NVRTC - CUDA Runtime Compilation

User Guide

DU-07529-001 _v11.8 | October 2022

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

NVRTC is a runtime compilation library for CUDA C++. It accepts CUDA C++ source code in character string form and creates handles that can be used to obtain the PTX. The PTX string generated by NVRTC can be loaded by cuModuleLoadData and cuModuleLoadDataEx, and linked with other modules by cuLinkAddData of the CUDA Driver API. This facility can often provide optimizations and performance not possible in a purely offline static compilation.

In the absence of NVRTC (or any runtime compilation support in CUDA), users needed to spawn a separate process to execute nvcc at runtime if they wished to implement runtime compilation in their applications or libraries, and, unfortunately, this approach has the following drawbacks:

- The compilation overhead tends to be higher than necessary, and
- End users are required to install nvcc and related tools which make it complicated to distribute applications that use runtime compilation.

NVRTC addresses these issues by providing a library interface that eliminates overhead associated with spawning separate processes, disk I/O, etc., while keeping application deployment simple.

Chapter 2. Getting Started

2.1. System Requirements

NVRTC requires the following system configuration:

- Operating System: Linux x86_64, Linux ppc64le, Linux aarch64 or Windows x86_64.
- GPU: Any GPU with CUDA Compute Capability 2.0 or higher.
- CUDA Toolkit and Driver.

2.2. Installation

NVRTC is part of the CUDA Toolkit release and the components are organized as follows in the CUDA toolkit installation directory:

- On Windows:
  - `include\nvrtc.h`
  - `bin\nvrtc64_Major Release VersionMinor Release Version_0.dll`
  - `bin\nvrtc-builtins64_Major Release VersionMinor Release Version.dll`
  - `lib\x64\nvrtc.lib`
  - `lib\x64\nvrtc_static.lib`
  - `lib\x64\nvrtc-builtins_static.lib`
  - `doc\pdf\NVRTC_User_Guide.pdf`
- On Linux:
  - `include/nvrtc.h`
  - `lib64/libnvrtc.so`
  - `lib64/libnvrtc.so.Major Release Version.Minor Release Version`

- `lib64/libnvrtc.so.Major Release Version.Minor Release Version.<build version>`
- `lib64/libnvrtc-builtins.so`
- `lib64/libnvrtc-builtins.so.Major Release Version.Minor Release Version`
- `lib64/libnvrtc-builtins.so.Major Release Version.Minor Release Version.<build version>`
- `lib64/libnvrtc_static.a`
- `lib64/libnvrtc-builtins_static.a`
- `doc/pdf/NVRTC_User_Guide.pdf`

Chapter 3. User Interface

This chapter presents the API of NVRTC. Basic usage of the API is explained in Basic Usage.

- Error Handling
- General Information Query
- Compilation
- Supported Compile Options
- Host Helper

3.1. Error Handling

NVRTC defines the following enumeration type and function for API call error handling.

enum nvrtcResult

The enumerated type nvrtcResult defines API call result codes. NVRTC API functions return nvrtcResult to indicate the call result.

Values

NVRTC_SUCCESS = 0
NVRTC_ERROR_OUT_OF_MEMORY = 1
NVRTC_ERROR_PROGRAM_CREATION_FAILURE = 2
NVRTC_ERROR_INVALID_INPUT = 3
NVRTC_ERROR_INVALID_PROGRAM = 4
NVRTC_ERROR_INVALID_OPTION = 5
NVRTC_ERROR_COMPILATION = 6
NVRTC_ERROR_BUILTIN_OPERATION_FAILURE = 7
NVRTC_ERROR_NO_NAME_EXPRESSIONS_AFTER_COMPILATION = 8
NVRTC_ERROR_NO_LOWERED_NAMES_BEFORE_COMPILATION = 9
NVRTC_ERROR_NAME_EXPRESSION_NOT_VALID = 10
NVRTC_ERROR_INTERNAL_ERROR = 11

const char *nvrtcGetErrorString (nvrtcResult result)

nvrtcGetErrorString is a helper function that returns a string describing the given nvrtcResult code, e.g., NVRTC_SUCCESS to `"NVRTC_SUCCESS"`. For unrecognized enumeration values, it returns `"NVRTC_ERROR unknown"`.

Parameters

result

CUDA Runtime Compilation API result code.

Returns

Message string for the given nvrtcResult code.

3.2. General Information Query

NVRTC defines the following function for general information query.

nvrtcResult nvrtcGetNumSupportedArchs (int *numArchs)

nvrtcGetNumSupportedArchs sets the output parameter `numArchs` with the number of architectures supported by NVRTC. This can then be used to pass an array to nvrtcGetSupportedArchs to get the supported architectures.

Parameters

numArchs

number of supported architectures.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT

Description

see nvrtcGetSupportedArchs

nvrtcResult nvrtcGetSupportedArchs (int *supportedArchs)

nvrtcGetSupportedArchs populates the array passed via the output parameter `supportedArchs` with the architectures supported by NVRTC. The array is sorted in the ascending order. The size of the array to be passed can be determined using nvrtcGetNumSupportedArchs.

Parameters

supportedArchs
sorted array of supported architectures.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT

Description

see nvrtcGetNumSupportedArchs

nvrtcResult nvrtcVersion (int *major, int *minor)

nvrtcVersion sets the output parameters `major` and `minor` with the CUDA Runtime Compilation version number.

Parameters

major
CUDA Runtime Compilation major version number.
minor
CUDA Runtime Compilation minor version number.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT

3.3. Compilation

NVRTC defines the following type and functions for actual compilation.

typedef _nvrtcProgram *nvrtcProgram

nvrtcProgram is the unit of compilation, and an opaque handle for a program.

To compile a CUDA program string, an instance of nvrtcProgram must be created first with nvrtcCreateProgram, then compiled with nvrtcCompileProgram.

nvrtcResult nvrtcAddNameExpression (nvrtcProgram prog, const char *name_expression)

nvrtcAddNameExpression notes the given name expression denoting the address of a __global__ function or __device__/__constant__ variable.

Parameters

prog
CUDA Runtime Compilation program.

name_expression
constant expression denoting the address of a __global__ function or __device__/__constant__ variable.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_NO_NAME_EXPRESSIONS_AFTER_COMPILATION

Description

The identical name expression string must be provided on a subsequent call to nvrtcGetLoweredName to extract the lowered name.

See also:

nvrtcGetLoweredName

nvrtcResult nvrtcCompileProgram (nvrtcProgram prog, int numOptions, const char **options)

nvrtcCompileProgram compiles the given program.

Parameters

prog
CUDA Runtime Compilation program.

numOptions
Number of compiler options passed.

options
Compiler options in the form of C string array. `options` can be `NULL` when `numOptions` is 0.

Returns

- NVRTC_SUCCESS

- NVRTC_ERROR_OUT_OF_MEMORY
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM
- NVRTC_ERROR_INVALID_OPTION
- NVRTC_ERROR_COMPILATION
- NVRTC_ERROR_BUILTIN_OPERATION_FAILURE

Description

It supports compile options listed in Supported Compile Options.

nvrtcResult nvrtcCreateProgram (nvrtcProgram *prog, const char *src, const char *name, int numHeaders, const char **headers, const char **includeNames)

nvrtcCreateProgram creates an instance of nvrtcProgram with the given input parameters, and sets the output parameter `prog` with it.

Parameters

prog
CUDA Runtime Compilation program.

src
CUDA program source.

name
CUDA program name. `name` can be `NULL`; `"default_program"` is used when `name` is `NULL` or "".

numHeaders
Number of headers used. `numHeaders` must be greater than or equal to 0.

headers
Sources of the headers. `headers` can be `NULL` when `numHeaders` is 0.

includeNames
Name of each header by which they can be included in the CUDA program source. `includeNames` can be `NULL` when `numHeaders` is 0.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_OUT_OF_MEMORY
- NVRTC_ERROR_PROGRAM_CREATION_FAILURE

- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcDestroyProgram

nvrtcResult nvrtcDestroyProgram (nvrtcProgram *prog)

nvrtcDestroyProgram destroys the given program.

Parameters

prog
CUDA Runtime Compilation program.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcCreateProgram

nvrtcResult nvrtcGetCUBIN (nvrtcProgram prog, char *cubin)

nvrtcGetCUBIN stores the cubin generated by the previous compilation of `prog` in the memory pointed by `cubin`. No cubin is available if the value specified to `-arch` is a virtual architecture instead of an actual architecture.

Parameters

prog
CUDA Runtime Compilation program.

cubin
Compiled and assembled result.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetCUBINSize

nvrtcResult nvrtcGetCUBINSize (nvrtcProgram prog, size_t *cubinSizeRet)

nvrtcGetCUBINSize sets `cubinSizeRet` with the size of the cubin generated by the previous compilation of `prog`. The value of cubinSizeRet is set to 0 if the value specified to `-arch` is a virtual architecture instead of an actual architecture.

Parameters

prog
CUDA Runtime Compilation program.

cubinSizeRet
Size of the generated cubin.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetCUBIN

nvrtcResult nvrtcGetLoweredName (nvrtcProgram prog, const char *name_expression, const char **lowered_name)

nvrtcGetLoweredName extracts the lowered (mangled) name for a __global__ function or __device__/__constant__ variable, and updates *lowered_name to point to it. The memory containing the name is released when the NVRTC program is destroyed by nvrtcDestroyProgram. The identical name expression must have been previously provided to nvrtcAddNameExpression.

Parameters

prog
CUDA Runtime Compilation program.

name_expression
constant expression denoting the address of a __global__ function or __device__/ __constant__ variable.

lowered_name
initialized by the function to point to a C string containing the lowered (mangled) name corresponding to the provided name expression.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_NO_LOWERED_NAMES_BEFORE_COMPILATION
- NVRTC_ERROR_NAME_EXPRESSION_NOT_VALID

Description

See also:

nvrtcAddNameExpression

nvrtcResult nvrtcGetNVVM (nvrtcProgram prog, char *nvvm)

nvrtcGetNVVM stores the NVVM generated by the previous compilation of `prog` in the memory pointed by `nvvm`. The program must have been compiled with -dlto, otherwise will return an error.

Parameters

prog
CUDA Runtime Compilation program.

nvvm

Compiled result.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetNVVMSize

nvrtcResult nvrtcGetNVVMSize (nvrtcProgram prog, size_t *nvvmSizeRet)

nvrtcGetNVVMSize sets `nvvmSizeRet` with the size of the NVVM generated by the previous compilation of `prog`. The value of nvvmSizeRet is set to 0 if the program was not compiled with `-dlto`.

Parameters

prog

CUDA Runtime Compilation program.

nvvmSizeRet

Size of the generated NVVM.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetNVVM

nvrtcResult nvrtcGetProgramLog (nvrtcProgram prog, char *log)

nvrtcGetProgramLog stores the log generated by the previous compilation of `prog` in the memory pointed by `log`.

Parameters

prog
CUDA Runtime Compilation program.

log
Compilation log.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetProgramLogSize

nvrtcResult nvrtcGetProgramLogSize (nvrtcProgram prog, size_t *logSizeRet)

nvrtcGetProgramLogSize sets `logSizeRet` with the size of the log generated by the previous compilation of `prog` (including the trailing `NULL`).

Parameters

prog
CUDA Runtime Compilation program.

logSizeRet
Size of the compilation log (including the trailing `NULL`).

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

Note that compilation log may be generated with warnings and informative messages, even when the compilation of `prog` succeeds.

See also:

nvrtcGetProgramLog

nvrtcResult nvrtcGetPTX (nvrtcProgram prog, char *ptx)

nvrtcGetPTX stores the PTX generated by the previous compilation of `prog` in the memory pointed by `ptx`.

Parameters

prog
CUDA Runtime Compilation program.

ptx
Compiled result.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetPTXSize

nvrtcResult nvrtcGetPTXSize (nvrtcProgram prog, size_t *ptxSizeRet)

nvrtcGetPTXSize sets `ptxSizeRet` with the size of the PTX generated by the previous compilation of `prog` (including the trailing `NULL`).

Parameters

prog
CUDA Runtime Compilation program.

ptxSizeRet

Size of the generated PTX (including the trailing `NULL`).

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetPTX

3.4. Supported Compile Options

NVRTC supports the compile options below. Option names with two preceding dashs (`--`) are long option names and option names with one preceding dash (`-`) are short option names. Short option names can be used instead of long option names. When a compile option takes an argument, an assignment operator (=) is used to separate the compile option argument from the compile option name, e.g., `"--gpu-architecture=compute_60"`. Alternatively, the compile option name and the argument can be specified in separate strings without an assignment operator, .e.g, `"--gpu-architecture" "compute_60"`. Single-character short option names, such as `-D`, `-U`, and `-I`, do not require an assignment operator, and the compile option name and the argument can be present in the same string with or without spaces between them. For instance, `"-D=<def>"`, `"-D<def>"`, and `"-D <def>"` are all supported.

The valid compiler options are:

- Compilation targets
  - `--gpu-architecture=<arch>` (`-arch`)

    Specify the name of the class of GPU architectures for which the input must be compiled.

    - Valid `<arch>`s:
      - `compute_35`
      - `compute_37`
      - `compute_50`
      - `compute_52`

        - compute_53
        - compute_60
        - compute_61
        - compute_62
        - compute_70
        - compute_72
        - compute_75
        - compute_80
        - sm_35
        - sm_37
        - sm_50
        - sm_52
        - sm_53
        - sm_60
        - sm_61
        - sm_62
        - sm_70
        - sm_72
        - sm_75
        - sm_80
    - Default: compute_52
- Separate compilation / whole-program compilation
    - --device-c (-dc)

      Generate relocatable code that can be linked with other relocatable device code. It is equivalent to --relocatable-device-code=true.
    - --device-w (-dw)

      Generate non-relocatable code. It is equivalent to --relocatable-device-code=false.
    - --relocatable-device-code={true|false} (-rdc)

      Enable (disable) the generation of relocatable device code.

    - Default: `false`
  - `--extensible-whole-program` (`-ewp`)

    Do extensible whole program compilation of device code.

    - Default: `false`
- Debugging support
  - `--device-debug` (`-G`)

    Generate debug information. If --dopt is not specified, then turns off all optimizations.
  - `--generate-line-info` (`-lineinfo`)

    Generate line-number information.
- Code generation
  - `--dopt` on (`-dopt`)
  - `--dopt=on`

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

    Specify options directly to ptxas, the PTX optimizing assembler.
  - `--maxrregcount=<N>` (`-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. Value less than the minimum registers required by ABI will be bumped up by the compiler to ABI minimum limit.
  - `--ftz={true|false}` (`-ftz`)

    When performing single-precision floating-point operations, flush denormal values to zero or preserve denormal values. `--use_fast_math` implies `--ftz=true`.

    - Default: `false`
  - `--prec-sqrt={true|false}` (`-prec-sqrt`)

For single-precision floating-point square root, use IEEE round-to-nearest mode or use a faster approximation. `--use_fast_math` implies `--prec-sqrt=false`.

- Default: `true`

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

  For single-precision floating-point division and reciprocals, use IEEE round-to-nearest mode or use a faster approximation. `--use_fast_math` implies `--prec-div=false`.

  - Default: `true`

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

  Enables (disables) the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA). `--use_fast_math` implies `--fmad=true`.

  - Default: `true`

- `--use_fast_math` (`-use_fast_math`)

  Make use of fast math operations. `--use_fast_math` implies `--ftz=true --prec-div=false --prec-sqrt=false --fmad=true`.

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

  Enables more aggressive device code vectorization in the NVVM optimizer.

- `--modify-stack-limit={true|false}` (`-modify-stack-limit`)

  On Linux, during compilation, use `setrlimit()` to increase stack size to maximum allowed. The limit is reset to the previous value at the end of compilation. Note: `setrlimit()` changes the value for the entire process.

  - Default: `true`

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

  Generate intermediate code for later link-time optimization. It implies `-rdc=true`. Note: when this is used the nvrtcGetNVVM API should be used, as PTX or Cubin will not be generated.

- Preprocessing

  - `--define-macro=<def>` (`-D`)

    `<def>` can be either `<name>` or `<name=definitions>`.

    - `<name>`

      Predefine `<name>` as a macro with definition `1`.

    - `<name>=<definition>`

The contents of `<definition>` are tokenized and preprocessed as if they appeared during translation phase three in a `#define` directive. In particular, the definition will be truncated by embedded new line characters.

- `--undefine-macro=<def>` (`-U`)

  Cancel any previous definition of `<def>`.

- `--include-path=<dir>` (`-I`)

  Add the directory `<dir>` to the list of directories to be searched for headers. These paths are searched after the list of headers given to nvrtcCreateProgram.

- `--pre-include=<header>` (`-include`)

  Preinclude `<header>` during preprocessing.

- `--no-source-include` (`-no-source-include`) The preprocessor by default adds the directory of each input sources to the include path. This option disables this feature and only considers the path specified explicitly.

- Language Dialect
  - `--std={c++03|c++11|c++14|c++17|c++20}` (`-std={c++11|c++14|c++17|c++20}`)

    Set language dialect to C++03, C++11, C++14, C++17 or C++20

  - `--builtin-move-forward={true|false}` (`-builtin-move-forward`)

    Provide builtin definitions of `std::move` and `std::forward`, when C++11 language dialect is selected.

    - Default: `true`

  - `--builtin-initializer-list={true|false}` (`-builtin-initializer-list`)

    Provide builtin definitions of `std::initializer_list` class and member functions when C++11 language dialect is selected.

    - Default: `true`

- Misc.
  - `--disable-warnings` (`-w`)

    Inhibit all warning messages.

  - `--restrict` (`-restrict`)

    Programmer assertion that all kernel pointer parameters are restrict pointers.

  - `--device-as-default-execution-space` (`-default-device`)

    Treat entities with no execution space annotation as `__device__` entities.

- `--device-int128` (`-device-int128`)

  Allow the `__int128` type in device code. Also causes the macro `__CUDACC_RTC_INT128__` to be defined.

- `--optimization-info=<kind>` (`-opt-info`)

  Provide optimization reports for the specified kind of optimization. The following kind tags are supported:

  - `inline` : emit a remark when a function is inlined.

- `--version-ident={true|false}` (`-dQ`)

  Embed used compiler's version info into generated PTX/CUBIN

  - Default: `false`

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

  Display diagnostic number for warning messages. (Default)

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

  Disables the display of a diagnostic number for warning messages.

- `--diag-error=<error-number>`,... (`-diag-error`)

  Emit error for specified diagnostic message number(s). Message numbers can be separated by comma.

- `--diag-suppress=<error-number>`,... (`-diag-suppress`)

  Suppress specified diagnostic message number(s). Message numbers can be separated by comma.

- `--diag-warn=<error-number>`,... (`-diag-warn`)

  Emit warning for specified diagnostic message number(s). Message numbers can be separated by comma.

3.5. Host Helper

NVRTC defines the following functions for easier interaction with host code.

template < typename T > nvrtcResult nvrtcGetTypeName (std::string *result)

nvrtcGetTypeName stores the source level name of the template type argument T in the given std::string location.

Parameters

result

pointer to std::string in which to store the type name.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INTERNAL_ERROR

Description

This function is only provided when the macro NVRTC_GET_TYPE_NAME is defined with a non-zero value. It uses abi::__cxa_demangle or UnDecorateSymbolName function calls to extract the type name, when using gcc/clang or cl.exe compilers, respectively. If the name extraction fails, it will return NVRTC_INTERNAL_ERROR, otherwise *result is initialized with the extracted name.

Windows-specific notes:

- nvrtcGetTypeName() is not multi-thread safe because it calls UnDecorateSymbolName(), which is not multi-thread safe.
- The returned string may contain Microsoft-specific keywords such as __ptr64 and __cdecl.

nvrtcResult nvrtcGetTypeName (const std::type_info tinfo, std::string *result)

nvrtcGetTypeName stores the source level name of a type in the given std::string location.

Parameters

tinfo

reference to object of type std::type_info for a given type.

result

pointer to std::string in which to store the type name.

Returns

- NVRTC_SUCCESS

- NVRTC_ERROR_INTERNAL_ERROR

Description

This function is only provided when the macro NVRTC_GET_TYPE_NAME is defined with a non-zero value. It uses abi::__cxa_demangle or UnDecorateSymbolName function calls to extract the type name, when using gcc/clang or cl.exe compilers, respectively. If the name extraction fails, it will return NVRTC_INTERNAL_ERROR, otherwise *result is initialized with the extracted name.

Windows-specific notes:

- nvrtcGetTypeName() is not multi-thread safe because it calls UnDecorateSymbolName(), which is not multi-thread safe.
- The returned string may contain Microsoft-specific keywords such as __ptr64 and __cdecl.

Chapter 4. Language

Unlike the offline nvcc compiler, NVRTC is meant for compiling only device CUDA C++ code. It does not accept host code or host compiler extensions in the input code, unless otherwise noted.

4.1. Execution Space

NVRTC uses `__host__` as the default execution space, and it generates an error if it encounters any host code in the input. That is, if the input contains entities with explicit `__host__` annotations or no execution space annotation, NVRTC will emit an error. `__host__ __device__` functions are treated as device functions.

NVRTC provides a compile option, `--device-as-default-execution-space`, that enables an alternative compilation mode, in which entities with no execution space annotations are treated as `__device__` entities.

4.2. Separate Compilation

NVRTC itself does not provide any linker. Users can, however, use cuLinkAddData in the CUDA Driver API to link the generated relocatable PTX code with other relocatable code. To generate relocatable PTX code, the compile option `--relocatable-device-code=true` or `--device-c` is required.

4.3. Dynamic Parallelism

NVRTC supports dynamic parallelism under the following conditions:

- Compilation target must be compute 35 or higher.
- Either separate compilation (`--relocatable-device-code=true` or `--device-c`) or extensible whole program compilation (`--extensible-whole-program`) must be enabled.
- Generated PTX must be linked against the CUDA device runtime (cudadevrt) library (see Separate Compilation).

Example: Dynamic Parallelism provides a simple example.

4.4. Integer Size

Different operating systems define integer type sizes differently. Linux x86_64 implements LP64, and Windows x86_64 implements LLP64.

Table 1. Integer sizes in bits for LLP64 and LP64

	short	**int**	**long**	**long long**	**pointers and size_t**
LLP64	16	32	32	64	64
LP64	16	32	64	64	64

NVRTC implements LP64 on Linux and LLP64 on Windows.

NVRTC supports 128-bit integer types through the '__int128' type. This can be enabled with the `--device-int128` flag. 128-bit integer support is not available on Windows.

4.5. Include Syntax

When `nvrtcCompileProgram()` is called, the current working directory is added to the header search path used for locating files included with the quoted syntax (e.g., `#include "foo.h"`), before the code is compiled.

4.6. Predefined Macros

- `__CUDACC_RTC__`: useful for distinguishing between runtime and offline `nvcc` compilation in user code.
- `__CUDACC__`: defined with same semantics as with offline `nvcc` compilation.
- `__CUDACC_RDC__`: defined with same semantics as with offline `nvcc` compilation.
- `__CUDACC_EWP__`: defined with same semantics as with offline `nvcc` compilation.
- `__CUDACC_DEBUG__`: defined with same semantics as with offline `nvcc` compilation.
- `__CUDA_ARCH__`: defined with same semantics as with offline `nvcc` compilation.
- `__CUDA_ARCH_LIST__`: defined with same semantics as with offline `nvcc` compilation.
- `__CUDACC_VER_MAJOR__`: defined with the major version number as returned by `nvrtcVersion`.
- `__CUDACC_VER_MINOR__`: defined with the minor version number as returned by `nvrtcVersion`.

- `__CUDACC_VER_BUILD__`: defined with the build version number.
- `__NVCC_DIAG_PRAGMA_SUPPORT__`: defined with same semantics as with offline `nvcc` compilation.
- `__CUDACC_RTC_INT128__`: defined when `-device-int128` flag is specified during compilation, and indicates that `__int128` type is supported.
- `NULL`: null pointer constant.
- `va_start`
- `va_end`
- `va_arg`
- `va_copy` : defined when language dialect C++11 or later is selected.
- `__cplusplus`
- `_WIN64` : defined on Windows platforms.
- `__LP64__` : defined on non-Windows platforms where `long int` and pointer types are 64-bits.
- `__cdecl` : defined to empty on all platforms.
- `__ptr64` : defined to empty on Windows platforms.

4.7. Predefined Types

- `clock_t`
- `size_t`
- `ptrdiff_t`
- `va_list`: Note that the definition of this type may be different than the one selected by nvcc when compiling CUDA code.
- Predefined types such as `dim3`, `char4`, etc., that are available in the CUDA Runtime headers when compiling offline with `nvcc` are also available, unless otherwise noted.

4.8. Builtin Functions

Builtin functions provided by the CUDA Runtime headers when compiling offline with `nvcc` are available, unless otherwise noted.

Chapter 5. Basic Usage

This section of the document uses a simple example, *Single-Precision #·X Plus Y* (SAXPY), shown in Figure 1 to explain what is involved in runtime compilation with NVRTC. For brevity and readability, error checks on the API return values are not shown. The complete code listing is available in Example: SAXPY.

Figure 1. CUDA source string for SAXPY

```
const char *saxpy = "                                              \n\
extern \"C\" __global__                                            \n\
void saxpy(float a, float *x, float *y, float *out, size_t n)     \n\
{                                                                  \n\
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;              \n\
  if (tid < n) {                                                   \n\
    out[tid] = a * x[tid] + y[tid];                                \n\
  }                                                                \n\
}                                                                  \n";
```

First, an instance of nvrtcProgram needs to be created. Figure 2 shows creation of nvrtcProgram for SAXPY. As SAXPY does not require any header, 0 is passed as numHeaders, and NULL as headers and includeNames.

Figure 2. nvrtcProgram creation for SAXPY

```
nvrtcProgram prog;
nvrtcCreateProgram(&prog,          // prog
                   saxpy,          // buffer
                   "saxpy.cu",     // name
                   0,              // numHeaders
                   NULL,           // headers
                   NULL);          // includeNames
```

If SAXPY had any #include directives, the contents of the files that are #include'd can be passed as elements of headers, and their names as elements of includeNames. For example, #include <foo.h> and #include <bar.h> would require 2 as numHeaders, { "<contents of foo.h>", "<contents of bar.h>" } as headers, and { "foo.h", "bar.h" } as includeNames (<contents of foo.h> and <contents of bar.h> must be replaced by the actual contents of foo.h and bar.h). Alternatively, the compile option -I can be used if the header is guaranteed to exist in the file system at runtime.

Once the instance of nvrtcProgram for compilation is created, it can be compiled by nvrtcCompileProgram as shown in Figure 3. Two compile options are used in this example, --

gpu-architecture=compute_80 and --fmad=false, to generate code for the compute_80 architecture and to disable the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations. Other combinations of compile options can be used as needed and Supported Compile Options lists valid compile options.

Figure 3. Compilation of SAXPY for compute_80 with FMAD enabled

```
const char *opts[] = {"--gpu-architecture=compute_80",
                      "--fmad=false"};
nvrtcCompileProgram(prog,     // prog
                    2,        // numOptions
                    opts);    // options
```

After the compilation completes, users can obtain the program compilation log and the generated PTX as Figure 4 shows. NVRTC does not generate valid PTX when the compilation fails, and it may generate program compilation log even when the compilation succeeds if needed.

A nvrtcProgram can be compiled by nvrtcCompileProgram multiple times with different compile options, and users can only retrieve the PTX and the log generated by the last compilation.

Figure 4. Obtaining generated PTX and program compilation log

```
// Obtain compilation log from the program.
size_t logSize;
nvrtcGetProgramLogSize(prog, &logSize);
char *log = new char[logSize];
nvrtcGetProgramLog(prog, log);
// Obtain PTX from the program.
size_t ptxSize;
nvrtcGetPTXSize(prog, &ptxSize);
char *ptx = new char[ptxSize];
nvrtcGetPTX(prog, ptx);
```

When the instance of nvrtcProgram is no longer needed, it can be destroyed by nvrtcDestroyProgram as shown in Figure 5.

Figure 5. Destruction of nvrtcProgram

```
nvrtcDestroyProgram(&prog);
```

The generated PTX can be further manipulated by the CUDA Driver API for execution or linking. Figure 6 shows an example code sequence for execution of the generated PTX.

Figure 6. Execution of SAXPY using the PTX generated by NVRTC

```
CUdevice cuDevice;
CUcontext context;
CUmodule module;
CUfunction kernel;
cuInit(0);
cuDeviceGet(&cuDevice, 0);
cuCtxCreate(&context, 0, cuDevice);
cuModuleLoadDataEx(&module, ptx, 0, 0, 0);
cuModuleGetFunction(&kernel, module, "saxpy");
size_t n = size_t n = NUM_THREADS * NUM_BLOCKS;
size_t bufferSize = n * sizeof(float);
float a = ...;
float *hX = ..., *hY = ..., *hOut = ...;
CUdeviceptr dX, dY, dOut;
cuMemAlloc(&dX, bufferSize);
cuMemAlloc(&dY, bufferSize);
cuMemAlloc(&dOut, bufferSize);
cuMemcpyHtoD(dX, hX, bufferSize);
cuMemcpyHtoD(dY, hY, bufferSize);
void *args[] = { &a, &dX, &dY, &dOut, &n };
cuLaunchKernel(kernel,
               NUM_THREADS, 1, 1,   // grid dim
               NUM_BLOCKS, 1, 1,    // block dim
               0, NULL,             // shared mem and stream
               args,                // arguments
               0);
cuCtxSynchronize();
cuMemcpyDtoH(hOut, dOut, bufferSize);
```

Chapter 6. Accessing Lowered Names

6.1. Introduction

NVRTC will mangle `__global__` function names and names of `__device__` and `__constant__` variables as specified by the IA64 ABI. If the generated PTX is being loaded using the CUDA Driver API, the kernel function or `__device__`/`__constant__` variable must be looked up by name, but this is hard to do when the name has been mangled. To address this problem, NVRTC provides API functions that map source level `__global__` function or `__device__`/`__constant__` variable names to the mangled names present in the generated PTX.

The two API functions `nvrtcAddNameExpression` and `nvrtcGetLoweredName` work together to provide this functionality. First, a 'name expression' string denoting the address for the `__global__` function or `__device__`/`__constant__` variable is provided to `nvrtcAddNameExpression`. Then, the program is compiled with `nvrtcCompileProgram`. During compilation, NVRTC will parse the name expression string as a C++ constant expression at the end of the user program. The constant expression must provide the address of the `__global__` function or `__device__`/`__constant__` variable. Finally, the function `nvrtcGetLoweredName` is called with the original name expression and it returns a pointer to the lowered name. The lowered name can be used to refer to the kernel or variable in the CUDA Driver API.

NVRTC guarantees that any `__global__` function or `__device__`/`__constant__` variable referenced in a call to `nvrtcAddNameExpression` will be present in the generated PTX (if the definition is available in the input source code).

6.2. Example

Example: Using Lowered Name lists a complete runnable example. Some relevant snippets:

1. The GPU source code ('gpu_program') contains definitions of various `__global__` functions/function templates and `__device__`/`__constant__` variables:

```
const char *gpu_program = "                                      \n\
__device__ int V1; // set from host code                         \n\
static __global__ void f1(int *result) { *result = V1 + 10; }    \n\
namespace N1 {                                                   \n\
  namespace N2 {                                                 \n\
     __constant__ int V2; // set from host code                  \n\
```

```
    __global__ void f2(int *result) { *result = V2 + 20; }      \n\
  }                                                             \n\
}                                                               \n\
template<typename T>                                            \n\
__global__ void f3(int *result) { *result = sizeof(T); }        \n\
```

2. The host source code invokes `nvrtcAddNameExpression` with various name expressions referring to the address of `__global__` functions and `__device__`/`__constant__` variables:

```
kernel_name_vec.push_back("&f1");
..
kernel_name_vec.push_back("N1::N2::f2");
..
kernel_name_vec.push_back("f3<int>");
..
kernel_name_vec.push_back("f3<double>");

// add name expressions to NVRTC. Note this must be done before
// the program is compiled.
for (size_t i = 0; i < name_vec.size(); ++i)
NVRTC_SAFE_CALL(nvrtcAddNameExpression(prog, kernel_name_vec[i].c_str()));
..
// add expressions for  __device__ / __constant__ variables to NVRTC
variable_name_vec.push_back("&V1");
..
variable_name_vec.push_back("&N1::N2::V2");
..
for (size_t i = 0; i < variable_name_vec.size(); ++i)
  NVRTC_SAFE_CALL(nvrtcAddNameExpression(prog,
  variable_name_vec[i].c_str()));
```

3. The GPU program is then compiled with `nvrtcCompileProgram`. The generated PTX is loaded on the GPU. The mangled names of the `__device__`/`__constant__` variables and `__global__` functions are looked up:

```
// note: this call must be made after NVRTC program has been
// compiled and before it has been destroyed.
NVRTC_SAFE_CALL(nvrtcGetLoweredName(
prog,
variable_name_vec[i].c_str(), // name expression
&name                         // lowered name
));
..
NVRTC_SAFE_CALL(nvrtcGetLoweredName(
prog,
kernel_name_vec[i].c_str(), // name expression
&name // lowered name
));
```

4. The mangled name of the `__device__`/`__constant__` variable is then used to lookup the variable in the module and update its value using the CUDA Driver API:

```
CUdeviceptr variable_addr;
CUDA_SAFE_CALL(cuModuleGetGlobal(&variable_addr, NULL, module, name));
CUDA_SAFE_CALL(cuMemcpyHtoD(variable_addr,
&initial_value, sizeof(initial_value)));
```

5. The mangled name of the kernel is then used to launch it using the CUDA Driver API:

```
CUfunction kernel;
CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, name));
...
```

```
CUDA_SAFE_CALL(
cuLaunchKernel(kernel,
1, 1, 1, // grid dim
1, 1, 1, // block dim
0, NULL, // shared mem and stream
args, 0));
```

6.3. Notes

1. Sequence of calls: All name expressions must be added using `nvrtcAddNameExpression` before the NVRTC program is compiled with `nvrtcCompileProgram`. This is required because the name expressions are parsed at the end of the user program, and may trigger template instantiations. The lowered names must be looked up by calling `nvrtcGetLoweredName` only after the NVRTC program has been compiled, and before it has been destroyed. The pointer returned by `nvrtcGetLoweredName` points to memory owned by NVRTC, and this memory is freed when the NVRTC program has been destroyed (`nvrtcDestroyProgram`). Thus the correct sequence of calls is : `nvrtcAddNameExpression`, `nvrtcCompileProgram`, `nvrtcGetLoweredName`, `nvrtcDestroyProgram`.
2. Identical Name Expressions: The name expression string passed to `nvrtcAddNameExpression` and `nvrtcGetLoweredName` must have identical characters. For example, "foo" and "foo " are not identical strings, even though semantically they refer to the same entity (foo), because the second string has a extra whitespace character.
3. Constant Expressions: The characters in the name expression string are parsed as a C ++ constant expression at the end of the user program. Any errors during parsing will cause compilation failure and compiler diagnostics will be generated in the compilation log. The constant expression must refer to the address of a `__global__` function or `__device__`/`__constant__` variable.
4. Address of overloaded function: If the NVRTC source code has multiple overloaded `__global__` functions, then the name expression must use a cast operation to disambiguate. However, casts are not allowed in constant expression for C++ dialects before C++11. If using such name expressions, please compile the code in C++11 or later dialect using the '-std' command line flag. Example: Consider that the GPU code string contains:

```
__global__ void foo(int) { }
__global__ void foo(char) { }
```

The name expression `'(void(*)(int))foo'` correctly disambiguates `'foo(int)'`, but the program must be compiled in C++11 or later dialect (e.g. `'-std=c++11'`) because casts are not allowed in pre-C++11 constant expressions.

Chapter 7. Interfacing With Template Host Code

7.1. Introduction

In some scenarios, it is useful to instantiate `__global__` function templates in device code based on template arguments in host code. The NVRTC helper function `nvrtcGetTypeName` can be used to extract the source level name of a type in host code, and this string can be used to instantiate a `__global__` function template and get the mangled name of the instantiation using the `nvrtcAddNameExpression` and `nvrtcGetLoweredName` functions.

`nvrtcGetTypeName` is defined inline in the NVRTC header file, and is available when the macro `NVRTC_GET_TYPE_NAME` is defined with a non-zero value. It uses the `abi::__cxa_demangle` and `UnDecorateSymbolName` host code functions when using `gcc/clang` and `cl.exe` compilers, respectively. Users may need to specify additional header paths and libraries to find the host functions used (`abi::__cxa_demangle` / `UnDecorateSymbolName`). See the build instructions for the example below for reference (Build Instructions).

7.2. Example

Example: Using nvrtcGetTypeName lists a complete runnable example. Some relevant snippets:

1. The GPU source code ('gpu_program') contains definitions of a `__global__` function template:

```
const char *gpu_program = " \n\
namespace N1 { struct S1_t { int i; double d; }; } \n\
template<typename T> \n\
__global__ void f3(int *result) { *result = sizeof(T); } \n\
\n";
```

2. The host code function `getKernelNameForType` creates the name expression for a `__global__` function template instantiation based on the host template type `T`. The name of the type `T` is extracted using `nvrtcGetTypeName`:

```
template <typename T>
```

```
std::string getKernelNameForType(void)
{
// Look up the source level name string for the type "T" using
// nvrtcGetTypeName() and use it to create the kernel name
std::string type_name;
NVRTC_SAFE_CALL(nvrtcGetTypeName<T>(&type_name));
return std::string("f3<") + type_name + ">";
}
```

3. The name expressions are presented to NVRTC using the `nvrtcAddNameExpression` function:

```
name_vec.push_back(getKernelNameForType<int>());
..
name_vec.push_back(getKernelNameForType<double>());
..
name_vec.push_back(getKernelNameForType<N1::S1_t>());
..
for (size_t i = 0; i < name_vec.size(); ++i)
NVRTC_SAFE_CALL(nvrtcAddNameExpression(prog, name_vec[i].c_str()));
```

4. The GPU program is then compiled with `nvrtcCompileProgram`. The generated PTX is loaded on the GPU. The mangled names of the `__global__` function template instantiations are looked up:

```
// note: this call must be made after NVRTC program has been
// compiled and before it has been destroyed.
NVRTC_SAFE_CALL(nvrtcGetLoweredName(
prog,
name_vec[i].c_str(), // name expression
&name // lowered name
));
```

5. The mangled name is then used to launch the kernel using the CUDA Driver API:

```
CUfunction kernel;
CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, name));
...
CUDA_SAFE_CALL(
cuLaunchKernel(kernel,
1, 1, 1, // grid dim
1, 1, 1, // block dim
0, NULL, // shared mem and stream
args, 0));
```

Chapter 8. Versioning Scheme

8.1. NVRTC Shared Library Versioning

In the following, `MAJOR` and `MINOR` denote the major and minor versions of the CUDA Toolkit. e.g. for CUDA 11.2, `MAJOR` is "11" and `MINOR` is "2".

- Linux:
  - In CUDA toolkits prior to CUDA 11.3, the soname was set to "`MAJOR.MINOR`".
  - In CUDA 11.3 and later 11.x toolkits, the soname field is set to "`11.2`".
  - In CUDA toolkits with major version > 11 (e.g. CUDA 12.x), the soname field is set to "`MAJOR`".
- Windows:
  - In CUDA toolkits prior to cuda 11.3, the DLL name was of the form "`nvrtc64_XY_0.dll`", where `X` = `MAJOR`, `Y` = `MINOR`.
  - In CUDA 11.3 and later 11.x toolkits, the DLL name is "`nvrtc64_112_0.dll`".
  - In CUDA toolkits with major version > 11 (e.g. CUDA 12.x), the DLL name is of the form "`nvrtc64_X0_0.dll`" where `X` = `MAJOR`.

Consider a CUDA toolkit with major version > 11. The NVRTC shared library in this CUDA toolkit will have the same soname (Linux) or DLL name (Windows) as an NVRTC shared library in a previous minor version of the same CUDA toolkit. Similarly, the NVRTC shared library in CUDA 11.3 and later 11.x releases will have the same soname (Linux) or DLL name (Windows) as the NVRTC shared library in CUDA 11.2.

As a consequence of the versioning scheme described above, an NVRTC client that links against a particular NVRTC shared library will continue to work with a future NVRTC shared library with a matching soname (Linux) or DLL name (Windows). This allows the NVRTC client to take advantage of bug fixes and enhancements available in the more recent NVRTC shared library[1]. However, the more recent NVRTC shared library may generate PTX with a version that is not accepted by the CUDA Driver API functions of an older CUDA driver, as explained in the CUDA Compatibility document.

[1] Changes to compiler optimizer heuristics in the newer NVRTC shared library may also potentially cause performance perturbations for generated code.

Some approaches to resolving this issue:

- Install a more recent CUDA driver that is compatible with the CUDA toolkit containing the NVRTC library being used.
- Compile directly to SASS instead of PTX with NVRTC (see CUDA Compatibility document).

Alternately, an NVRTC client can either link against the static NVRTC library or redistribute a specific version of the NVRTC shared library and use `dlopen` (Linux) or `LoadLibrary` (Windows) functions to use that library at run time. Either approach allows the NVRTC client to maintain control over the version of NVRTC being used during deployment, to ensure predictable functionality and performance.

8.2. NVRTC-builtins Library

The NVRTC-builtins library contains helper code that is part of the NVRTC package. It is only used by the NVRTC library internally. Each NVRTC library is only compatible with the NVRTC-builtins library from the same CUDA toolkit.

Chapter 9. Miscellaneous Notes

9.1. Thread safety

Multiple threads can invoke NVRTC API functions concurrently, as long as there is no race condition. In this context, a race condition is defined to occur if multiple threads concurrently invoke NVRTC API functions with the same `nvrtcProgram` argument, where at least one thread is invoking either `nvrtcCompileProgram` or `nvrtcAddNameExpression` [2].

9.2. Stack Size

On Linux, NVRTC will increase the stack size to the maximum allowed using the `setrlimit()` function during compilation. This reduces the chance that the compiler will run out of stack when processing complex input sources. The stack size is reset to the previous value when compilation is completed.

Because `setrlimit()` changes the stack size for the entire process, it will also affect other application threads that may be executing concurrently. The command line flag `-modify-stack-limit=false` will prevent NVRTC from modifying the stack limit.

9.3. NVRTC Static Library

The NVRTC static library references functions defined in the NVRTC-builtins static library and the PTX compiler static library. Please see Build Instructions for an example.

[2] These API functions modify the state of the associated `nvrtcProgram`.

Appendix A. Example: SAXPY

A.1. Code (saxpy.cpp)

```
#include <nvrtc.h>
#include <cuda.h>
#include <iostream>

#define NUM_THREADS 128
#define NUM_BLOCKS 32
#define NVRTC_SAFE_CALL(x)                                        \
  do {                                                            \
    nvrtcResult result = x;                                       \
    if (result != NVRTC_SUCCESS) {                                \
      std::cerr << "\nerror: " #x " failed with error "           \
                << nvrtcGetErrorString(result) << '\n';           \
      exit(1);                                                    \
    }                                                             \
  } while(0)
#define CUDA_SAFE_CALL(x)                                         \
  do {                                                            \
    CUresult result = x;                                          \
    if (result != CUDA_SUCCESS) {                                 \
      const char *msg;                                            \
      cuGetErrorName(result, &msg);                               \
      std::cerr << "\nerror: " #x " failed with error "           \
                << msg << '\n';                                   \
      exit(1);                                                    \
    }                                                             \
  } while(0)

const char *saxpy = "                                           \n\
extern \"C\" __global__                                         \n\
void saxpy(float a, float *x, float *y, float *out, size_t n)   \n\
{                                                               \n\
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;           \n\
  if (tid < n) {                                                \n\
    out[tid] = a * x[tid] + y[tid];                             \n\
  }                                                             \n\
}                                                               \n";

int main()
{
  // Create an instance of nvrtcProgram with the SAXPY code string.
  nvrtcProgram prog;
```

```
  NVRTC_SAFE_CALL(
    nvrtcCreateProgram(&prog,         // prog
                       saxpy,         // buffer
```

```
                       "saxpy.cu",     // name
                       0,              // numHeaders
                       NULL,           // headers
                       NULL));         // includeNames
  // Compile the program with fmad disabled.
  // Note: Can specify GPU target architecture explicitly with '-arch' flag.
  const char *opts[] = {"--fmad=false"};
  nvrtcResult compileResult = nvrtcCompileProgram(prog,  // prog
                                                  1,     // numOptions
                                                  opts); // options
  // Obtain compilation log from the program.
  size_t logSize;
  NVRTC_SAFE_CALL(nvrtcGetProgramLogSize(prog, &logSize));
  char *log = new char[logSize];
  NVRTC_SAFE_CALL(nvrtcGetProgramLog(prog, log));
  std::cout << log << '\n';
  delete[] log;
  if (compileResult != NVRTC_SUCCESS) {
    exit(1);
  }
  // Obtain PTX from the program.
  size_t ptxSize;
  NVRTC_SAFE_CALL(nvrtcGetPTXSize(prog, &ptxSize));
  char *ptx = new char[ptxSize];
  NVRTC_SAFE_CALL(nvrtcGetPTX(prog, ptx));
  // Destroy the program.
  NVRTC_SAFE_CALL(nvrtcDestroyProgram(&prog));
  // Load the generated PTX and get a handle to the SAXPY kernel.
  CUdevice cuDevice;
  CUcontext context;
  CUmodule module;
  CUfunction kernel;
  CUDA_SAFE_CALL(cuInit(0));
  CUDA_SAFE_CALL(cuDeviceGet(&cuDevice, 0));
  CUDA_SAFE_CALL(cuCtxCreate(&context, 0, cuDevice));
  CUDA_SAFE_CALL(cuModuleLoadDataEx(&module, ptx, 0, 0, 0));
  CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, "saxpy"));
  // Generate input for execution, and create output buffers.
  size_t n = NUM_THREADS * NUM_BLOCKS;
  size_t bufferSize = n * sizeof(float);
  float a = 5.1f;
  float *hX = new float[n], *hY = new float[n], *hOut = new float[n];
  for (size_t i = 0; i < n; ++i) {
    hX[i] = static_cast<float>(i);
    hY[i] = static_cast<float>(i * 2);
  }
  CUdeviceptr dX, dY, dOut;
  CUDA_SAFE_CALL(cuMemAlloc(&dX, bufferSize));
  CUDA_SAFE_CALL(cuMemAlloc(&dY, bufferSize));
  CUDA_SAFE_CALL(cuMemAlloc(&dOut, bufferSize));
  CUDA_SAFE_CALL(cuMemcpyHtoD(dX, hX, bufferSize));
  CUDA_SAFE_CALL(cuMemcpyHtoD(dY, hY, bufferSize));
  // Execute SAXPY.
  void *args[] = { &a, &dX, &dY, &dOut, &n };
  CUDA_SAFE_CALL(
    cuLaunchKernel(kernel,
                   NUM_BLOCKS, 1, 1,    // grid dim
                   NUM_THREADS, 1, 1,   // block dim
                   0, NULL,             // shared mem and stream
                   args, 0));           // arguments
  CUDA_SAFE_CALL(cuCtxSynchronize());
  // Retrieve and print output.
  CUDA_SAFE_CALL(cuMemcpyDtoH(hOut, dOut, bufferSize));
```

```
  for (size_t i = 0; i < n; ++i) {
    std::cout << a << " * " << hX[i] << " + " << hY[i]
```

```
              << " = " << hOut[i] << '\n';
  }
  // Release resources.
  CUDA_SAFE_CALL(cuMemFree(dX));
  CUDA_SAFE_CALL(cuMemFree(dY));
  CUDA_SAFE_CALL(cuMemFree(dOut));
  CUDA_SAFE_CALL(cuModuleUnload(module));
  CUDA_SAFE_CALL(cuCtxDestroy(context));
  delete[] hX;
  delete[] hY;
  delete[] hOut;
  delete[] ptx;
  return 0;
}
```

A.2. Build Instructions

Assuming the environment variable `CUDA_PATH` points to CUDA Toolkit installation directory, build this example as:

- With NVRTC shared library:
  - Windows:

    ```
    cl.exe saxpy.cpp /Fesaxpy ^
          /I "%CUDA_PATH%"\include ^
          "%CUDA_PATH%"\lib\x64\nvrtc.lib "%CUDA_PATH%"\lib\x64\cuda.lib
    ```

  - Linux:

    ```
    g++ saxpy.cpp -o saxpy \
          -I $CUDA_PATH/include \
          -L $CUDA_PATH/lib64 \
          -lnvrtc -lcuda \
          -Wl,-rpath,$CUDA_PATH/lib64
    ```

- With NVRTC static library:
  - Windows:

    ```
    cl.exe saxpy.cpp /Fesaxpy  ^
           /I "%CUDA_PATH%"\include ^
           "%CUDA_PATH%"\lib\x64\nvrtc_static.lib ^
           "%CUDA_PATH%"\lib\x64\nvrtc-builtins_static.lib ^
           "%CUDA_PATH%"\lib\x64\nvptxcompiler_static.lib ^
           "%CUDA_PATH%"\lib\x64\cuda.lib user32.lib Ws2_32.lib
    ```

  - Linux:

    ```
    g++ saxpy.cpp -o saxpy \
          -I $CUDA_PATH/include \
          -L $CUDA_PATH/lib64 \
          -lnvrtc_static -lnvrtc-builtins_static -lnvptxcompiler_static -lcuda \
          -lpthread
    ```

Appendix B. Example: Using Lowered Name

B.1. Code (lowered-name.cpp)

```
#include <nvrtc.h>
#include <cuda.h>
#include <iostream>
#include <vector>
#include <string>

#define NVRTC_SAFE_CALL(x)                                        \
  do {                                                            \
    nvrtcResult result = x;                                       \
    if (result != NVRTC_SUCCESS) {                                \
      std::cerr << "\nerror: " #x " failed with error "           \
                << nvrtcGetErrorString(result) << '\n';           \
      exit(1);                                                    \
    }                                                             \
  } while(0)
#define CUDA_SAFE_CALL(x)                                         \
  do {                                                            \
    CUresult result = x;                                          \
    if (result != CUDA_SUCCESS) {                                 \
      const char *msg;                                            \
      cuGetErrorName(result, &msg);                               \
      std::cerr << "\nerror: " #x " failed with error "           \
                << msg << '\n';                                   \
      exit(1);                                                    \
    }                                                             \
  } while(0)

const char *gpu_program = "                                     \n\
__device__ int V1; // set from host code                        \n\
static __global__ void f1(int *result) { *result = V1 + 10; }   \n\
namespace N1 {                                                  \n\
  namespace N2 {                                                \n\
     __constant__ int V2; // set from host code                 \n\
    __global__ void f2(int *result) { *result = V2 + 20; }      \n\
  }                                                             \n\
}                                                               \n\
template<typename T>                                            \n\
__global__ void f3(int *result) { *result = sizeof(T); }        \n\
                                                                \n";
```

```
int main()
{
  // Create an instance of nvrtcProgram
  nvrtcProgram prog;
  NVRTC_SAFE_CALL(nvrtcCreateProgram(&prog,         // prog
                                     gpu_program,   // buffer
                                     "prog.cu",     // name
                                     0,             // numHeaders
                                     NULL,          // headers
                                     NULL));        // includeNames

  // add all name expressions for kernels
  std::vector<std::string> kernel_name_vec;
  std::vector<std::string> variable_name_vec;
  std::vector<int> variable_initial_value;

  std::vector<int> expected_result;

  // note the name expressions are parsed as constant expressions
  kernel_name_vec.push_back("&f1");
  expected_result.push_back(10 + 100);

  kernel_name_vec.push_back("N1::N2::f2");
  expected_result.push_back(20 + 200);

  kernel_name_vec.push_back("f3<int>");
  expected_result.push_back(sizeof(int));

  kernel_name_vec.push_back("f3<double>");
  expected_result.push_back(sizeof(double));

  // add kernel name expressions to NVRTC. Note this must be done before
  // the program is compiled.
  for (size_t i = 0; i < kernel_name_vec.size(); ++i)
    NVRTC_SAFE_CALL(nvrtcAddNameExpression(prog, kernel_name_vec[i].c_str()));

  // add expressions for  __device__ / __constant__ variables to NVRTC
  variable_name_vec.push_back("&V1");
  variable_initial_value.push_back(100);

  variable_name_vec.push_back("&N1::N2::V2");
  variable_initial_value.push_back(200);

  for (size_t i = 0; i < variable_name_vec.size(); ++i)
    NVRTC_SAFE_CALL(nvrtcAddNameExpression(prog, variable_name_vec[i].c_str()));

  nvrtcResult compileResult = nvrtcCompileProgram(prog,  // prog
                                                  0,     // numOptions
                                                  NULL); // options
  // Obtain compilation log from the program.
  size_t logSize;
  NVRTC_SAFE_CALL(nvrtcGetProgramLogSize(prog, &logSize));
  char *log = new char[logSize];
  NVRTC_SAFE_CALL(nvrtcGetProgramLog(prog, log));
  std::cout << log << '\n';
  delete[] log;
  if (compileResult != NVRTC_SUCCESS) {
    exit(1);
  }
  // Obtain PTX from the program.
  size_t ptxSize;
  NVRTC_SAFE_CALL(nvrtcGetPTXSize(prog, &ptxSize));
  char *ptx = new char[ptxSize];
  NVRTC_SAFE_CALL(nvrtcGetPTX(prog, ptx));
```

```
// Load the generated PTX
CUdevice cuDevice;
CUcontext context;
CUmodule module;

CUDA_SAFE_CALL(cuInit(0));
CUDA_SAFE_CALL(cuDeviceGet(&cuDevice, 0));
CUDA_SAFE_CALL(cuCtxCreate(&context, 0, cuDevice));
CUDA_SAFE_CALL(cuModuleLoadDataEx(&module, ptx, 0, 0, 0));

CUdeviceptr dResult;
int hResult = 0;
CUDA_SAFE_CALL(cuMemAlloc(&dResult, sizeof(hResult)));
CUDA_SAFE_CALL(cuMemcpyHtoD(dResult, &hResult, sizeof(hResult)));

// for each of the __device__/__constant__ variable address
// expressions provided to NVRTC, extract the lowered name for the
// corresponding variable, and set its value
for (size_t i = 0; i < variable_name_vec.size(); ++i) {
  const char *name;

  // note: this call must be made after NVRTC program has been
  // compiled and before it has been destroyed.
  NVRTC_SAFE_CALL(nvrtcGetLoweredName(
                      prog,
   variable_name_vec[i].c_str(), // name expression
   &name                         // lowered name
                                   ));
  int initial_value = variable_initial_value[i];

  // get pointer to variable using lowered name, and set its
  // initial value
  CUdeviceptr variable_addr;
  CUDA_SAFE_CALL(cuModuleGetGlobal(&variable_addr, NULL, module, name));
  CUDA_SAFE_CALL(cuMemcpyHtoD(variable_addr,
&initial_value, sizeof(initial_value)));
}


// for each of the kernel name expressions previously provided to NVRTC,
// extract the lowered name for corresponding __global__ function,
// and launch it.

for (size_t i = 0; i < kernel_name_vec.size(); ++i) {
  const char *name;

  // note: this call must be made after NVRTC program has been
  // compiled and before it has been destroyed.
  NVRTC_SAFE_CALL(nvrtcGetLoweredName(
                      prog,
   kernel_name_vec[i].c_str(), // name expression
   &name                // lowered name
                                   ));

  // get pointer to kernel from loaded PTX
  CUfunction kernel;
  CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, name));
```

```
  // launch the kernel
  std::cout << "\nlaunching " << name << " ("
     << kernel_name_vec[i] << ")" << std::endl;

  void *args[] = { &dResult };
```

```
    CUDA_SAFE_CALL(
      cuLaunchKernel(kernel,
        1, 1, 1,              // grid dim
        1, 1, 1,              // block dim
        0, NULL,              // shared mem and stream
        args, 0));            // arguments
    CUDA_SAFE_CALL(cuCtxSynchronize());

    // Retrieve the result
    CUDA_SAFE_CALL(cuMemcpyDtoH(&hResult, dResult, sizeof(hResult)));

    // check against expected value
    if (expected_result[i] != hResult) {
      std::cout << "\n Error: expected result = " << expected_result[i]
                << " , actual result = " << hResult << std::endl;
      exit(1);
    }
  }  // for

  // Release resources.
  CUDA_SAFE_CALL(cuMemFree(dResult));
  CUDA_SAFE_CALL(cuModuleUnload(module));
  CUDA_SAFE_CALL(cuCtxDestroy(context));
  delete[] ptx;

  // Destroy the program.
  NVRTC_SAFE_CALL(nvrtcDestroyProgram(&prog));

  return 0;
}
```

B.2. Build Instructions

Assuming the environment variable `CUDA_PATH` points to CUDA Toolkit installation directory, build this example as:

- With NVRTC shared library:
  - Windows:

```
cl.exe lowered-name.cpp /Felowered-name ^
      /I "%CUDA_PATH%"\include ^
      "%CUDA_PATH%"\lib\x64\nvrtc.lib "%CUDA_PATH%"\lib\x64\cuda.lib
```

  - Linux:

```
g++ lowered-name.cpp -o lowered-name \
      -I $CUDA_PATH/include \
      -L $CUDA_PATH/lib64 \
      -lnvrtc -lcuda \
      -Wl,-rpath,$CUDA_PATH/lib64
```

- With NVRTC static library:
  - Windows:

```
cl.exe lowered-name.cpp /Felowered-name  ^
      /I "%CUDA_PATH%"\include ^
      "%CUDA_PATH%"\lib\x64\nvrtc_static.lib ^
      "%CUDA_PATH%"\lib\x64\nvrtc-builtins_static.lib ^
      "%CUDA_PATH%"\lib\x64\nvptxcompiler_static.lib ^
      "%CUDA_PATH%"\lib\x64\cuda.lib user32.lib Ws2_32.lib
```

- Linux:

```
g++ lowered-name.cpp -o lowered-name \
      -I $CUDA_PATH/include \
      -L $CUDA_PATH/lib64 \
      -lnvrtc_static -lnvrtc-builtins_static -lnvptxcompiler_static \
      -lcuda -lpthread
```

Appendix C. Example: Using nvrtcGetTypeName

C.1. Code (host-type-name.cpp)

```
#include <nvrtc.h>
#include <cuda.h>
#include <iostream>
#include <vector>
#include <string>

#define NVRTC_SAFE_CALL(x)                                        \
  do {                                                            \
    nvrtcResult result = x;                                       \
    if (result != NVRTC_SUCCESS) {                                \
      std::cerr << "\nerror: " #x " failed with error "           \
                << nvrtcGetErrorString(result) << '\n';           \
      exit(1);                                                    \
    }                                                             \
  } while(0)
#define CUDA_SAFE_CALL(x)                                         \
  do {                                                            \
    CUresult result = x;                                          \
    if (result != CUDA_SUCCESS) {                                 \
      const char *msg;                                            \
      cuGetErrorName(result, &msg);                               \
      std::cerr << "\nerror: " #x " failed with error "           \
                << msg << '\n';                                   \
      exit(1);                                                    \
    }                                                             \
  } while(0)

const char *gpu_program = "                                     \n\
namespace N1 { struct S1_t { int i; double d; }; }              \n\
template<typename T>                                            \n\
__global__ void f3(int *result) { *result = sizeof(T); }        \n\
                                                                \n";


// note: this structure is also defined in GPU code string. Should ideally
// be in a header file included by both GPU code string and by CPU code.
namespace N1 { struct S1_t { int i; double d; }; };
```

```
template <typename T>
std::string getKernelNameForType(void)
{
```

```
   // Look up the source level name string for the type "T" using
   // nvrtcGetTypeName() and use it to create the kernel name
   std::string type_name;
   NVRTC_SAFE_CALL(nvrtcGetTypeName<T>(&type_name));
   return std::string("f3<") + type_name + ">";
}

int main()
{
  // Create an instance of nvrtcProgram
  nvrtcProgram prog;
  NVRTC_SAFE_CALL(
    nvrtcCreateProgram(&prog,         // prog
                       gpu_program,   // buffer
                       "gpu_program.cu",    // name
                       0,             // numHeaders
                       NULL,          // headers
                       NULL));        // includeNames

  // add all name expressions for kernels
  std::vector<std::string> name_vec;
  std::vector<int> expected_result;

  // note the name expressions are parsed as constant expressions
  name_vec.push_back(getKernelNameForType<int>());
  expected_result.push_back(sizeof(int));

  name_vec.push_back(getKernelNameForType<double>());
  expected_result.push_back(sizeof(double));

  name_vec.push_back(getKernelNameForType<N1::S1_t>());
  expected_result.push_back(sizeof(N1::S1_t));


  // add name expressions to NVRTC. Note this must be done before
  // the program is compiled.
  for (size_t i = 0; i < name_vec.size(); ++i)
    NVRTC_SAFE_CALL(nvrtcAddNameExpression(prog, name_vec[i].c_str()));

  nvrtcResult compileResult = nvrtcCompileProgram(prog,  // prog
                                                  0,     // numOptions
                                                  NULL); // options
  // Obtain compilation log from the program.
  size_t logSize;
  NVRTC_SAFE_CALL(nvrtcGetProgramLogSize(prog, &logSize));
  char *log = new char[logSize];
  NVRTC_SAFE_CALL(nvrtcGetProgramLog(prog, log));
  std::cout << log << '\n';
  delete[] log;
  if (compileResult != NVRTC_SUCCESS) {
    exit(1);
  }
  // Obtain PTX from the program.
  size_t ptxSize;
  NVRTC_SAFE_CALL(nvrtcGetPTXSize(prog, &ptxSize));
  char *ptx = new char[ptxSize];
  NVRTC_SAFE_CALL(nvrtcGetPTX(prog, ptx));
```

```
  // Load the generated PTX
  CUdevice cuDevice;
  CUcontext context;
  CUmodule module;
```

```
  CUDA_SAFE_CALL(cuInit(0));
  CUDA_SAFE_CALL(cuDeviceGet(&cuDevice, 0));
  CUDA_SAFE_CALL(cuCtxCreate(&context, 0, cuDevice));
  CUDA_SAFE_CALL(cuModuleLoadDataEx(&module, ptx, 0, 0, 0));

  CUdeviceptr dResult;
  int hResult = 0;
  CUDA_SAFE_CALL(cuMemAlloc(&dResult, sizeof(hResult)));
  CUDA_SAFE_CALL(cuMemcpyHtoD(dResult, &hResult, sizeof(hResult)));

  // for each of the name expressions previously provided to NVRTC,
  // extract the lowered name for corresponding __global__ function,
  // and launch it.

  for (size_t i = 0; i < name_vec.size(); ++i) {
    const char *name;

    // note: this call must be made after NVRTC program has been
    // compiled and before it has been destroyed.
    NVRTC_SAFE_CALL(nvrtcGetLoweredName(
                          prog,
     name_vec[i].c_str(), // name expression
     &name                // lowered name
                                        ));

    // get pointer to kernel from loaded PTX
    CUfunction kernel;
    CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, name));

    // launch the kernel
    std::cout << "\nlaunching " << name << " ("
       << name_vec[i] << ")" << std::endl;

    void *args[] = { &dResult };
    CUDA_SAFE_CALL(
      cuLaunchKernel(kernel,
       1, 1, 1,             // grid dim
       1, 1, 1,             // block dim
       0, NULL,             // shared mem and stream
       args, 0));           // arguments
    CUDA_SAFE_CALL(cuCtxSynchronize());

    // Retrieve the result
    CUDA_SAFE_CALL(cuMemcpyDtoH(&hResult, dResult, sizeof(hResult)));

    // check against expected value
    if (expected_result[i] != hResult) {
      std::cout << "\n Error: expected result = " << expected_result[i]
  << " , actual result = " << hResult << std::endl;
      exit(1);
    }
  }  // for

  // Release resources.
  CUDA_SAFE_CALL(cuMemFree(dResult));
  CUDA_SAFE_CALL(cuModuleUnload(module));
  CUDA_SAFE_CALL(cuCtxDestroy(context));
  delete[] ptx;

  // Destroy the program.
  NVRTC_SAFE_CALL(nvrtcDestroyProgram(&prog));

  return 0;
}
```

C.2. Build Instructions

Assuming the environment variable `CUDA_PATH` points to CUDA Toolkit installation directory, build this example as:

- With NVRTC shared library:
  - Windows:

```
cl.exe -DNVRTC_GET_TYPE_NAME=1 host-type-name.cpp /Fehost-type-name ^
      /I "%CUDA_PATH%"\include ^
      "%CUDA_PATH%"\lib\x64\nvrtc.lib "%CUDA_PATH%"\lib\x64\cuda.lib
 DbgHelp.lib
```

  - Linux:

```
g++ -DNVRTC_GET_TYPE_NAME=1 host-type-name.cpp -o host-type-name \
      -I $CUDA_PATH/include \
      -L $CUDA_PATH/lib64 \
      -lnvrtc -lcuda \
      -Wl,-rpath,$CUDA_PATH/lib64
```

- With NVRTC static library:
  - Windows:

```
cl.exe -DNVRTC_GET_TYPE_NAME=1 host-type-name.cpp /Fehost-type-name  ^
      /I "%CUDA_PATH%"\include ^
      "%CUDA_PATH%"\lib\x64\nvrtc_static.lib ^
      "%CUDA_PATH%"\lib\x64\nvrtc-builtins_static.lib ^
      "%CUDA_PATH%"\lib\x64\nvptxcompiler_static.lib ^
      "%CUDA_PATH%"\lib\x64\cuda.lib DbgHelp.lib user32.lib Ws2_32.lib
```

  - Linux:

```
g++ -DNVRTC_GET_TYPE_NAME=1 host-type-name.cpp -o host-type-name \
      -I $CUDA_PATH/include \
      -L $CUDA_PATH/lib64 \
      -lnvrtc_static -lnvrtc-builtins_static -lnvptxcompiler_static \
      -lcuda -lpthread
```

Appendix D. Example: Dynamic Parallelism

D.1. Code (dynamic-parallelism.cpp)

```
#include <nvrtc.h>
#include <cuda.h>
#include <iostream>

#define NVRTC_SAFE_CALL(x)                                        \
  do {                                                            \
    nvrtcResult result = x;                                       \
    if (result != NVRTC_SUCCESS) {                                \
      std::cerr << "\nerror: " #x " failed with error "           \
                << nvrtcGetErrorString(result) << '\n';           \
      exit(1);                                                    \
    }                                                             \
  } while(0)
#define CUDA_SAFE_CALL(x)                                         \
  do {                                                            \
    CUresult result = x;                                          \
    if (result != CUDA_SUCCESS) {                                 \
      const char *msg;                                            \
      cuGetErrorName(result, &msg);                               \
      std::cerr << "\nerror: " #x " failed with error "           \
                << msg << '\n';                                   \
      exit(1);                                                    \
    }                                                             \
  } while(0)

const char *dynamic_parallelism = "                             \n\
extern \"C\" __global__                                         \n\
void child(float *out, size_t n)                                \n\
{                                                               \n\
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;           \n\
  if (tid < n) {                                                \n\
    out[tid] = tid;                                             \n\
  }                                                             \n\
}                                                               \n\
                                                                \n\
extern \"C\" __global__                                         \n\
void parent(float *out, size_t n,                               \n\
            size_t numBlocks, size_t numThreads)                \n\
{                                                               \n\
  child<<<numBlocks, numThreads>>>(out, n);                     \n\
  cudaDeviceSynchronize();                                      \n\
}                                                               \n";
```

```
int main(int argc, char *argv[])
{
  if (argc < 2) {
    std::cout << "Usage: dynamic-parallelism <path to cudadevrt library>\n\n"
              << "<path to cudadevrt library> must include the cudadevrt\n"
              << "library name itself, e.g., Z:\\path\\to\\cudadevrt.lib on \n"
              << "Windows and /path/to/libcudadevrt.a on Linux.\n";
    exit(1);
  }
  size_t numBlocks = 32;
  size_t numThreads = 128;
  // Create an instance of nvrtcProgram with the code string.
  nvrtcProgram prog;
  NVRTC_SAFE_CALL(
    nvrtcCreateProgram(&prog,                       // prog
                       dynamic_parallelism,         // buffer
                       "dynamic_parallelism.cu",    // name
                       0,                           // numHeaders
                       NULL,                        // headers
                       NULL));                      // includeNames
  // Compile the program for compute_35 with rdc enabled.
  const char *opts[] = {"--gpu-architecture=compute_35",
                        "--relocatable-device-code=true"};
  nvrtcResult compileResult = nvrtcCompileProgram(prog,  // prog
                                                  2,     // numOptions
                                                  opts); // options
  // Obtain compilation log from the program.
  size_t logSize;
  NVRTC_SAFE_CALL(nvrtcGetProgramLogSize(prog, &logSize));
  char *log = new char[logSize];
  NVRTC_SAFE_CALL(nvrtcGetProgramLog(prog, log));
  std::cout << log << '\n';
  delete[] log;
  if (compileResult != NVRTC_SUCCESS) {
    exit(1);
  }
  // Obtain PTX from the program.
  size_t ptxSize;
  NVRTC_SAFE_CALL(nvrtcGetPTXSize(prog, &ptxSize));
  char *ptx = new char[ptxSize];
  NVRTC_SAFE_CALL(nvrtcGetPTX(prog, ptx));
  // Destroy the program.
  NVRTC_SAFE_CALL(nvrtcDestroyProgram(&prog));
  // Load the generated PTX and get a handle to the parent kernel.
  CUdevice cuDevice;
  CUcontext context;
  CUlinkState linkState;
  CUmodule module;
  CUfunction kernel;
  CUDA_SAFE_CALL(cuInit(0));
  CUDA_SAFE_CALL(cuDeviceGet(&cuDevice, 0));
  CUDA_SAFE_CALL(cuCtxCreate(&context, 0, cuDevice));
  CUDA_SAFE_CALL(cuLinkCreate(0, 0, 0, &linkState));
  CUDA_SAFE_CALL(cuLinkAddFile(linkState, CU_JIT_INPUT_LIBRARY, argv[1],
                               0, 0, 0));
  CUDA_SAFE_CALL(cuLinkAddData(linkState, CU_JIT_INPUT_PTX,
                               (void *)ptx, ptxSize, "dynamic_parallelism.ptx",
                               0, 0, 0));
  size_t cubinSize;
  void *cubin;
  CUDA_SAFE_CALL(cuLinkComplete(linkState, &cubin, &cubinSize));
  CUDA_SAFE_CALL(cuModuleLoadData(&module, cubin));
  CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, "parent"));
```

```
  // Generate input for execution, and create output buffers.
```

```
  size_t n = numBlocks * numThreads;
  size_t bufferSize = n * sizeof(float);
  float *hOut = new float[n];
  CUdeviceptr dX, dY, dOut;
  CUDA_SAFE_CALL(cuMemAlloc(&dOut, bufferSize));
  // Execute parent kernel.
  void *args[] = { &dOut, &n, &numBlocks, &numThreads };
  CUDA_SAFE_CALL(
    cuLaunchKernel(kernel,
                   1, 1, 1,    // grid dim
                   1, 1, 1,    // block dim
                   0, NULL,    // shared mem and stream
                   args, 0));  // arguments
  CUDA_SAFE_CALL(cuCtxSynchronize());
  // Retrieve and print output.
  CUDA_SAFE_CALL(cuMemcpyDtoH(hOut, dOut, bufferSize));

  for (size_t i = 0; i < n; ++i) {
    std::cout << hOut[i] << '\n';
  }
  // Release resources.
  CUDA_SAFE_CALL(cuMemFree(dOut));
  CUDA_SAFE_CALL(cuModuleUnload(module));
  CUDA_SAFE_CALL(cuLinkDestroy(linkState));
  CUDA_SAFE_CALL(cuCtxDestroy(context));
  delete[] hOut;
  delete[] ptx;
  return 0;
}
```

D.2. Build Instructions

Assuming the environment variable `CUDA_PATH` points to CUDA Toolkit installation directory, build this example as:

- With NVRTC shared library:
  - Windows:

```
cl.exe dynamic-parallelism.cpp /Fedynamic-parallelism ^
      /I "%CUDA_PATH%\include" ^
      "%CUDA_PATH%"\lib\x64\nvrtc.lib "%CUDA_PATH%"\lib\x64\cuda.lib
```

  - Linux:

```
g++ dynamic-parallelism.cpp -o dynamic-parallelism \
      -I $CUDA_PATH/include \
      -L $CUDA_PATH/lib64 \
      -lnvrtc -lcuda \
      -Wl,-rpath,$CUDA_PATH/lib64
```

- With NVRTC static library:
  - Windows:

```
cl.exe dynamic-parallelism.cpp /Fedynamic-parallelism  ^
      /I "%CUDA_PATH%"\include ^
      "%CUDA_PATH%"\lib\x64\nvrtc_static.lib ^
      "%CUDA_PATH%"\lib\x64\nvrtc-builtins_static.lib ^
      "%CUDA_PATH%"\lib\x64\nvptxcompiler_static.lib ^
      "%CUDA_PATH%"\lib\x64\cuda.lib user32.lib Ws2_32.lib
```

- Linux:

```
g++ dynamic-parallelism.cpp -o dynamic-parallelism \
      -I $CUDA_PATH/include \
      -L $CUDA_PATH/lib64 \
      -lnvrtc_static -lnvrtc-builtins_static -lnvptxcompiler_static -lcuda \
      -lpthread
```

Appendix E. Example: Device LTO (link time optimization)

This section demonstrates device link time optimization (LTO). There are two units of NVVM IR. The first unit is generated offline using `nvcc`, by specifying the architecture as '`-arch lto_XX`' (see offline.cu). The generated NVVM IR is packaged in a fatbinary.

The second unit is generated online using NVRTC, by specifying the flag '`-dlto`' (see online.cpp).

These two units are then passed to `cuLink*` CUDA Driver API functions, which link together the NVVM IR, run the optimizer on the linked IR and generate a cubin (see online.cpp). The cubin is then loaded on the GPU and executed.

E.1. Code (offline.cu)

```
__device__  float compute(float a, float x, float y) {
  return a * x + y;
}
```

E.2. Code (online.cpp)

```
#include <nvrtc.h>
#include <cuda.h>
#include <iostream>

#define NUM_THREADS 128
#define NUM_BLOCKS 32

#define NVRTC_SAFE_CALL(x)                                        \
  do {                                                            \
    nvrtcResult result = x;                                       \
    if (result != NVRTC_SUCCESS) {                                \
      std::cerr << "\nerror: " #x " failed with error "           \
                << nvrtcGetErrorString(result) << '\n';           \
      exit(1);                                                    \
    }                                                             \
  } while(0)
#define CUDA_SAFE_CALL(x)                                         \
  do {                                                            \
    CUresult result = x;                                          \
```

```
    if (result != CUDA_SUCCESS) {                                  \
      const char *msg;                                             \
      cuGetErrorName(result, &msg);                                \
      std::cerr << "\nerror: " #x " failed with error "            \
                << msg << '\n';                                    \
      exit(1);                                                     \
    }                                                              \
  } while(0)

const char *lto_saxpy = "                                        \n\
extern __device__ float compute(float a, float x, float y);     \n\
                                                                \n\
extern \"C\" __global__                                         \n\
void saxpy(float a, float *x, float *y, float *out, size_t n)   \n\
{                                                               \n\
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;           \n\
  if (tid < n) {                                                \n\
    out[tid] = compute(a, x[tid], y[tid]);                      \n\
  }                                                             \n\
}                                                               \n";
```

```
int main(int argc, char *argv[])
{
  size_t numBlocks = 32;
  size_t numThreads = 128;
  // Create an instance of nvrtcProgram with the code string.
  nvrtcProgram prog;
  NVRTC_SAFE_CALL(
    nvrtcCreateProgram(&prog,                          // prog
                       lto_saxpy,                      // buffer
                       "lto_saxpy.cu",                 // name
                       0,                              // numHeaders
                       NULL,                           // headers
                       NULL));                         // includeNames

  // specify that NVVM IR should be generated for LTO operation
  const char *opts[] = {"-dlto",
                        "--relocatable-device-code=true"};
  nvrtcResult compileResult = nvrtcCompileProgram(prog,  // prog
                                                  2,     // numOptions
                                                  opts); // options
  // Obtain compilation log from the program.
  size_t logSize;
  NVRTC_SAFE_CALL(nvrtcGetProgramLogSize(prog, &logSize));
  char *log = new char[logSize];
  NVRTC_SAFE_CALL(nvrtcGetProgramLog(prog, log));
  std::cout << log << '\n';
  delete[] log;
  if (compileResult != NVRTC_SUCCESS) {
    exit(1);
  }
  // Obtain generated NVVM IR from the program.
  size_t nvvmSize;
  NVRTC_SAFE_CALL(nvrtcGetNVVMSize(prog, &nvvmSize));
  char *nvvm = new char[nvvmSize];
  NVRTC_SAFE_CALL(nvrtcGetNVVM(prog, nvvm));
  // Destroy the program.
  NVRTC_SAFE_CALL(nvrtcDestroyProgram(&prog));

  // Load the generated NVVM IR and the NVVM IR generated offline
  // and link them together.
  CUdevice cuDevice;
  CUcontext context;
  CUlinkState linkState;
  CUmodule module;
  CUfunction kernel;
```

```
CUDA_SAFE_CALL(cuInit(0));
CUDA_SAFE_CALL(cuDeviceGet(&cuDevice, 0));
CUDA_SAFE_CALL(cuCtxCreate(&context, 0, cuDevice));
CUjit_option options[1];
void*        optionVals[1];

options[0] = CU_JIT_LTO;
optionVals[0] = (void *)1;
CUDA_SAFE_CALL(cuLinkCreate(1, options, optionVals, &linkState));

// NOTE: assumes "offline.fatbin" is in the current directory
// The fatbinary contains NVVM IR generated offline using nvcc
CUDA_SAFE_CALL(cuLinkAddFile(linkState, CU_JIT_INPUT_FATBINARY, "offline.fatbin",
                             0, 0, 0));
CUDA_SAFE_CALL(cuLinkAddData(linkState, CU_JIT_INPUT_NVVM,
                             (void *)nvvm, nvvmSize, "lto_online",
                             0, 0, 0));
size_t cubinSize;
void *cubin;

// The call to cuLinkComplete causes linker to link together the two
// NVVM IR modules (offline and online), do optimization on the linked NVVM IR,
// and generate cubin from it.
CUDA_SAFE_CALL(cuLinkComplete(linkState, &cubin, &cubinSize));
```

```
CUDA_SAFE_CALL(cuModuleLoadData(&module, cubin));
CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, "saxpy"));

// Generate input for execution, and create output buffers.
size_t n = NUM_THREADS * NUM_BLOCKS;
size_t bufferSize = n * sizeof(float);
float a = 5.1f;
float *hX = new float[n], *hY = new float[n], *hOut = new float[n];
for (size_t i = 0; i < n; ++i) {
  hX[i] = static_cast<float>(i);
  hY[i] = static_cast<float>(i * 2);
}
CUdeviceptr dX, dY, dOut;
CUDA_SAFE_CALL(cuMemAlloc(&dX, bufferSize));
CUDA_SAFE_CALL(cuMemAlloc(&dY, bufferSize));
CUDA_SAFE_CALL(cuMemAlloc(&dOut, bufferSize));
CUDA_SAFE_CALL(cuMemcpyHtoD(dX, hX, bufferSize));
CUDA_SAFE_CALL(cuMemcpyHtoD(dY, hY, bufferSize));
// Execute SAXPY.
void *args[] = { &a, &dX, &dY, &dOut, &n };
CUDA_SAFE_CALL(
  cuLaunchKernel(kernel,
                 NUM_BLOCKS, 1, 1,    // grid dim
                 NUM_THREADS, 1, 1,   // block dim
                 0, NULL,             // shared mem and stream
                 args, 0));           // arguments
CUDA_SAFE_CALL(cuCtxSynchronize());
// Retrieve and print output.
CUDA_SAFE_CALL(cuMemcpyDtoH(hOut, dOut, bufferSize));

for (size_t i = 0; i < n; ++i) {
  std::cout << a << " * " << hX[i] << " + " << hY[i]
            << " = " << hOut[i] << '\n';
}
// Release resources.
CUDA_SAFE_CALL(cuMemFree(dX));
CUDA_SAFE_CALL(cuMemFree(dY));
CUDA_SAFE_CALL(cuMemFree(dOut));
CUDA_SAFE_CALL(cuLinkDestroy(linkState));
CUDA_SAFE_CALL(cuModuleUnload(module));
```

```
    CUDA_SAFE_CALL(cuCtxDestroy(context));
    delete[] hX;
    delete[] hY;
    delete[] hOut;
    delete[] nvvm;
    return 0;
}
```

E.3. Build Instructions

Assuming the environment variable `CUDA_PATH` points to CUDA Toolkit installation directory, build this example as:

- Compile offline.cu to fatbinary containing NVVM IR (change `lto_52` to a different `lto_XX` architecture as appropriate).

  ```
  nvcc -arch lto_52 -rdc=true -fatbin offline.cu
  ```

- With NVRTC shared library:
  - Windows:

    ```
    cl.exe online.cpp /Feonline ^
          /I "%CUDA_PATH%\include" ^
          "%CUDA_PATH%"\lib\x64\nvrtc.lib "%CUDA_PATH%"\lib\x64\cuda.lib
    ```

  - Linux:

    ```
    g++ online.cpp -o online \
          -I $CUDA_PATH/include \
          -L $CUDA_PATH/lib64 \
          -lnvrtc -lcuda \
          -Wl,-rpath,$CUDA_PATH/lib64
    ```

- With NVRTC static library:
  - Windows:

    ```
    cl.exe online.cpp /Feonline  ^
          /I "%CUDA_PATH%"\include ^
          "%CUDA_PATH%"\lib\x64\nvrtc_static.lib ^
          "%CUDA_PATH%"\lib\x64\nvrtc-builtins_static.lib ^
          "%CUDA_PATH%"\lib\x64\nvptxcompiler_static.lib ^
          "%CUDA_PATH%"\lib\x64\cuda.lib user32.lib Ws2_32.lib
    ```

  - Linux:

    ```
    g++ online.cpp -o online \
          -I $CUDA_PATH/include \
          -L $CUDA_PATH/lib64 \
          -lnvrtc_static -lnvrtc-builtins_static -lnvptxcompiler_static -lcuda \
          -lpthread
    ```

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com