaMemcpy (res, acosRes, N * sizeof(res[0]),
                           cudaMemcpyDeviceToHost);

    ... process result array 'res' ....
}
```

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.

0 provides a full explanation of the nvcc command line options. 0 will explain more on the the different input and intermediate file types. The following paragraphs will list the recognized file name suffixes and the supported compilation phases.

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 source file	-cuda	".c" appended to source file name, as in *x.cu.c*
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"
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 as specified in nvcc.profile (see Section 0) 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	Value of environment variable TEMP, or */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 in Section 0.

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 0).

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-bin-dir* 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, if cl is in the current executable search path, then nvcc assumes that vsvars.bat has already been executed, and hence that the relevant environment variables already have the necessary values. Otherwise, nvcc will locate *vsvars.bat* via the specified *compiler-bin-dir* 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 couse '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.

The following table lists the available options:

--cuda	-cuda	Compile all .cu input files to .cu.c 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.
--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
--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 Studion 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 debuggable 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. Currently only to be used when compiling with –cubin on linux64 platform.

Options for passing specific phase options

These allow for passing options directly to the intended compilation phase. Using these, users have the ability to pass options to the lower level compilation tools without the need for nvcc to know about each and every such option.

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

--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

--dryrun	-dryrun	Do not execute the compilation commands generated by nvcc. Instead, list them.
--verbose	-v	List the compilation commands generated by this compiler driver, but do not suppress their execution.
--keep	-keep	Keep all intermediate files that are generated during internal compilation steps.
--save-temps	-save-temps	This option is an alias of --keep.
--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-drive-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.
--drive-prefix *prefix*	-dp	Specifies <prefix> as both input-drive-prefix and dependency-drive-prefix.

Options for steering CUDA compilation

--device-emulation	-deviceemu	Generate code for the GPGPU emulation library.
--use_fast_math	-use_fast_math	Make use of fast math library.
--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.
--no-cpp-cudafe	-ncfe	On Windows, preprocessing steps are performed using cudafe whent this tool can be found. The Microsoft compiler (cl) is avoided

		because this compiler insists on echoeing the name of its input file, which makes CUDA compilation and dependency generation quite noisy. Specifying this option will cause cl to be used instead.

Options for steering GPU code generation

--gpu-name *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, plus GPU architectures sm_10 and sm_11 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 will embed 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. For instance, 'arch'=compute_11 is not compatible with 'code'=sm_10, because the earlier compilation stages will assume the availability of compute_11 features that are not present on sm_10. This option defaults to the value of option '-arch'. Currently supported GPU architectures: sm_10 and sm_11.
--export-dir *file*	-dir	Specify the name of a file to which all 'external' code imates will be copied (see below), 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. This file 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 not used, all 'external' images will be silently discarded. When this option is specified with the name of a nonexisting file, then this file will be created as

		a directory.
--extern-mode *all|none|real|virtual*	-ext	Specifie which of the 'code' images will be copied into the directory specified with option 'export-dir'. If this option is not specified, then the behavior is as follows: if option 'intern-mode' is not specified, then all code images that are not defined as intern will be considered extern. Otherwise, if neither of these options are specified, then all code images will be considered intern.
--intern- mode *all|none|real|virtual*	-int	Specifie which of the 'code' images will be embedded in the resulting object file/executable. If this option is not specified, then the behavior is as follows: if option 'extern-mode' is not specified, then all code images that are not defined as extern will be considered intern. Otherwise, if neither of these options are specified, then all code images will be considered intern.
--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.

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 .c to .cu.c 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_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_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_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@xxxxxxxxxx" --source-name="x.cu" --usage-mode="" --
embedded-fatbin="x.fatbin.c" --image=profile=sm_10,file=x.cubin
#$ cudafe -E --c -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

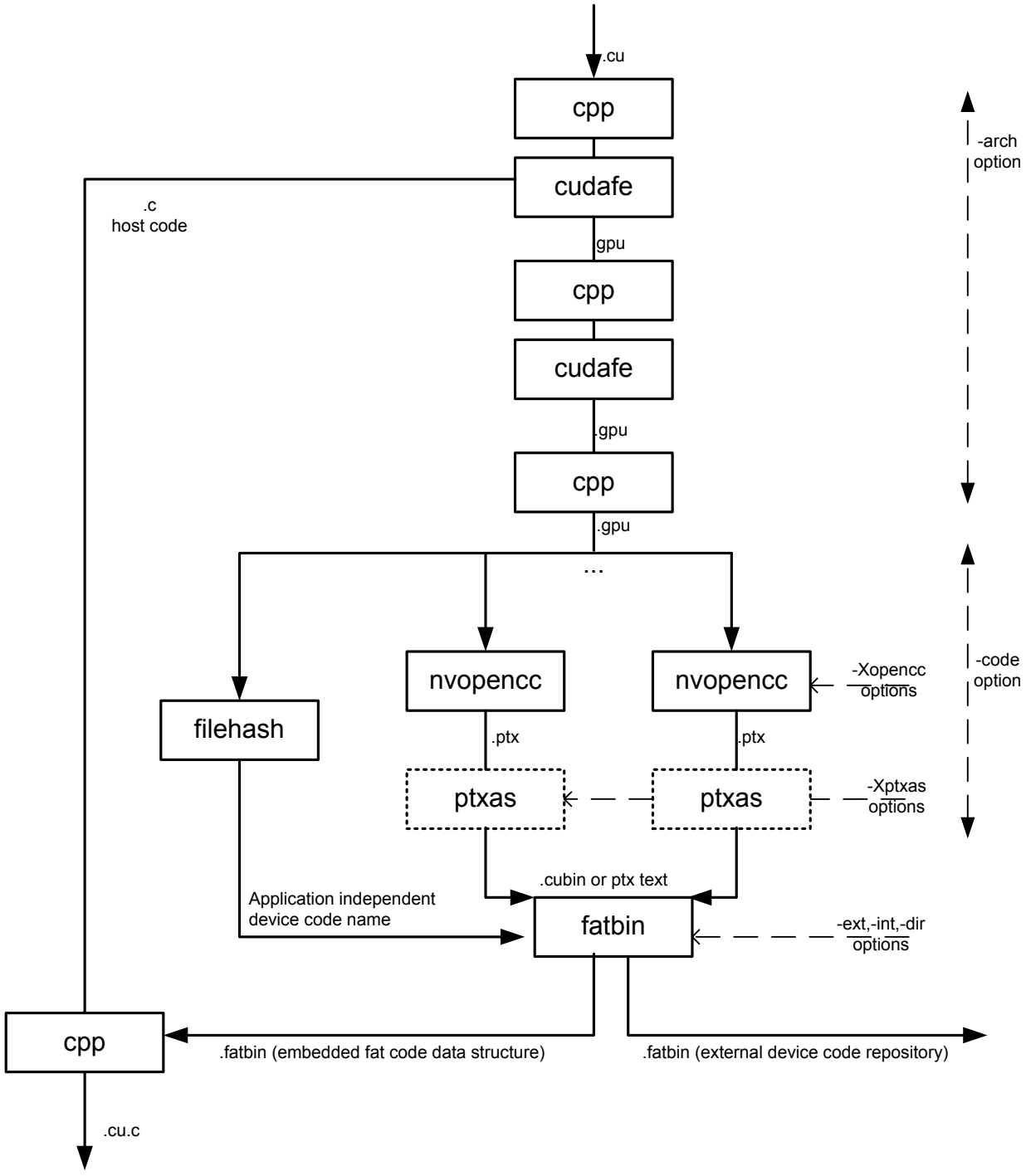


Figure 2: CUDA compilation from .cu to .cu.c

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.

Using cudafe for preprocessing

Figure 2 shows that a full CUDA compilation step requires 4 preprocessing steps, which are ultimately performed using the platform compiler. An unfortunate side effect of this on Windows platforms would be a quite noisy CUDA compilation, due to the fact that *cl* insists on echoing the name of its input file each time it is invoked. For this reason, nvcc will use cudafe for preprocessing whenever it finds this internal CUDA tool on the the executable search PATH (which normally is the case in CUDA releases).

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
```

Device code repositories

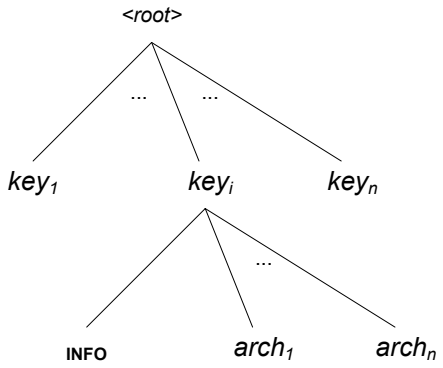
With the existence of multiple NVIDIA GPU architectures, it is not always predictable at compile time on what type of GPU the application will run. The importance of this issue is directly proportional to the number of different GPUs.

Nvcc, together with the CUDA runtime system, provides three 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.



These repositories contain subdirectories that each correspond with the device code that occurred in a single .cu source file, compiled for a specific architecture (nvcc option –*arch*). Because the compilation trajectory up to ptx generation is affected by the value of option –arch, compiling for a different architecture might result in a different subdirectory in the code repository. The name of each subdirectory is generated by nvcc, and included in the object file for the corresponding CUDA source file so that the CUDA runtime system is able to unambiguously select all different compiled code instances that belong to that source file.

Each of these code directories contains in separate files the code that has been generated for the different GPU architectures: these are binary load images (in textual format) for real architectures, and ptx intermediate code for 'virtual' architectures. Each code file is named according to the GPU architecture to which it corresponds (see also the examples later on in this chapter).

Each directory also contains an INFO file (by that name), containing the following information: name of the source file as it was specified to nvcc, and worst case usage information specified for the device code. Worst case usage information currently is information on the maximum thread block size, encoded by the Xptxas options that were specified to nvcc during static compilation. When 'manually' compiling the ptx code instances using ptxas, the options specified in the INFO file must be passed to the ptxas invocation.

Generating code in an external repository

External code repositories can be populated using nvcc options –dir, -ext, and –int. By using these 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, or both, or any combination in between.

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

4. nvcc acos.cu –o a.out –arch compute_10 –code compute_10,sm_10 -dir=a.out.devcode -ext=virtual
5. nvcc acos.cu –o a.out –arch compute_10 –code compute_10,sm_10 -dir=a.out.devcode -ext=all
6. nvcc acos.cu –o a.out –arch compute_10 –code compute_10,sm_10 -dir=a.out.devcode -ext=all –int=all

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 keep the 'real' chip binaries embedded, but store the ptx code in an external repository file, called *a.out.devcode*. When starting executables *E*, the CUDA driver will automatically search for a device code repository *E.devcode* in the same directory. Having the ptx code external allows later compilation for unanticipated 'real' GPUs without having to recompile or even relink a.out itself.
5. Generate the same mix of code alternatives, but this time store all of the generated code in the external repository file. No device code is embedded in the executable itself.
6. Generate the same mix of code alternatives, but this time store all of the generated code in the external repository file, *as well as keep it embedded in the executable*. This might not be the most sensible situation, but it is allowed.

In a sample situation, commands 5 and 6 will produce the following repository structure, with the contents of the INFO file as shown below:

```
a.out.devcode
a.out.devcode/acos@2603d9b33c621ef5
a.out.devcode/acos@2603d9b33c621ef5/INFO
a.out.devcode/acos@2603d9b33c621ef5/sm_10
a.out.devcode/acos@2603d9b33c621ef5/compute_10
```

INFO=″

```
IDENT=acos.cu
USAGE_MODE=
″
```

Repository formats

By default, nvcc will *create* code repositories as directory trees. However, if the tree exists a as a file in the Unix *tar* format, nvcc will *extend* it as a tar file. For instance, supposed that a.out.devcode does not initially exist, the first situation will end up producing a code repository as a normal directory, while the second situation will end up with the repository in a tar file:

7. nvcc acos.cu –ext=all –dir=a.out.devcode
8. tar cvf a.out.devcode dummy /dev/null
   nvcc acos.cu –ext=all –dir=a.out.devcode

Similar holds for the CUDA runtime system while creating/extending a translation cache.

Note: on Linux platforms, also gzipped tar formats are supported

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:

Implicitly, by the existence of code repository files E.devcode, or E.devcode.tar, or E.devcode.tar.gz (Linux only). All of these files may exist, in which case E.devcode will take precedence.

Explicitly, 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 for R as well as R.zip with similar precedence and format remarks as for the executable repository.

Explicitly, by defininining 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.

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 or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use 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 United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2006 NVIDIA Corporation. All rights reserved.

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